Linux Kernel Build Guideline

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Change by** | **Change Description** |
| 2023-11-02 | 1.0 | Trinh Tran Trung – BV Corp.  Nguyen Duc Nghia – BV Corp.  Le Thanh Phuc – BV Corp. | Initial. |

# ToolChain

## About toolchain

* Toolchain is a collection of tools used to compile source code into executable files that can run on a target device (real hardware).
* The toolchain includes a compiler, linker, runtime libraries, and some miscellaneous tools. Typically, a Linux toolchain is based on components from the GNU Project. A basic toolchain will always have at least the following three components:
  + Binutils: a collection of binary tools such as the assembler (as), linker (ld), archiver (ar), object copy tool (objcopy), etc.
  + Compiler: both C compiler and C++ compiler.
  + C library: a set of standard APIs based on POSIX. Applications use the C library to communicate with the kernel.
* Building your own toolchain for your specific platform from source code using build systems like Buildroot or Yocto Project is a common practice. Alternatively, you can obtain pre-built binary toolchains from a vendor, which may be either paid or free. Some hardware vendors also provide toolchains as part of the hardware purchase or offer them as complimentary tools. The choice depends on your specific needs, preferences, and the level of customization you require for your development environment.
* Toolchains can be categorized into two types based on their purpose: Native toolchain and Cross toolchain:
  + Native toolchain: This type of toolchain runs on a system similar to the target device (the device used to run programs built by the toolchain). It's typically used for building programs that run directly on the same type of system, such as desktop applications.
  + Cross toolchain: This toolchain runs on a system different from the target device. For example, it can be used on a desktop to build programs that will run on a different embedded device. This type of toolchain speeds up the software development process since you can develop software on a desktop PC and then load and run it on the target device.
* Most software developed for embedded Linux systems is done on desktop PCs using a cross toolchain. This is because embedded devices often have significantly lower specifications in terms of CPU, memory, and storage, making them unsuitable for direct development work.

## Toolchain Components

### Binutils

* The GNU Binutils is the first component of a toolchain. The GNU Binutils contains two very important tools:
  + *as*, the assembler, that turns assembly code (generated by GCC) to binary.
  + *ld*, the linker, that links several object code into a library, or an executable.
* Binutils also contains a couple of other binary file manipulation or analysis tools, such as objcopy, objdump, nm, readelf, strip, and so on. The Binutils website has some documentation on all these tools.

### C, C++, Java, Ada, Fortran, Objective-C compiler

* The second major component of a toolchain is the compiler. In the embedded Linux, the only realistic solution today is GCC, the GNU Compiler Collection. Nowadays, as input, it not only supports C, but also C++, Java, Fortran, Objective-C and Ada. As output, it supports a very wide range of architectures.

### C library

* The C library implements the traditional POSIX API that can be used to develop userspace applications. It interfaces with the kernel through system calls and provides higher-level services.
* Realistically, there are nowadays two options for the C Library:
  + glibc is the C library from the GNU project. It's the C library used by virtually all desktop and server GNU/Linux systems. It's feature-full, portable, complies to standards, but a bit bloated.
  + Embedded GLIBC (EGLIBC) is a variant of the GNU C Library (GLIBC) optimized for embedded systems. Its goals include reduced footprint, support for cross-compiling and cross-testing, while maintaining source and binary compatibility with GLIBC. The project is discontinued.
  + uClibc is an alternate C library, which features a much smaller footprint. This library can be an interesting alternative if flash space and/or memory footprint is an issue. However, the space advantages gained using uClibc are becoming less important as the price of memory and flash continues to drop. It is still useful C library for embedded systems without an MMU.
  + uClibc-ng is a spin-off of uClibc C library. The main goal of the spin-off is to do regular releases and do a lot of automatic runtime testing.
  + musl New standard C library. musl is lightweight, fast, simple, free, and strives to be correct in the sense of standards-conformance and safety.
* The C library has a special relation with the C compiler, so the choice of the C library *must* be done when the toolchain is generated. Once the toolchain has been built, it is no longer possible to switch to another library.

### Debugger

* The debugger is also usually part of the toolchain, as a cross-debugger is needed to debug applications running on your target machine. In the embedded Linux world, the typical debugger is GDB.

### Lazarus and Free Pascal

* Free Pascal is a professional but free 32 bit / 64 bit compiler for Pascal and Object Pascal. It supports a wide variety of processors and Linux distributions including the Raspberry Pi.
* The Free Pascal toolchain is widely independent from GCC and other external tools. Major components are the Free Pascal compiler (FPC), a command-line tool, a text-mode IDE and, as an optional component, Lazarus, a full-featured GUI-based IDE. FPCUnit is a framework allowing for unit-testing.
* On most platforms Free Pascal makes use of the GDB debugger.

## Set up toolchain on Linux

* Download toolchain from [Toolchain](https://renesasgroup.sharepoint.com/:f:/r/sites/team_RCL-EX/Shared%20Documents/Gen3%20SW%20Maintenance/Toolchain?csf=1&web=1&e=v5ixnz)
* Unzip this folder

**$ unzip Toolchain.zip**

**$ cd Toolchain**

* Merge all splited files, then unzip the merged file

**$ cat poky-glibc-x86\_64-core-image-weston-sdk-aarch64-salvator-x-toolchain-3.1.11.7z.00\* > file.7z**

**$ 7za x file.7z**

* Execute the script file

**$ chmod 777 poky-glibc-x86\_64-core-image-weston-sdk-aarch64-salvator-x-toolchain-3.1.11.sh**

**$ ./poky-glibc-x86\_64-core-image-weston-sdk-aarch64-salvator-x-toolchain-3.1.11.sh**

* The excution of script file will install the toolchain to path

For more information about getting toolchain, visit: [Yocto Project Application Development and the Extensible Software Development Kit (eSDK) — The Yocto Project ® 3.1.11 documentation](https://docs.yoctoproject.org/3.1.11/sdk-manual/sdk-manual.html)

# How to build Kernel Core version 5.10.194

## Steps to build

* Clone source code of Renesas Linux BSP using the following command:

$ wget <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/snapshot/linux-5.10.194.tar.gz>

* Decompress the downloaded file:

$ tar xvzf linux-5.10.194.tar.gz

* Go to the decompressed folder:

$ cd linux-5.10.194

* Continuing with the building steps, Source Toolchain

$ source /prj/soc-sw/local/common/opt/poky/3.1.11/environment-setup-aarch64-poky-linux

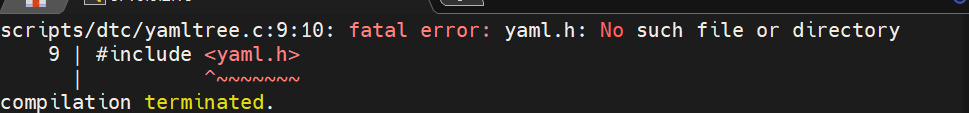
* Check toolchain cross compile:

$ echo $CROSS\_COMPILE

* Set up ARCH:

**$ export ARCH=arm64**

* Unset flags: Prevent unwanted errors when building kernel. For example, when build kernel without unsetting these flags, the following error has been raised:



**$ unset PKG\_CONFIG\_PATH**

**$ unset LD\_LIBRARY\_PATH**

* Clean before starting build:

**$ make clean**

**$ make distclean**

* Create version file

**$ touch .scmversion**

* Create .config file

**$ make defconfig**

* Start to build:

$ make

* Get image after build process at linux-bsp/arch/arm64/boot

## Build log

* Check reference file: [BuildLog\_Release5.10.194.txt](https://renesasgroup.sharepoint.com/:t:/r/sites/team_RCL-EX/Shared%20Documents/Gen3%20SW%20Maintenance/Training_Output/BV/Nov%201st%20-%20Build%20Linux%20Kernel/Task%201%20-%20Build%20Linux%20Kernel%20Core/BuildLog_Release5.10.194.txt?csf=1&web=1&e=xHzY41)

# How to build Renesas Linux BSP

## Steps to build

* Clone source code of Renesas Linux BSP using the following command:

$ git clone <https://github.com/renesas-rcar/linux-bsp.git>

* Checkout to target commit. For instance, to have Renesas Linux BSP version 5.1.4.2 with commit 452163c75612e4161099c0bf5178fd0cf60e2cad, execute command below:

$ git checkout 452163c75612e4161099c0bf5178fd0cf60e2cad

* Continuing with the building steps, Source Toolchain

$ source /prj/soc-sw/local/common/opt/poky/3.1.11/environment-setup-aarch64-poky-linux

* Check toolchain cross compile

**$ echo $CROSS\_COMPILE**

* Set up ARCH:

**$ export ARCH=arm64**

* Unset flags

**$unset PKG\_CONFIG\_PATH**

**$unset LD\_LIBRARY\_PATH**

* Clean before starting build

**$ make clean**

**$ make distclean**

* Create version file

**$ touch .scmversion**

* Create .config file

**$ make defconfig**

* Start to build

$ make

* Get image after build process at linux-bsp/arch/arm64/boot

## Build log

* Check reference file: [BuildLog\_RenesasLinuxBSPv5.1.4.2.txt](https://renesasgroup.sharepoint.com/:t:/r/sites/team_RCL-EX/Shared%20Documents/Gen3%20SW%20Maintenance/Training_Output/BV/Nov%201st%20-%20Build%20Linux%20Kernel/Task%202%20-%20Build%20Renesas%20Linux%20BSP/BuildLog_RenesasLinuxBSPv5.1.4.2.txt?csf=1&web=1&e=S40ejk)