Getting started with Buildroot

Trevor Woerner
trevor@toganlabs.com

Corrections, suggestions, contributions and translations are welcome!

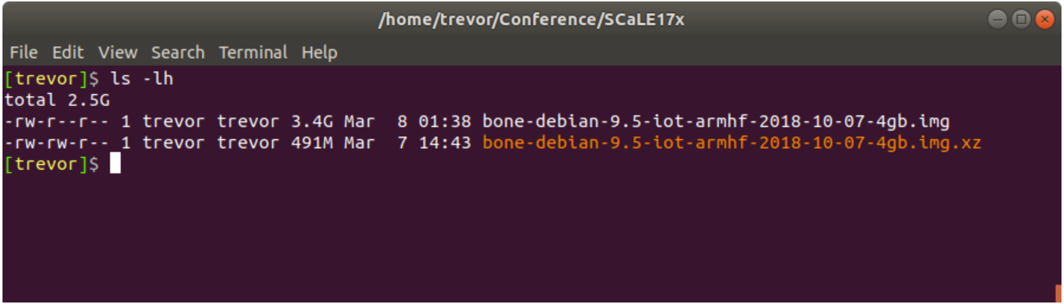
Trevor Woerner

- Senior Software Developer at Togán Labs
  - Embedded Linux Development and Consulting
  - Specialize in **OpenEmbedded/Yocto**
  - Strong open-source focus

Thomas Petazzoni

- Initially created this presentation
- CTO and Embedded Linux engineer at Bootlin
- **major** contributor to buildroot
  - one of 3 people with commit access to buildroot
- Strong open-source focus

Using an embedded Linux system



- the first thing you did at the very start of the labs
- has allowed you to do everything you've done with the board
- wrote one **HUGE** image to the SDcard – *3.4GB*!!
- it took my laptop over 13 minutes

Using an embedded Linux system

- what does the SDcard look like post-flashing?

```
/home/trevor/Conference/SCaLE17x
File  Edit  View  Search  Terminal  Help
[root]# fdisk -l /dev/sdb
Disk /dev/sdb: 14.6 GiB, 15665725440 bytes, 30597120 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x7aac34a0

Device     Boot Start     End Sectors  Size Id Type
/dev/sdb1  *     8192 6963199 6955008  3.3G 83 Linux
[root]# 
```

Using an embedded Linux system

- one big file was flashed, but the card now has partitions and lots of files
- works cross-platform (x86_64 host, ARM target)

/home/trevor/Conference/SCaLE17x

File Edit View Search Terminal Help

```
Disklabel type: dos
Disk identifier: 0x7aac34a0

Device     Boot Start     End Sectors  Size Id Type
/dev/sdb1  *     8192 6963199 6955008  3.3G 83 Linux
[root]# mkdir mnt
[root]# mount /dev/sdb1 mnt
[root]# ls mnt
bbb-uEnv.txt  boot  etc   ID.txt  lost+found  mnt           opt   root  sbin  sys  usr
bin           dev   home  lib     media       nfs-uEnv.txt  proc  run   srv   tmp  var
[root]# 
```

Using an embedded Linux system

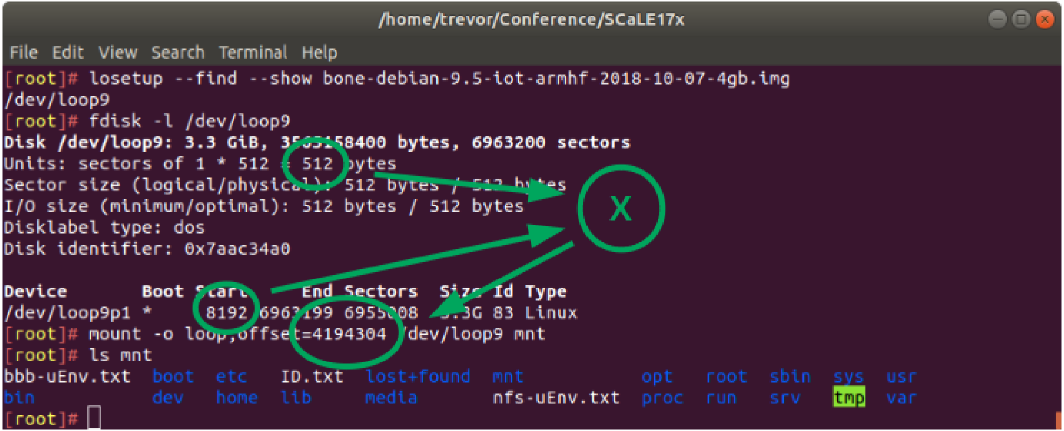
- it's all there in the `*.img` file

```
/home/trevor/Conference/SCaLE17x
File  Edit  View  Search  Terminal  Help
[root]# losetup --find --show bone-debian-9.5-iot-armhf-2018-10-07-4gb.img
/dev/loop9
[root]# fdisk -l /dev/loop9
Disk /dev/loop9: 3.3 GiB, 3565158400 bytes, 6963200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x7aac34a0

Device       Boot Start     End Sectors  Size Id Type
/dev/loop9p1 *     8192 6963199 6955008  3.3G 83 Linux
[root]# mount -o loop,offset=4194304 /dev/loop9 mnt
[root]# ls mnt
bbb-uEnv.txt  boot  etc   ID.txt  lost+found  mnt           opt   root  sbin  sys  usr
bin           dev   home  lib     media       nfs-uEnv.txt  proc  run   srv   tmp  var
[root]# 
```

Using an embedded Linux system

- it's all there in the `*.img` file



Can I build an embedded Linux system?

- with my own contents
  - bootloader
  - kernel
  - init
  - apps ($\rightarrow$ dependencies)
- to target multiple devices
- within a certain size (smaller)
- containing only what a production device needs
- with the latest fixes
- built up from nothing, not simply imaging a disk (horse/cart)

Can I build an embedded Linux system?

YES!

Can I build an embedded Linux system?

Caveats:

- cross-compiling/toolchain
- find/get the sources (bzr, cvs, git, hg, scp, svn, wget, ...) can't hope for pre-compiled
- checksum (md5, sha256, sha512, ...)
- dependencies
- unpack (gzip, bzip2, xz, zip, ...)
- patch
- configure
- build (make, cmake, ant, maven, waf, ninja, gyp, meson, ...)
- device-specific tweaks/tricks
- lots of choice, hard to get right
- one doesn't slap together a cohesive, working set, by accident

Can I build an embedded Linux system?

problems:

- there is no "**one correct way**" to write (open-source) software
  - language
  - repository system
  - build system
  - location
  - licensing
- not enough developers consider cross-compilation

solution:

- use an embedded Linux build system
- **PLEASE** don't "roll your own"

Building an embedded Linux system

Pre-built binary Linux distributions

+ Readily available
- Large, usually 100+ MB ($\rightarrow$ GB!)
- Not available for all architectures/devices
- Not easy to customize
- Generally require native compilation

Building an embedded Linux system

Manual
system
building

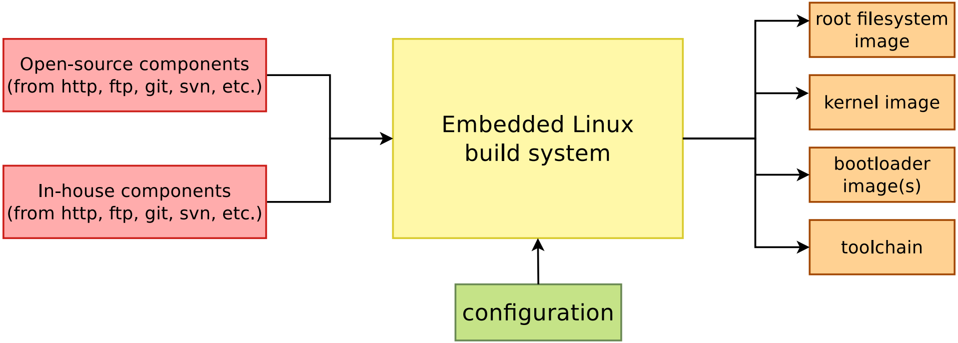
+ Smaller and flexible
- Very hard to handle cross-compilation and dependencies
- Not reproducible
- No benefit from other people's work

Building an embedded Linux system

Embedded
Linux
build systems

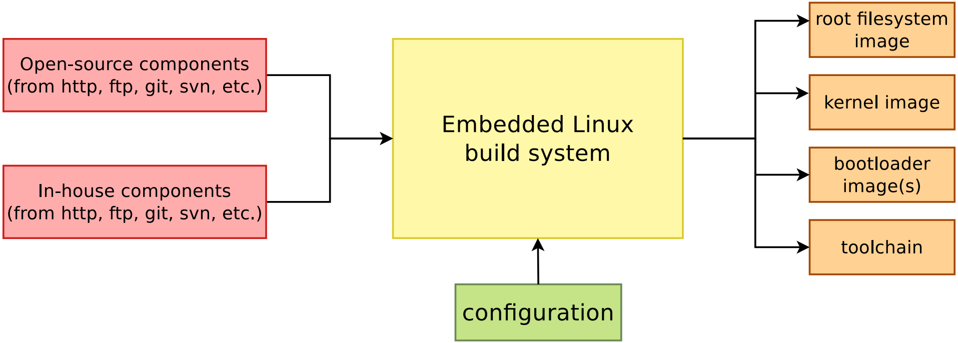
+ Small and flexible
+ Reproducible, handles cross-compilation and dependencies
+ Available for virtually all architectures
- One tool to learn
- Build time

Embedded Linux build system: principle



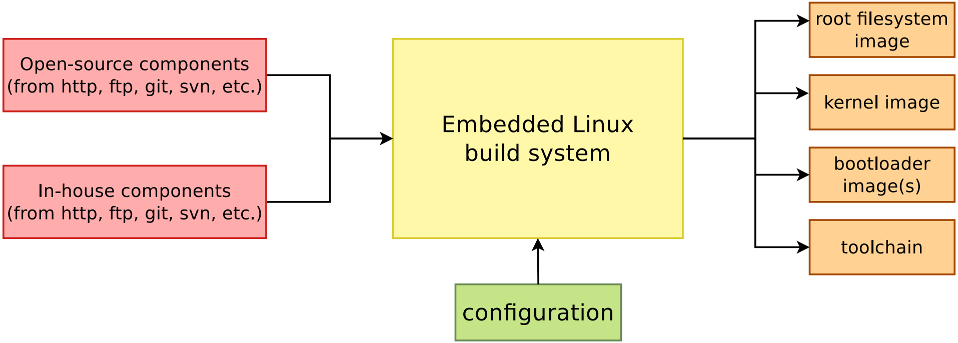
- Building from source $\rightarrow$ lot of flexibility

Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines

Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- **Recipes** for building components → easy

Recipes, meta-data

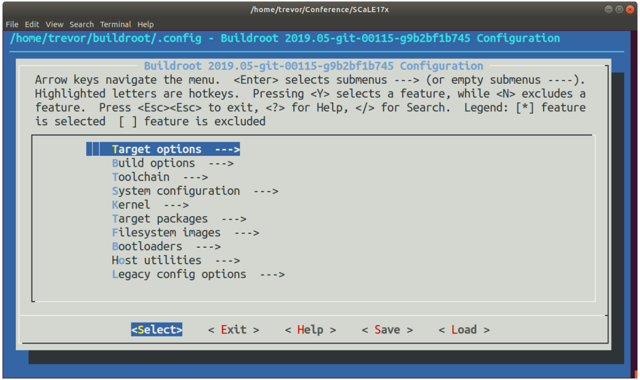
- all embedded Linux build systems have:
  - a tool for building the image (i.e. `make`)
  - the **recipes** or **meta-data** describing how to handle each component
    - where it's located (github, gitlab, bitbucket, ...)
    - how to fetch it (wget, svn, git, ...)
    - patches
    - how to build (make, meson, cmake, ...)
  - configuration

Buildroot at a glance

- Is an **embedded Linux build system**, builds from source:
  - cross-compilation toolchain
  - root filesystem with many libraries/applications, cross-built
  - kernel and bootloader images
- **Fast**, simple root filesystem in minutes
- **Easy** to use and understand: kconfig and make
- **Small** root filesystem, default 2 MB
- Roughly **2170 packages** available
- Generates filesystem images, not a distribution
- Vendor neutral
- Active community, stable releases every 3 months
- Started in 2001, oldest still maintained build system
- `http://buildroot.org`

Getting started

```
$ git clone git://git.busybox.net/buildroot
$ cd buildroot
$ make menuconfig
```



1. Target architecture

- Architecture
  ARC, ARM, Aarch64, csky, m68k, Microblaze, MIPS(64), NIOS II, OpenRISC, PowerPC(64), RISC-V, SuperH, SPARC(64), x86, x86_64, Xtensa
- Specific processor (variant) / Floating-point strategy
- ABI

Buildroot configuration

1. Target architecture
2. Build options

- Download directory
- Number of parallel jobs
- Use of *ccache*
- Shared or static libraries
- etc.

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain

- Buildroot toolchain
  - Buildroot builds the toolchain
  - uClibc-ng, glibc, musl
- External toolchain
  - Uses a pre-built toolchain
  - Profiles for existing popular toolchains Linaro, Sourcery CodeBench, etc.
  - Custom toolchains

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration

- Init system to use: Busybox, Sysvinit, Systemd
- `/dev` management solution: static, devtmpfs, mdev, udev
- Hostname, password, getty terminal, etc.
- Root filesystem overlay
- Custom post build and post image scripts
- etc.

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration
5. Kernel

- Kernel source (stable version, Git tree, patches)
- Kernel configuration
- Support for kernel extensions: RTAI, Xenomai, aufs, etc.

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration
5. Kernel
6. Target packages

- Roughly 2170 packages
- Qt5, X.org, Gtk, EFL
- GStreamer, ffmpeg
- Python, Perl, Ruby, Lua, Erlang
- Samba, OpenSSL, OpenSSH, dropbear, lighttpd
- OpenGL support for various platforms
- And many, many more libraries and utilities

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration
5. Kernel
6. Target packages
7. Filesystem images

- Major filesystem formats supported
  - axfs
  - btrfs
  - cloop
  - cpio, for kernel initramfs
  - cramfs
  - ext2/3/4
  - f2fs
  - jffs2
  - romfs
  - squashfs
  - tar
  - ubifs
  - yaffs2

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration
5. Kernel
6. Target packages
7. Filesystem images
8. Bootloaders

- Barebox
- Gummiboot
- Grub2
- shim
- Syslinux
- U-Boot
- and more platform-specific bootloaders: imx-bootlets, at91bootstrap, etc.

Buildroot configuration

1. Target architecture
2. Build options
3. Toolchain
4. System configuration
5. Kernel
6. Target packages
7. Filesystem images
8. Bootloaders
9. Host utilities

- Allows to build some native tools, useful for development.

Building and using

- To start the build: `make`
- Results in `output/images`:
  - `rootfs.ext4`, root filesystem in ext4 format
  - `zImage`, Linux kernel image
  - `am335x-pocketbeagle.dtb`, Linux kernel Device Tree blob
  - `u-boot.img`, U-Boot bootloader image
  - `MLO`, U-Boot bootloader image
- Ready to be flashed on your embedded system.

Exploring the build output

- All the output produced by Buildroot is stored in `output/`
- Can be customized using `O=` for out-of-tree build
- `output/` contains
  - `output/build`, with one sub-directory for the source code of each component
  - `output/host`, which contains all native utilities needed for the build, including the cross-compiler
  - `output/host/<tuple>/sysroot`, which contains all the headers and libraries built for the target
  - `output/target`, which contains *almost* the target root filesystem
  - `output/images`, the final images

Summarized build process

1. Check core dependencies
2. For each selected package, after taking care of its dependencies: download, extract, patch, configure, build, install
   - To `target/` for target apps and libs
   - To `host/<tuple>/sysroot` for target libs
   - To `host/` for native apps and libs
   - Filesystem skeleton and toolchain are handled as regular packages
3. Copy rootfs overlay
4. Call post build scripts
5. Generate the root filesystem image
6. Call post image scripts

Customizing the build

Besides the existing packages and options, there are multiple ways to customize the generated root filesystem:

- Create custom *post-build* and/or *post-image* scripts
- Use a *root filesystem overlay*, recommended to add all your config files
- Add your own packages

Adding a new package: Config.in

package/libmicrohttpd/Config.in

```
config BR2_PACKAGE_LIBMICROHTTPD
        bool "libmicrohttpd"
        depends on BR2_TOOLCHAIN_HAS_THREADS
        help
          GNU libmicrohttpd is a small C library that makes it easy to
          run an HTTP server as part of another application.

          http://www.gnu.org/software/libmicrohttpd/

comment "libmicrohttpd needs a toolchain w/ threads"
        depends on !BR2_TOOLCHAIN_HAS_THREADS
```

package/Config.in

```
[...]
source "package/libmicrohttpd/Config.in"
[...]
```

Adding a new package: <pkg>.mk, <pkg>.hash

package/libmicrohttpd/libmicrohttpd.mk

```
LIBMICROHTTPD_VERSION = 0.9.59
LIBMICROHTTPD_SITE = $(BR2_GNU_MIRROR)/libmicrohttpd
LIBMICROHTTPD_LICENSE = LGPL-2.1+
LIBMICROHTTPD_LICENSE_FILES = COPYING
LIBMICROHTTPD_INSTALL_STAGING = YES
LIBMICROHTTPD_CONF_OPT = --disable-curl --disable-examples

$(eval $(autotools-package))
```

package/libmicrohttpd/libmicrohttpd.hash

```
# Locally calculated
sha256 9b9ccd7d0b11b0e17...  libmicrohttpd-0.9.59.tar.gz
sha256 70e12e2a60151b9ed...  COPYING
```

Adding a new package: infrastructures

- In order to factorize similar behavior between packages using the same build mechanism, Buildroot has **package infrastructures**
  - `autotools-package` for autoconf/automake based packages
  - `cmake-package` for CMake based packages
  - `python-package` for Python Distutils and Setuptools based packages
  - `generic-package` for non-standard build systems
  - And more: `luarocks-package`, `perl-package`, `rebar-package`, `kconfig-package`, etc.

Defconfigs

- Pre-defined configurations for popular platforms
- They build a *minimal* system for the platform
- `make <foobar>_defconfig` to load one of them
- Some of the configs
  - RasberryPi
  - BeagleBone
  - CubieBoard
  - PandaBoard
  - Many Atmel development boards
  - Several Freescale i.MX6 boards
  - Many Qemu configurations
  - and more...
- `make list-defconfigs` for the full list

Buildroot design principles

- **Cross-compilation only**: no support for doing development on the target.
- **No package management system**: Buildroot doesn't generate a distribution, but a firmware
- **Don't be smart**: if you do a change in the configuration and restarts the build, Buildroot doesn't try to be smart. Only a full rebuild will guarantee the correct result.

Documentation and support

- Extensive manual: `https://buildroot.org/downloads/manual/manual.html`
- 3-day training course, with freely available materials: `https://bootlin.com/training/buildroot/`
- Mailing list: `http://lists.busybox.net/pipermail/buildroot/`
- IRC channel: `buildroot` on Freenode