

# Compile Linux on RISC-V processor

Iman Rasouli Parto

University of Tehran

*Abstract— This article provides instructions for compiling 32-bit Linux on RISC-V processor and test the program.*

*Keywords—config, make, Makefile, RISC-V toolchain, GCC*

**Introduction:** In embedded systems, the operating system (OS) communicates with the processor, manages resources, executes programs and performs other tasks. Therefore, understanding how an operating system compiles on a processor is essential. To test this process, a RISC-V processor is a suitable choice as it provides the appropriate tools for the task. In this article we are going to compile Linux on Spike simulator for RISC-V processor.

---

At the beginning, we need to update the system using the command below:

```
$ sudo apt update && sudo apt upgrade
```

Before compiling the Linux, we will need some required build tools. The following command will install them:

```
$ sudo apt-get install device-tree-compiler autoconf automake autotools-dev curl libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev \
libexpat-dev python2-dev python3-dev unzip libglib2.0-dev libpixman-1-dev git rsync wget cpio \
libncurses-dev
```

## Step 1: Get the simple repository from git

```
$ git clone https://github.com/riscv-zju/riscv-rss-sdk.git
```

Then, go to the directory and run quickstart:

```
$ cd riscv-rss-sdk/
```

```
$ sh quickstart.sh
```

After that, fetch all the system dependencies for the code:

```
$ git submodule update --init --recursive --progress
```

## Step 2: Set the RISC-V director as a variable and add it to PATH

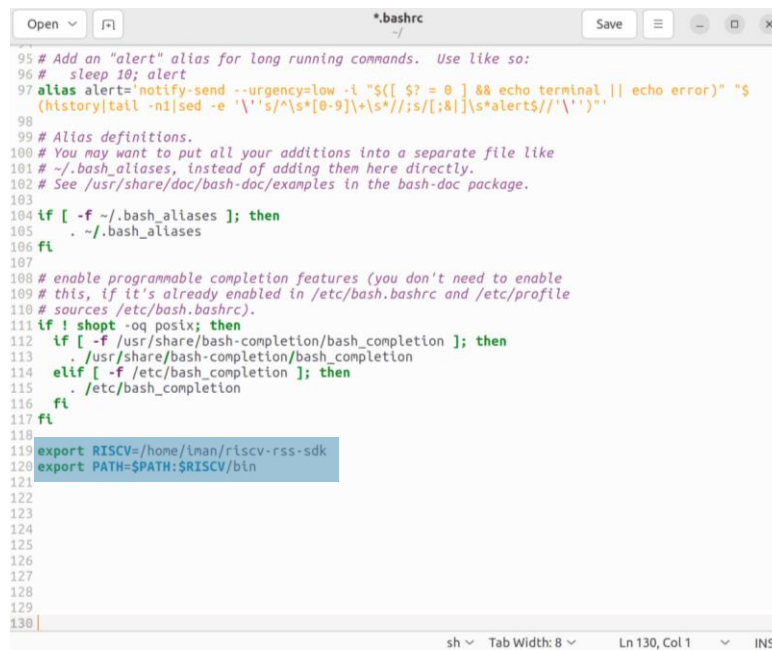
```
$ export RISCV=/home/(your username)/riscv-rss-sdk
```

```
$ export PATH=$PATH:$RISCV/bin
```

Note: Replace “your\_username” with your actual username. Be sure to use your own username in place of “your\_username” to avoid any confusion.

Variables will be lost after a reboot. To prevent them from being lost after a reboot, open the “bashrc” file using a text editor and add these two commands to it. Make sure you are in your home directory.

`$ gedit .bashrc`



```
95 # Add an "alert" alias for long running commands. Use like so:
96 # sleep 10; alert
97 alias alert='notify-send --urgency=low -i "${[ $? = 0 ]} && echo terminal || echo error" "$
(history|tail -n|sed -e '\''s/^s*[0-9]\+\s*//;s/[:&]]\s*alert$//'\''"'
98
99 # Alias definitions.
100 # You may want to put all your additions into a separate file like
101 # ~/.bash_aliases, instead of adding them here directly.
102 # See /usr/share/doc/bash-doc/examples in the bash-doc package.
103
104 if [ -f ~/.bash_aliases ]; then
105     . ~/.bash_aliases
106 fi
107
108 # enable programmable completion features (you don't need to enable
109 # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
110 # sources /etc/bash.bashrc).
111 if ! shopt -oq posix; then
112     if [ -f /usr/share/bash-completion/bash_completion ]; then
113         . /usr/share/bash-completion/bash_completion
114     elif [ -f /etc/bash_completion ]; then
115         . /etc/bash_completion
116     fi
117 fi
118
119 export RISCv=/home/iman/riscv-rss-sdk
120 export PATH=$PATH:$RISCv/bin
121
122
123
124
125
126
127
128
129
130
```

Figure 1. Add variables in bashrc file

Finally, save the file and exit.

### Step 3: Cross compile toolchain

`$ cd $RISCv/repo/riscv-gnu-toolchain/`

`$ mkdir build && cd build`

`$ ../configure --prefix=$RISCv --with-arch=rv32imafdc --with-abi=ilp32d`

`$ make`

The process will take some time.

Note: **Do not** execute any command with root access (don't use "sudo"). It could fail the process.

Once the compilation is complete, the GCC compiler will be installed. You can find GCC compiler and its libraries in the "RISCv" and "RISCv/bin" directories. Commands that start with "riscv32-unknown-elf-" are used for cross-compile for the RISC-V processor. In addition, we need to cross compile GCC for Linux kernel with the command below:

`$ make linux`

After compilation, the compiler for Linux will be appear in "RISCv/bin" directory with start in name "riscv32-unknown-linux-gnu-". Now you have all tools for compile all we need for simulations.

#### Step 4: Build the Proxy kernel

Execute following instructions to build the Proxy kernel:

```
$ cd $RISCV/repo/riscv-pk
$ mkdir build && cd build
$ ../configure --prefix=$RISCV --host=riscv32-unknown-elf
$ make
$ make install
```

#### Step 5: Build Spike simulator

Execute following commands to compile Spike:

```
$ cd $RISCV/repo/riscv-pk
$ mkdir build && cd build
$ ../configure --prefix=$RISCV --enable-histogram
$ make
$ make install
```

#### Step 6: Verify the simulator

To verify the compilation, run this simple code with the commands:



```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello World!\n");
6  }
7
```

```
$ riscv32-unknown-elf-gcc test.c
$ $RISCV/repo/riscv-isa-sim/build/spike --isa=RV32I $RISCV/repo/riscv-pk/build/pk a.out
```

If “Hello world!” appeared in the terminal it’s mean you have everything for compilation and simulations for linux kernel.

## Step 7: Configure the Linux to compile

We should configure some features in config menu to compile Linux correctly. With command below open the config menu:

```
$ cd $RISCV
```

```
$ make buildroot_initramfs-menuconfig
```

A menu will appear in terminal that needs to change a little bit.

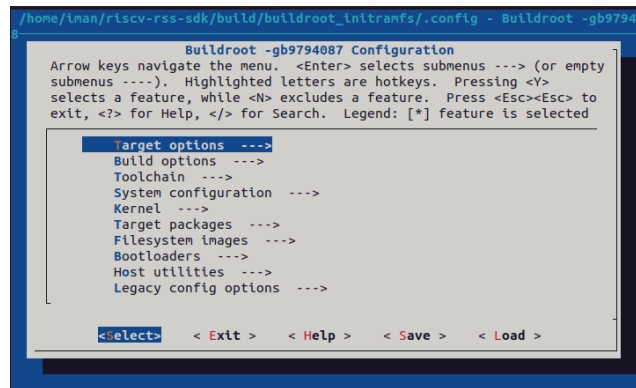


Figure 2. Config menu

In “Target options” section, enter “Target ABI” and select the “lp32” version. In “Toolchain” section, enter “Toolchain type” and select “Buildroot toolchain”. Next, enter “C library” and choose “glibc” (stay in “Toolchain” tab). Again in “Toolchain” section, enter “Kernel Headers” and choose “5.15.x” version. Then go to “System configuration” section, enable “Enable root login with password” and set a password in “Root password”. Exit the menu and save the changes, finally compile the Linux on simulator with the following command:

```
$ make
```

The compilation takes a huge time, after completed, you’ll see a Linux terminal (It’s not your system terminal). Login with root and the password you set in configuration.

## Step 8: Test the compiled Linux

To test the system, create a directory and use the “ls” command to view the directory. If the directory is created successfully, it indicates that the system is functioning properly.

```
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: OK

Welcome to Buildroot IMAN
buildroot login: root
root
Password: 123456

# mkdir test
mkdir test
# ls
ls
test
# mkdir test2
mkdir test2
# ls
ls
test test2
```

Figure 3. Test compiled Linux on simulator

## Appendix

- ✓ **Buildroot:** Buildroot is a set of Makefiles and patches that simplifies and automates the process of building a complete and bootable Linux environment for an embedded system, while using cross-compilation to allow building for multiple target platforms on a single Linux-based development system. Buildroot can automatically build the required cross-compilation toolchain, create a root file system, compile a Linux kernel image, and generate a boot loader for the targeted embedded system, or it can perform any independent combination of these steps. For example, an already installed cross-compilation toolchain can be used independently, while Buildroot only creates the root file system. Buildroot is primarily intended to be used with small or embedded systems based on various computer architectures and instruction set architectures (ISAs), including x86, ARM, MIPS and PowerPC. Multiple C standard libraries are supported as part of the toolchain, including the GNU C Library, uClibc and musl, as well as the C standard libraries that belong to various preconfigured development environments, such as those provided by Linaro. Buildroot's build configuration system internally uses Kconfig, which provides features such as a menu-driven interface, handling of dependencies, and contextual help; Kconfig is also used by the Linux kernel for its source-level configuration.
- ✓ **Busybox:** BusyBox is a software suite that provides several Unix utilities in a single executable file. It runs in a variety of POSIX environments such as Linux, Android, and FreeBSD, although many of the tools it provides are designed to work with interfaces provided by the Linux kernel. It was specifically created for embedded operating systems with very limited resources.

## References

- [1] <https://en.wikipedia.org/wiki/Buildroot>
- [2] <https://en.wikipedia.org/wiki/BusyBox>