

Buildroot

BASC2020 seminar

Giacomo Longo

University of Genoa

Q4 2020



Official website: <https://buildroot.org>

- ▶ Born in 2005
- ▶ Entirely based on **makefiles** and **kconfig**
- ▶ Only one goal: *producing root file system images for 100% custom Linux systems*

BuildRoot users

The most prominent users of BuildRoot are using it for building:

- ▶ IoT devices
- ▶ Automated factory controllers
- ▶ Point of sale devices
- ▶ Car multimedia units

Why BuildRoot

- ▶ Each buildroot is a 100% custom Linux "mini-distro"
- ▶ Buildroot images can be less than 100MB or even 10MB
- ▶ Complete customization of target architecture and build flags
- ▶ Multiple compiler / libc / system layout choices
- ▶ Updated every 3 months current version is **2020.11-rc1**
- ▶ Easily extendable

Why BuildRoot: architecture support

≈ 20 architectures supported

- ▶ ARC LE & BE
- ▶ **ARM** LE & BE
- ▶ AArch64 LE & BE
- ▶ csky
- ▶ **i386**
- ▶ Microblaze AXI & Non-AXI
- ▶ MIPS LE & BE
- ▶ MIPS64 LE & BE
- ▶ nds32
- ▶ Nios II
- ▶ PowerPC
- ▶ PowerPC64 LE & BE
- ▶ RISC-V
- ▶ SuperH
- ▶ SPARC
- ▶ **x86_64**
- ▶ Xtensa

The BuildRoot process

What the user sees

1. Create a configuration file
2. Start the build
3. Flash the image on the device

What BuildRoot does

1. Build a cross compiler on our machine
2. Resolve the configuration dependencies
3. Compile from source the requested packages
4. Assemble an image

Prerequisites

Packages for an ARM BuildRoot

Ubuntu 20.04

```
sudo apt-get update
sudo apt-get install -y \
    curl tar \
    make \
    gcc g++ \
    libncurses-dev libssl-dev \
    qemu-user-static \
    qemu-system-arm
```

Others

Binaries needed

Downloaders curl & wget

Extractor tar

Compilers gcc & g++

Libraries ncurses & openssl

Execution QEMU system for
ARM & QEMU static

Preparing our BuildRoot working directory

1. Clone the repository at
<https://github.com/gabibbo97/basc-buildroot>
2. Enter the directory
3. Run `sh ./seminar-scripts/get-buildroot.sh`

Please use the provided script

The script downloads BuildRoot 2020.11-rc1 but also applies two required patches that we need for today's seminar

Creating an ARM cross compiler

Initial setup

1. `cd buildroot-2020.11-rc1`
2. `cp ../scripts/gef-python.sh ./gef-python.sh`
3. `chmod +x *.sh`
4. `make distclean`
5. `make defconfig`

Creating an ARM cross compiler

Configuration options: 1/2

`make menuconfig`

- ▶ Target options

- ▶ Target Architecture = ARM (little endian)
- ▶ Target Architecture Variant = cortex-A7
- ▶ Floating point strategy = VFPv4-D16

- ▶ Build options

- ▶ ☒ Enable compiler cache
- ▶ ☒ build packages with debugging symbols
- ▶ gcc debug level = debug level 3
- ▶ ☐ strip target binaries
- ▶ gcc optimization level = optimize for debugging

Creating an ARM cross compiler

Configuration options: 2/2

- ▶ Toolchain
 - ▶ C library = glibc
 - ▶ ☒ Enable C++ support
 - ▶ ☒ Build cross gdb for the host
 - ▶ ☒ TUI support
 - ▶ Python support = Python3
 - ▶ GDB debugger Version = gdb 9.2.x
- ▶ System configuration
 - ▶ Custom scripts to run before creating filesystem images = `./gef-python.sh`
- ▶ Filesystem images
 - ▶ ☐ tar the root filesystem
- ▶ Host utilities
 - ▶ host python3
 - ▶ ssl

Creating an ARM cross compiler

Performing the build

1. Save the configuration to the default `.config` path
2. Download sources with `make source`
3. Start the build with `make sdk`

Creating an ARM root filesystem

Initial setup

1. `cd buildroot-2020.11-rc1`
2. `cp ../scripts/gef-python.sh ./gef-python.sh`
3. `chmod +x *.sh`
4. `make distclean`
5. `make defconfig`

Creating an ARM root filesystem

Configuration options: 1/3

`make menuconfig`

- ▶ Target options

- ▶ Target Architecture = ARM (little endian)
- ▶ Target Architecture Variant = cortex-A7
- ▶ Floating point strategy = VFPv4-D16

- ▶ Build options

- ▶ ☒ Enable compiler cache
- ▶ ☒ build packages with debugging symbols
- ▶ gcc debug level = debug level 3
- ▶ ☐ strip target binaries
- ▶ gcc optimization level = optimize for debugging

Creating an ARM root filesystem

Configuration options: 2/3

- ▶ Toolchain
 - ▶ C library = glibc
 - ▶ ☒ Enable C++ support
 - ▶ ☒ Build cross gdb for the host
 - ▶ ☒ TUI support
 - ▶ Python support = Python3
 - ▶ GDB debugger Version = gdb 9.2.x
- ▶ System configuration
 - ▶ Custom scripts to run before creating filesystem images = `./gef-python.sh`
- ▶ Target packages
 - ▶ Debugging, profiling and benchmark
 - ▶ ☒ gdb
 - ▶ ☒ full debugger
 - ▶ ☒ gdbserver
 - ▶ ☒ TUI support

Creating an ARM root filesystem

Configuration options: 3/3

- ▶ Filesystem images
 - ▶ ☒ tar the root filesystem
- ▶ Host utilities
 - ▶ host python3
 - ▶ ssl

Creating an ARM root filesystem

Performing the build

1. Save the configuration to the default `.config` path
2. Download sources with `make source`
3. Start the build with `make`

1. `cd buildroot-2020.11-rc1`
2. `cp ../kconfigs/virtio.kconfig ./virtio.kconfig`
3. `cp ../scripts/gef-python.sh ./gef-python.sh`
4. `cp ../scripts/enable-ssh-root-login.sh
./enable-ssh-root-login.sh`
5. `chmod +x *.sh`
6. `make distclean`
7. `make defconfig`

Creating a bootable ARM root filesystem

Configuration options: 1/3

`make menuconfig`

- ▶ Target options

- ▶ Target Architecture = ARM (little endian)
- ▶ Target Architecture Variant = cortex-A7
- ▶ Floating point strategy = VFPv4-D16

- ▶ Build options

- ▶ ☒ Enable compiler cache
- ▶ ☒ build packages with debugging symbols
- ▶ gcc debug level = debug level 3
- ▶ ☐ strip target binaries
- ▶ gcc optimization level = optimize for debugging

Creating a bootable ARM root filesystem

Configuration options: 2/3

▶ Toolchain

- ▶ C library = glibc
- ▶ ☒ Enable C++ support
- ▶ ☒ Build cross gdb for the host
- ▶ ☒ TUI support
- ▶ Python support = Python3
- ▶ GDB debugger Version = gdb 9.2.x

▶ System configuration

- ▶ System hostname = BASC2020
- ▶ System banner = Welcome to BASC2020 Buildroot
- ▶ Root password = BASC2020
- ▶ Network interface to configure through DHCP = eth0
- ▶ Custom scripts to run before creating filesystem images =
./enable-ssh-root-login.sh ./gef-python.sh

Creating a bootable ARM root filesystem

Configuration options: 3/3

- ▶ Target packages
 - ▶ Debugging, profiling and benchmark
 - ▶ ☒ gdb
 - ▶ ☒ full debugger
 - ▶ ☒ gdbserver
 - ▶ ☒ TUI support
 - ▶ ☒ ltrace
 - ▶ ☒ strace
 - ▶ ☒ valgrind
 - ▶ Networking applications
 - ▶ ☒ openssh
 - ▶ ☐ client
 - ▶ ☒ key utilities
- ▶ Filesystem images
 - ▶ ☒ ext2/3/4 root filesystem
 - ▶ exact size = 512M
 - ▶ ☐ tar the root filesystem
- ▶ Host utilities
 - ▶ host python3
 - ▶ ssl

Creating an ARM root filesystem

Performing the build

1. Save the configuration to the default `.config` path
2. Download sources with `make source`
3. Start the build with `make`

Customizing our images

Build time overlay

- ▶ Create a directory
- ▶ Add `BR2_ROOTFS_OVERLAY=my-overlay` to `.config`
- ▶ Rebuild using `make`
- ▶ The structure of `my-overlay` will be copied to the rootfs

How to specify multiple overlays

Multiple overlays can be specified by separating them with spaces in the `BR2_ROOTFS_OVERLAY` directive

Customizing our images

Build time script

Add `BR2_ROOTFS_POST_BUILD_SCRIPT=my-script.sh` to `.config`

Available environment variables inside:

<code>BR2_CONFIG</code>	path of <code>.config</code>
<code>HOST_DIR</code>	path of output/host
<code>STAGING_DIR</code>	path of output/staging
<code>TARGET_DIR</code>	path of output/target
<code>BUILD_DIR</code>	path of output/build
<code>BINARIES_DIR</code>	path of output/images
<code>BASE_DIR</code>	path of output

How to specify multiple scripts

Multiple scripts can be specified by separating them with spaces in the `BR2_ROOTFS_POST_BUILD_SCRIPT` directive

Customizing our images

Editing the target directory

1. Add your files to the output/target directory
2. Rebuild using `make`

Warning

Your files might be rewritten / deleted by buildroot

Customizing our images

D.I.Y. approach

1. Unpack your rootfs (with `tar -xzf` for instance)
2. Perform your modifications
3. Repack your rootfs (with `tar -cf` for instance)

Using the cross-compiler

1. Extract the cross-compiler
2. Run `relocate-sdk.sh`
3. Edit your `$PATH` variable: `export PATH="$PATH:$PWD/bin"`
4. You can invoke your cross compiler with commands like `arm-buildroot-linux-gnueabi-hf-<COMMAND NAME>`

Notable entries

- ▶ `arm-buildroot-linux-gnueabi-hf-gcc`
- ▶ `arm-buildroot-linux-gnueabi-hf-gdb`
- ▶ `arm-buildroot-linux-gnueabi-hf-nm`

Improving gdb with library symbols

See the section [▶ Using gdb](#)

Running dynamic executables in Docker

```
sudo docker import rootfs.tar basc-buildroot
sudo docker run --rm -it \
  --volume "$(which qemu-arm-static):/bin/qemu-arm-static" \
  --volume "${PWD}:/:/host" \
  --entrypoint /bin/qemu-arm-static \
  --workdir "/host" \
  basc-buildroot \
  /bin/sh
```

Running dynamic executables with systemd-nspawn

```
mkdir -p basc-rootfs
tar -xf rootfs.tar -C basc-rootfs
cp -f "$(which qemu-arm-static)" \
    basc-rootfs/bin/qemu-arm-static
sudo systemd-nspawn \
    --register=no \
    -D basc-rootfs \
    /bin/qemu-arm-static /bin/sh
```

Package needed

You might need to install the package `systemd-container`

Booting the rootfs

```
#!/bin/sh
#
# Boots the built rootfs
#
exec qemu-system-arm \
    -machine virt \
    -cpu cortex-a7 \
    -smp 2 -m 2000 \
    -kernel bootable-rootfs/zImage \
    -device virtio-blk-device,drive=rootfs \
    -drive file=bootable-rootfs/rootfs.ext2,if=none,format=raw,id=rootfs \
    -append "console=ttyAMA0,115200 rootwait root=/dev/vda" \
    -netdev user,id=user0,hostfwd=tcp::2222-:22,hostfwd=tcp::1234-:1234 \
    -device virtio-net-device,netdev=user0 \
    -serial stdio \
    -display none
```

Tips and tricks

Using SSH

Opening an SSH session

```
ssh \  
-o UserKnownHostsFile=/dev/null \  
-o StrictHostKeyChecking=no \  
-p 2222 root@localhost
```

Sharing a folder

```
mkdir -p guest-os-ssh  
sshfs root@localhost:/ ./guest-os-ssh \  
-f \  
-o port=2222 \  
-o reconnect \  
-o UserKnownHostsFile=/dev/null \  
-o StrictHostKeyChecking=no
```

Using {l,s,uf}trace

What did you expect?

- ▶ {l,s,uf}trace do work as expected
- ▶ Can only be performed on QEMU **system** emulation

Note that *ltrace* has a bug with unwinding of DWARF tables on ARM and will show limited information.

Using gdb

On the guest

`gdbserver :1234 command to debug`

On the host (*From the cross-compiler extracted folder*)

```
bin/arm-buildroot-linux-gnueabi-hf-gdb \  
-X arm-buildroot-linux-gnueabi-hf/sysroot/usr/share/buildroot/gdbinit \  
executable name
```

On the host gdb shell attach with target `remote localhost:1234`