Buildroot

BASC2020 seminar

Giacomo Longo

University of Genoa

11 December 2020

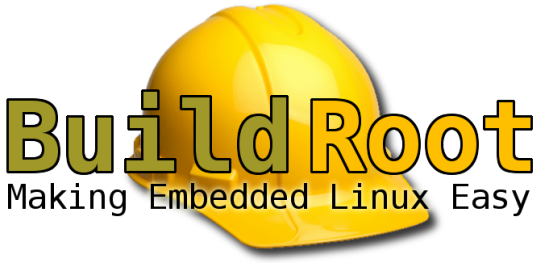
Table of contents

BuildRoot

BuildRoot

What's BuildRoot

BuildRoot



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
- High end Hi-Fi amplifiers

BuildRoot
Why BuildRoot

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-rc3
- 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
- RISCV
- 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

Creating some BuildRoots

Creating some BuildRoots

Prerequisites

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-rc3 but also applies two required patches that we need for today's seminar

Creating some BuildRoots

Creating an ARM cross compiler

1. `cd buildroot-2020.11-rc3`
2. `cp ../scripts/gef-python.sh ./gef-python.sh`
3. `chmod +x *.sh`
4. `make distclean`
5. `make defconfig`

Steps explaination

1. Entering the BuildRoot directory
2. We will need some Python packages in order to utilize gdb's GEF plugin.
   Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.
3. The script will not be invoked by a shell, but as an executable, so we should ensure correct permissions
4. This will cleanup leftover files (if it fails it means there's nothing to remove yet)
5. This will reset the configuration options to BuildRoot defaults

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

Options explaination

Target Options define what target architecture and variant we are targeting.

Our target architecture is ARM little-endian, a commonplace 32bit architecture used in mobile phones.

The variant defines the model of CPU we are targeting, different CPUs might have additional instructions, register layouts and cache hierarchies. Including an architecture allows the compiler to optimize register allocation and to use custom instructions.

Here our target variant is the Cortex A7: based on the ARM reference v7 architecture this model is most known for being the heart of the Raspberry Pi 2.

Not all ARM CPUs include hardware support for operation on floating point numbers, here we decide to emulate a processor that supports an optional set of instructions called VFPv4, with support for 16 double precision registers (D16)

Build Options define what flags BuildRoot will use to build our executables and libraries.

The compiler cache replaces all compiler invocations with calls to a

wrapper compiler called `ccache` that will avoid compiling the same source twice, don't worry, it's still going to take a lot of time to build everything.
We will be building the entirety of the OS with debugging symbols so gdb will be able to instrument even the system libraries.
The packages are going to be build the maximum debug optimization enabled, so they will be easier to inspect for our debugging tools.
Binary stripping is disabled so all information is preserved.

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

Options explaination
Toolchain options define what environment our packages will be linked against.
BuildRoot's choice of default libc is uClibc-ng, a minimal libc built for the most resource-constrained usages.
Given our desktop systems and that the majority of binaries in the wild are linked against the GNU libc, let's use it also inside our BuildRoot.
We need C++ support in order to build some packages without failures.
By requesting a cross gdb we instruct BuildRoot to generate a gdb target at our host architecture that can debug programs written for other architectures.
TUI and Python support are required in order to use GEF.
We will need some Python packages in order to utilize gdb's GEF plugin. Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.
We do not need a root filesystem to build a crosscompiler.

We need Python for GEF and OpenSSL in order to install GEF's dependencies using PIP.

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-rc3`
2. `cp ../scripts/gef-python.sh ./gef-python.sh`
3. `chmod +x *.sh`
4. `make distclean`
5. `make defconfig`

Steps explaination

1. Entering the BuildRoot directory
2. We will need some Python packages in order to utilize gdb's GEF plugin.
   Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.
3. The script will not be invoked by a shell, but as an executable, so we should ensure correct permissions
4. This will cleanup leftover files (if it fails it means there's nothing to remove yet)
5. This will reset the configuration options to BuildRoot defaults

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

Options explaination

Target Options define what target architecture and variant we are targeting.

Our target architecture is ARM little-endian, a commonplace 32bit architecture used in mobile phones.

The variant defines the model of CPU we are targeting, different CPUs might have additional instructions, register layouts and cache hierarchies. Including an architecture allows the compiler to optimize register allocation and to use custom instructions.

Here our target variant is the Cortex A7: based on the ARM reference v7 architecture this model is most known for being the heart of the Raspberry Pi 2.

Not all ARM CPUs include hardware support for operation on floating point numbers, here we decide to emulate a processor that supports an optional set of instructions called VFPv4, with support for 16 double precision registers (D16)

Build Options define what flags BuildRoot will use to build our executables and libraries.

The compiler cache replaces all compiler invocations with calls to a

wrapper compiler called `ccache` that will avoid compiling the same source twice, don't worry, it's still going to take a lot of time to build everything.
We will be building the entirety of the OS with debugging symbols so gdb will be able to instrument even the system libraries.
The packages are going to be build the maximum debug optimization enabled, so they will be easier to inspect for our debugging tools.
Binary stripping is disabled so all information is preserved.

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

Options explaination

Toolchain options define what environment our packages will be linked against.

BuildRoot's choice of default libc is uClibc-ng, a minimal libc built for the most resource-constrained usages.

Given our desktop systems and that the majority of binaries in the wild are linked against the GNU libc, let's use it also inside our BuildRoot.

We need C++ support in order to build some packages without failures.

By requesting a cross gdb we instruct BuildRoot to generate a gdb target at our host architecture that can debug programs written for other architectures.

TUI and Python support are required in order to use GEF.

We will need some Python packages in order to utilize gdb's GEF plugin. Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.

Our root filesystem will also contain gdb and its server.

Creating an ARM root filesystem

Configuration options: 3/3

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

We will package our root filesystem as a tar file.
We need Python for GEF and OpenSSL in order to install GEF's dependencies using PIP.

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`

Creating some BuildRoots

Creating a bootable ARM root filesystem

Creating a bootable ARM root filesystem

Initial setup

1. `cd buildroot-2020.11-rc3`
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`

Steps explaination

1. Entering the BuildRoot directory
2. In order to use our built rootfs inside QEMU we need to add some Kernel modules, this file contains the kernel configuration options that should be set to support smooth virtualization under QEMU
3. We will need some Python packages in order to utilize gdb's GEF plugin.
   Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.
4. By default OpenSSH server does not allow access to the `root` user as a security feature.
   In order to be able to simply login to our BuildRoot we need to disable this option.
5. The script will not be invoked by a shell, but as an executable, so we should ensure correct permissions

6. This will cleanup leftover files (if it fails it means there's nothing to remove yet)

7. This will reset the configuration options to BuildRoot defaults

Creating a bootable ARM root filesystem

Configuration options: 1/4

`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

Options explaination

Target Options define what target architecture and variant we are targeting.

Our target architecture is ARM little-endian, a commonplace 32bit architecture used in mobile phones.

The variant defines the model of CPU we are targeting, different CPUs might have additional instructions, register layouts and cache hierarchies. Including an architecture allows the compiler to optimize register allocation and to use custom instructions.

Here our target variant is the Cortex A7: based on the ARM reference v7 architecture this model is most known for being the heart of the Raspberry Pi 2.

Not all ARM CPUs include hardware support for operation on floating point numbers, here we decide to emulate a processor that supports an optional set of instructions called VFPv4, with support for 16 double precision registers (D16)

Build Options define what flags BuildRoot will use to build our executables and libraries.

The compiler cache replaces all compiler invocations with calls to a

wrapper compiler called `ccache` that will avoid compiling the same source twice, don't worry, it's still going to take a lot of time to build everything.
We will be building the entirety of the OS with debugging symbols so gdb will be able to instrument even the system libraries.
The packages are going to be build the maximum debug optimization enabled, so they will be easier to inspect for our debugging tools.
Binary stripping is disabled so all information is preserved.

Creating a bootable ARM root filesystem

Configuration options: 2/4

- 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

Options explaination
Toolchain options define what environment our packages will be linked against.
BuildRoot's choice of default libc is uClibc-ng, a minimal libc built for the most resource-constrained usages.
Given our desktop systems and that the majority of binaries in the wild are linked against the GNU libc, let's use it also inside our BuildRoot.
We need C++ support in order to build some packages without failures.
By requesting a cross gdb we instruct BuildRoot to generate a gdb target at our host architecture that can debug programs written for other architectures.
TUI and Python support are required in order to use GEF.
The system customization of hostname / banner is simply aestetic.
The root password has to be set in order to be able to login.
We choose to automatically configure the first ethernet network interface in order to avoid bothering with networking settings after the rootfs will be booted in QEMU.
By default OpenSSH server does not allow access to the `root` user as a security feature.

In order to be able to simply login to our BuildRoot we need to disable this option.
We will need some Python packages in order to utilize gdb's GEF plugin.
Given that the cross compiler we build is completely indipendent from our system with its own Python installation, we need a way to perform this customization at image build time, this premade script (`scripts/gef-python.sh`) will install PIP and the required packages inside the cross compiler environment so we will be able to utilize GEF.

Creating a bootable ARM root filesystem

Configuration options: 3/4

- Target packages
  - Debugging, profiling and benchmark
    - ☒ gdb
    - ☒ full debugger
    - ☒ gdbserver
    - ☒ TUI support
    - ☒ ltrace
    - ☒ strace
    - ☒ uftrace [1]
    - ☒ valgrind
  - Networking applications
    - ☒ openssh
    - ☐ client
    - ☒ key utilities

[1] Available only if you used `get-buildroot.sh`

Our root filesystem will also contain gdb and its server.
Given that we will be running an actual OS, ltrace, strace and valgrind will be supported and can be used.
An OpenSSH server will allow us to connect and sync files from/to the rootfs.

Creating a bootable ARM root filesystem

Configuration options: 4/4

- Filesystem images
  - ☒ ext2/3/4 root filesystem
    - exact size = 512M
  - ☐ tar the root filesystem
- Host utilities
  - host python3
  - ssl

We will package our root filesystem as an ext image.
We need Python for GEF and OpenSSL in order to install GEF's dependencies using PIP.

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

Using our BuildRoot

Using our BuildRoot
Producing binaries for the target

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-gnueabihf-<COMMAND NAME>`
   Notable entries
   - `arm-buildroot-linux-gnueabihf-gcc`
   - `arm-buildroot-linux-gnueabihf-gdb`
   - `arm-buildroot-linux-gnueabihf-nm`

Improving gdb with library simbols

See the section ▸ Using gdb

1.

2. this is needed because BuildRoot gdbinit file references an absolute path (the one in which this buildroot was assembled) and you probably aren't running it with the same path.

3. this is simply a convenience, you can perform the same actions by specifying the complete path manually.

4. the extremely long name is actually called an **host triplet**

   - The first component is the architecture (arm)
   - The following string is the optional vendor string, it is used in order to distinguish between compilers targeting the same triplet (buildroot)
   - The second component is the target operating system (linux)
   - The third component is the ABI specification (gnueabi)
     GNU indicates that we are using a GNU toolchain built system (gcc, GNU ld, ...)
     it is relevant because on ARM we do have the concept of EABIs: Embedded Application Binary Interfaces abstract less details of the underlying architecture and allow to minimize footprint, but compatibility between different EABIs is usually not possible.

Using our BuildRoot
Running dynamic executables

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
```

Docker supports importing .tar files as images for containers.
That docker import command is equivalent to pulling an image consisting of a single layer comprised of our rootfs.
We will run the executables in QEMU by leveraging the static binary provided by the host, in this configuration most dynamic executables will work but some calls to fork/exec syscalls and any tracing request will fail.

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`

Systemd-nspawn is like chroot, with some nifty additions (for istance it automatically mounts the pseudo filesystems for us), the --register=no is due to the fact that we want an ephemeral machine and not a system service.
We will run the executables in QEMU by leveraging the static binary provided by the host, in this configuration most dynamic executables will work but some calls to fork/exec syscalls and any tracing request will fail.

Using our BuildRoot

Performing dynamic analysis

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
```

QEMU system emulation requires a lot of arguments:

machine QEMU can emulate a specific machine or a more "generic" one, virt is the recommended hardware configuration for general purpose ARM emulation.
The usage of a preexisting machine can also limit the availability of hardware, lock the number of cpus or cap the maximum memory available.
ARM machines do not have the equivalent of BIOS / EFI standards and so there is no hardware automatic discovery, hardware support has to be precompiled into the kernel with device tree files.
In order to avoid limitations we have explicitly compiled in VIRTIO paravirtualization drivers to make the system aware of it being emulated.

cpu This sets which instructions are available to the guest, unincluded instructions will trap the hypervisor giving an effect similar to issuing an illegal instruction on a real cpu

smp Sets the number of processors

m Sets the amount of RAM in megabytes

kernel The image has no bootloader in it so we need QEMU to load the kernel for us

device virtio-blk-device This creates a virtual hard disk, supported by the virtio drivers we included in the kernel

drive This binds the disk drive to the rootfs we specified

append This will be the kernel command line, specifying a serial console and the root device location

netdev This sets up the virtual network card and forward host port 2222 to port 22 of guest and port 1234 of host to port 1234 of guest

device virtio-net-device This will create a virtual network card, supported by the virtio drivers we included in the kernel

serial stdio This will allow us to interact with the console on the calling terminal

display none We do not have a display nor the rootfs has support for one

Opening an SSH session

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

Sharing a folder

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

StrictHostKeyChecking=no in order to protect us from MITM attacks, our SSH client does store the public key fingerprint of every host we connect to.
If for some reason a host we contacted before changes its public key, we will not be able to connect (unless we reset the stored data) because it has changed its identity.
Since we are working on localhost with a machine we just built there's no risk disabling this setting.

UserKnownHostsFile=/dev/null the command given above will disable checking but our host file might still be modified and accessed, by using /dev/null we preserve our SSH client's state as-is.

sshfs works thanks to FUSE virtual filesystem in userspace and allows accessing the remote system files as if they were attached locally.

Using {l,s,uf}trace

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

ltrace is buggy on ARM

ltrace has a bug with unwinding 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*)[2]

```
bin/arm-buildroot-linux-gnueabihf-gdb \
 -x arm-buildroot-linux-gnueabihf/sysroot/usr/share/buildroot/gdbinit \
 executable name
```

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

[2] or use `run-cross-gdb.sh` from my release package

The `-x` argument points gdb to an additional `gdbinit`.
The only command inside is `set sysroot` that will allow gdb to access libraries pertaining to the guest.
This gdb can only be used with its own gdbserver: the gdb protocol depends on the architecture.

Customizing our images

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:

`BR2_CONFIG`	path of `.config`
`HOST_DIR`	path of `output/host`
`STAGING_DIR`	path of `output/staging`
`TARGET_DIR`	path of `output/target`
`BUILD_DIR`	path of `output/build`
`BINARIES_DIR`	path of `output/images`
`BASE_DIR`	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)