Buildroot Workshop

Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com

Thomas Petazzoni

- Embedded Linux engineer and trainer at Free Electrons since 2008
  - Embedded Linux development: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux training, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
  - `http://www.free-electrons.com`
- Major contributor to Buildroot, an open-source, simple and fast embedded Linux build system
- Speaker at Embedded Linux Conference, Embedded Linux Conference Europe, FOSDEM, Libre Software Meeting, etc.
- Living in Toulouse, south west of France

Agenda

- Install necessary tools and packages
- Get Buildroot
- Build a minimal system, and boot it on the target
- Customize the system
- Create new packages, one library, one application
- Generate an UBIFS image, and flash the system in the NAND Flash

Workshop elements

You will find the slides and other files needed for this workshop at:

`http://free-electrons.com/~thomas/lsm-tutorial/`

Our target platform

- IGEPv2 from ISEE
- DM3730 (ARM OMAP3) at 1 GHz
- 512 MB of RAM, 512 MB of Flash
- microSD, HDMI, audio, Ethernet, Bluetooth, Wifi

http://igep.es/products/processor-boards/igepv2-board

Tools to interact with the target

We'll need:

- A **terminal emulator** program to interact with the target over the serial port

```
apt-get install picocom
```

- A **TFTP server** to transfer the kernel image to the target

```
apt-get install tftpd-hpa
```

- A **NFS server** to mount the root filesystem over the network

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

Of course, adapt those instructions if you're not using a Debian-derived distribution.

Buildroot dependencies

Even though Buildroot builds most of the tools it needs, it still requires a few dependencies on the build system:

```
apt-get install \
  build-essential gawk bison flex gettext \
  texinfo patch gzip bzip2 perl tar wget  \
  cpio python unzip rsync
```

Getting Buildroot

- Tarballs are available for major versions, but since one generally needs to make changes to Buildroot, using Git is recommended
- Clone the repository

```
git clone git://git.busybox.net/buildroot
```

- Create a branch starting from a stable release

```
git branch workshop 2012.05
```

- Switch to this branch

```
git checkout workshop
```

Initial configuration

Run `make menuconfig`

- **Target architecture**: *ARM Little Endian*
- **Target architecture variant**: *Cortex-A8*
- **Toolchain**
  - Toolchain type: *External toolchain*
  - Toolchain: *CodeSourcery 2011.09*
- **System configuration**
  - /dev management: *Dynamic using devtmpfs only*
  - Port to run a getty on: *ttyO2*
- **Package selection for the target**
  - Only Busybox is selected. This is fine for now.
- **Kernel**
  - Kernel version: *Custom version*
  - Kernel version: *3.2*
  - Custom kernel patches: `board/lsm/demo/linux-3.2-patches/`
  - Defconfig name: `omap2plus`

Kernel patches

For this board to work with kernel 3.2, we need two patches to enable NAND support.

```
mkdir -p board/lsm/demo/linux-3.2-patches/

cd board/lsm/demo/linux-3.2-patches/

wget http://free-electrons.com/~thomas/lsm-tutorial/
     linux-3.2-arm-omap3-igep0020-add-support-for-micron-nand-flash.patch

wget http://free-electrons.com/~thomas/lsm-tutorial/
     linux-3.2-omap2-make-board-onenand-init-visible-to-board-code.patch
```

Running the build

Let's run the build, and keep a log from it:

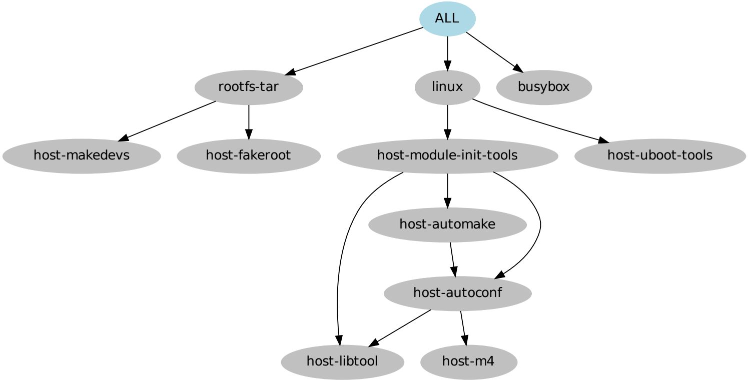
```
make 2>&1 | tee logfile
```

Build results

In the `output` directory, we have:

- `build`, with one subdirectory per package that has been built. The source code of the packages is extracted here, and they are compiled here.
- `host`, where host tools are installed. The external toolchain has been extracted in `host/opt`, in `host/usr/bin`, you have a few host tools, and in `host/usr/arm-unknown-linux-gnueabi/sysroot` you have the *sysroot*
- `staging`, symbolic link to the *sysroot*
- `target`, where the target libraries and applications are installed.
- `toolchain`, empty because we're using an external toolchain
- `images`, which contains the root filesystem as a tarball, and the kernel image. Look at the root filesystem size (it is uncompressed!)

A look at the dependencies



Generated with:

```
./support/scripts/graph-depends > deps.dot
dot -Tpdf -o deps.pdf deps.dot
```

Preparing to boot

1. Extract the root filesystem:

```
mkdir /tmp/rootfs/
sudo tar -C /tmp/rootfs/ output/images/rootfs.tar
```

2. Export it over NFS, add the following line to `/etc/exports`:

```
/tmp/rootfs/ 192.168.42.2(rw,no_root_squash,no_subtree_check)
```

3. And restart the NFS server:

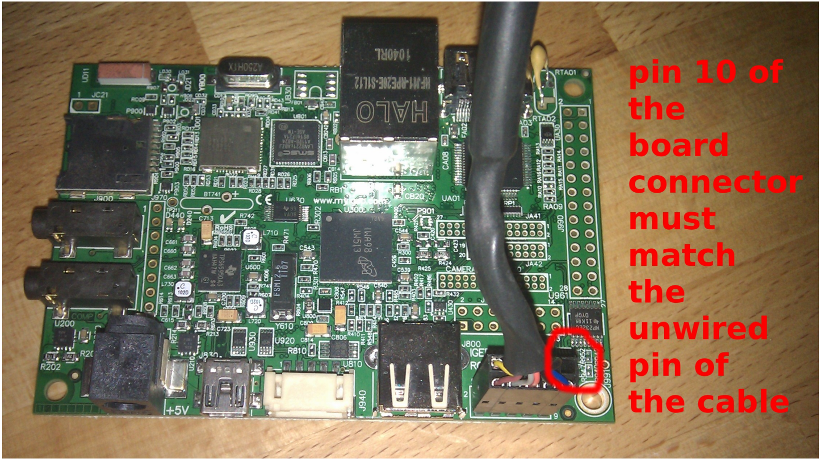
```
sudo /etc/init.d/nfs-kernel-server restart
```

4. Copy the kernel image to the TFTP exported directory:

```
sudo output/images/uImage /var/lib/tftpboot/
```

5. Configure your system to assign the `192.168.42.1` static IP address to the USB-ethernet interface (using Network Manager or `ifconfig`)

Connecting the serial cable to the board



Preparing to boot, on the board

1. Start a serial emulator program:

```
picocom -b 115200 /dev/ttyUSB0
```

2. When the board boots, interrupt in U-Boot during `Hit any key to stop autoboot:` by pressing a key, and enter the following commands:

```
setenv ipaddr 192.168.42.2
setenv serverip 192.168.42.1
setenv bootcmd 'tftp 80000000 uImage; bootm'
setenv bootargs 'console=ttyO2,115200 root=/dev/nfs ip=192.168.42.2 \
                 nfsroot=192.168.42.1:/tmp/rootfs'
saveenv
reset
```

3. The system should boot automatically.

The system

- Login as root, no password will be prompted.
- Explore the system. You'll see that it is fairly minimal. We have:
  - Busybox installed (in `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`)
  - The C librari in `/lib`
  - A bunch of configuration files and init scripts in `/etc/`
  - `/proc` and `/sys` mounted
- In the running processes, we only have the usual kernel threads, the `init` process, a shell, and the `syslogd`/`klogd` daemons for login
- For Buildroot, it is important that the **default is small**

Customize the kernel configuration

Let's learn now how to customize the kernel configuration from Buildroot.

1. Run `make linux-menuconfig`
2. In `Device Drivers` → `LED Support`, enable as static options (with a `*`, not a `M`):
   - `LED Class support`
   - `LED Support for GPIO connected LEDs`
   - `LED Trigger support`
   - `LED Timer trigger`
   - `LED heartbeat trigger`
3. Rebuild by running `make`
4. Copy your kernel image to the TFTP directory:

```
sudo cp output/images/uImage /var/lib/tftpboot/
```

Test the LEDs

Reboot your system, and try the following commands:

```
cd /sys/class/leds/gpio-led:green:d0
echo 255 > brightness
echo 0 > brightness
echo timer > trigger
echo heartbeat > trigger
echo none > trigger
```

Saving the kernel configuration

Our kernel configuration change has only been made to `output/build/linux-3.2/.configg`, which will be removed if we do a `make clean`, so let's save our kernel configuration changes.

1. Generate a minimal defconfig for our kernel configuration:

```
make linux-savedefconfig
```

2. Store in our project-specific directory

```
mv output/build/linux-3.2/defconfig board/lsm/demo/linux-3.2.config
```

3. Adjust the Buildroot configuration:

```
make menuconfig
```

- `Linux Kernel` → `Kernel configuration` → `Using a custom configuration file`
- `Configuration file path`: `board/lsm/demo/linux-3.2.config`

Enabling a new package: Dropbear

Let's enable a new package, the lightweight SSH client/server Dropbear.

```
make menuconfig
```

```
Package selection for the target
 -> Networking applications
    -> dropbear
```

Set root password

1. Create `board/lsm/demo/post-build.sh` with:

```
#!/bin/sh
TARGETDIR=$1

# Set the root password to 'demo'
sed -i 's%^root::%root:pT41pCOBIPj3Q:%' $TARGETDIR/etc/shadow

# Disable login with the 'default' user
sed -i 's/^default::/default:*:/' $TARGETDIR/etc/shadow
```

2. Add executable permissions to the script
3. In `make menuconfig`, `System configuration` → `Custom script to run before creating filesystem images` set `board/lsm/demo/post-build.sh`.

Testing SSH

1. Run `make` to rebuild your system
2. Re-extract the root filesystem tarball

```
sudo tar -C /tmp/rootfs/ output/images/rootfs.tar
```

3. Boot your system, you should see Dropbear being started
4. From your machine, log into your board through SSH:

```
ssh root@192.168.42.2
```

Adding new packages

We'll now see how to add new packages, by taking the example of two dummy packages:

- **libfoo**, a dummy library that implements just a `int foo_add(int a, int b);` function.
  Available at `http://free-electrons.com/~thomas/lsm-tutorial/libfoo-0.1.tar.gz`
- **foo**, a dummy application that uses **libfoo**
  Available at `http://free-electrons.com/~thomas/lsm-tutorial/foo-0.1.tar.gz`

libfoo: Config.in

Create the `package/libfoo` directory, and edit `package/libfoo/Config.in`:

```
config BR2_PACKAGE_LIBFOO
        bool "libfoo"
        help
          libfoo is a wonderful package.

          http://free-electrons.com/~thomas/lsm-tutorial/
```

Then, edit `package/Config.in`, and under `Libraries` → `Other`, add:

```
source "package/libfoo/Config.in"
```

libfoo: package/libfoo/libfoo.mk

Download the package tarball, and quickly study its build system. It uses the traditional `./configure; make; make install` mechanism, using the *autotools*. We'll use the `AUTOTARGETS` infrastructure for our package.

```
LIBFOO_VERSION = 0.1
LIBFOO_SITE    = http://free-electrons.com/~thomas/lsm-tutorial/

$(eval $(call AUTOTARGETS))
```

`LIBFOO_SOURCE` could be defined to `libfoo-$(LIBFOO_VERSION).tar.gz`, but since this is the default, there's no need to mention it.

libfoo: first test

1. Enable your package in `make menuconfig`
2. Run `make`
3. Your library is correctly present in `output/target/usr/lib/libfoo.so.0.1`
4. But the header files, and other developments files, are not present in `output/staging/usr/include/libfoo`

libfoo: installation to staging

For libraries, we need to explicitely tell Buildroot to install them to the *staging* directory.

```
LIBFOO_VERSION = 0.1
LIBFOO_SITE    = http://free-electrons.com/~thomas/lsm-tutorial/

LIBFOO_INSTALL_STAGING = YES

$(eval $(call AUTOTARGETS))
```

libfoo: second testing

1. `make libfoo-dirclean`
2. `make`
3. Check in `output/staging/usr/include/libfoo` that the header file is installed.
4. You should also have the static version of the library in `output/staging/usr/lib/` and the *pkgconfig* file `foo.pc` in `output/staging/usr/lib/pkgconfig`

libfoo: adding a configuration option (1/2)

Our wonderful **libfoo** library supports one `./configure` option: `--enable-debug`. Let's add a new Buildroot option for it. In `package/libfoo/Config.in`, add:

```
config BR2_PACKAGE_LIBFOO_DEBUG
        bool "Enable debugging support"
        depends on BR2_PACKAGE_LIBFOO
        help
          Enable debugging support in libfoo.
```

libfoo: adding a configuration option (2/2)

In the `package/libfoo/libfoo.mk`:

```
ifeq ($(BR2_PACKAGE_LIBFOO_DEBUG),y)
LIBFOO_CONF_OPT += --enable-debug
endif
```

- In `menuconfig`, enable your new option
- Run `make libfoo-dirclean` to clean the package and force its rebuild
- Run `make`

foo: package/foo/Config.in

Now, let's create a package for the application. First the package/foo/Config.in file:

```
config BR2_PACKAGE_FOO
        bool "foo"
        select BR2_PACKAGE_LIBFOO
        help
          Wonderful foo application

          http://free-electrons.com/~thomas/lsm-tutorial/
```

And source it from the Miscellaneous section of package/Config.in:

```
source "package/foo/Config.in"
```

foo: `package/foo/foo.mk`

Before writing the `foo.mk`, let's download `http://free-electrons.com/~thomas/lsm-tutorial/foo-0.1.tar.gz` and look at its build system:

- It is based on a manual *Makefile*, so we will have to use the `GENTARGETS` infrastructure and not the `AUTOTARGETS` one
- It uses `pkg-config` to find the library `foo`. So we will have to depend on `libfoo` and `host-pkg-config`
- For the build, we will have to pass `CC`, `CFLAGS`, `LDFLAGS`, etc. with appropriate values. To do this, we'll use the Buildroot variable `TARGET_CONFIGURE_OPTS`
- For the installation, we'll have to pass value for the `DESTDIR` and `prefix` variables

foo: package/foo/foo.mk

```
FOO_VERSION = 0.1
FOO_SITE    = http://free-electrons.com/~thomas/lsm-tutorial/

FOO_DEPENDENCIES += libfoo host-pkg-config

define FOO_BUILD_CMDS
	$(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)
endef

define FOO_INSTALL_TARGET_CMDS
	$(MAKE) $(TARGET_CONFIGURE_OPTS) \
		DESTDIR=$(TARGET_DIR) \
		prefix=/usr \
		install -C $(@D)
endef

$(eval $(call GENTARGETS))
```

Test `foo`

- Enable the `foo` package in `menuconfig`
- Build your system with `make`
- Re-extract the root filesystem tarball to `/tmp/rootfs/`
- Reboot your system, and test the new `foo` application

Save our configuration

In order to make our configuration usable by others, we'll create a *defconfig* from it:

```
make savedefconfig
mv defconfig configs/lsm_demo_defconfig
```

Now, users of your Buildroot can simply do:

```
make lsm_demo_defconfig
make
```

To rebuild an identical environment from scratch.

Booting from flash

We know want to store the kernel and root filesystem in the NAND flash. To do this, we will:

1. Add a custom `/etc/network/interfaces` file to the filesystem in order to not depend on the `ip=` kernel parameter
2. Configure Buildroot to generate an UBIFS/UBI image for the root filesystem
3. Adjust the U-Boot configuration and kernel arguments to boot from NAND flash.

Custom /etc/network/interfaces

1. Create the board/lsm/demo/rootfs-additions directory, which will be an overlay of our filesystem
2. In our post-build.sh script, add:

```
# Copy the rootfs additions
cp -a board/lsm/demo/rootfs-additions/* $TARGETDIR/
```

3. Create the board/lsm/demo/rootfs-additions/etc/network/interfaces file, with:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 192.168.42.2
        netmask 255.255.255.0
```

Generate an UBIFS/UBI image

In `menuconfig`

1. Go in the `Filesystem images` menu
2. Enable `ubifs root filesystem`
3. Enable `Embed into an UBI image`

Then, rebuild with `make`, and copy `output/images/rootfs.ubi` to `/var/lib/tftpboot`.

Adjust U-Boot configuration

We will adjust the U-Boot environment variables.

- Kernel command line

```
setenv bootargs 'console=tty02,115200
   mtdparts=omap2-nand.0:512k(xloader),1536k(uboot),512k(env),4m(kernel),16m(rootfs)
   ubi.mtd=4 root=ubi0:rootfs rootfstype=ubifs'
```

- At boot time, load the kernel from NAND

```
setenv bootcmd 'nboot 80000000 0 280000; bootm'
```

- Helper script to flash the kernel in NAND

```
setenv flashkernel 'tftp 80000000 uImage;
  nand erase 0x280000 0x400000;
  nand write 0x80000000 0x280000 0x400000'
```

- Helper script to flash the rootfs in NAND

```
setenv flashrootfs 'tftp 80000000 rootfs.ubi;
  nand erase 0x680000 0x1000000;
  nand write 0x80000000 0x680000 ${filesize}'
```

Adjust U-Boot configuration

We will adjust the U-Boot environment variables.

- Helper script to flash the kernel and rootfs

```
setenv flashall 'run flashkernel; run flashrootfs'
```

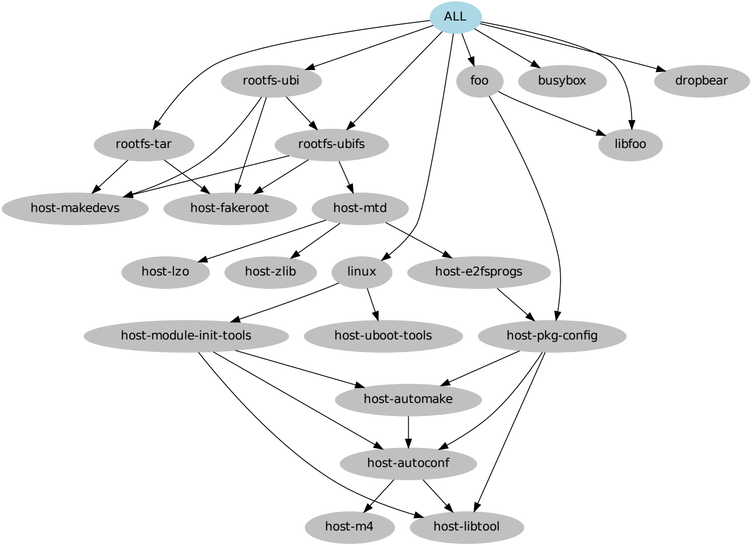
- Reflash

```
run flashall
```

- And reboot to test the system

```
reset
```

A final look at the dependencies



Thanks for attending, have fun with Buildroot!

Thomas Petazzoni

thomas.petazzoni@free-electrons.com

 PDF and sources will be available on
http://free-electrons.com/pub/conferences/2012/lsm/