Building a custom Slackware AArch64 Linux Kernel package from source

Document	Describe how to customise the Slackware AArch64 Kernel packages
Version	1.02, June 2023
Author	Stuart Winter <mozes@slackware>
References	Slackware ARM source tree README
See also	Raspberry Pi Kernel fork self build

Caveats / Warnings

General Guidance

- There are no guarantees that your customisations will provide a working Kernel, even if it built successfully. You may need to rescue the system, which is beyond the scope of this document.
- When customising the Kernel, it's best to remain at the same major version and patch level. For example, if the Slackware AArch64 Kernel is Linux 5.18.3, it's safer to remain either at this exact version or to use another Kernel within the 5.18 series rather than moving to 5.19 (or even backwards to 5.17).

If you want to add support for a Hardware Model that isn't yet supported, building a custom Kernel is just one part of the integration. Please read the Slackware ARM Direct Integration Guide.

Raspberry Pi Kernel fork

This document primarily describes how to customise the standard/mainline Kernel source tree that Slackware uses. Using the Raspberry Pi Kernel fork is also described to build a Kernel for the Raspberry Pi 4, but this isn't recommended because the Kernel configuration is out of line with the standard. In particular this:

- prevents use of the serial port and
- the Kernel module for the Real Time Clock that's recommended within the Raspberry Pi installation documentation isn't present.

There will undoubtedly be more issues, but these aren't show stoppers and can be fixed by changing the Kernel configuration.

*You should consider using the Raspberry Pi Kernel fork as an **experimentation** rather than a suggestion or endorsement.*

Raspberry Pi & alternate Kernel Fork: automated build system

To simplify building the Raspberry Pi Kernel fork, you should follow this guide.

You can also use this guide to help get started using the script for building Slackware packages from alternate source repositories, which helps when Hardware Model vendors have yet to upstream their changes to the mainline Linux Kernel.

Introduction

If you'd like to customise the Linux Kernel for your system - perhaps to add some hardware support or a Kernel feature, or perhaps add a patch, the easiest way is to use the Slackware ARM Kernel package build scripts. This is because the Kernel and Operating System Initial RAM Disk are tightly coupled.

If you are rebuilding your Kernel to enable some additional hardware support or a Kernel feature, please drop a note to the forum. The Slackware ARM/AArch64 Kernels were based on default platform configurations, but the aim is to have behavioural and feature parity with Slackware x86/64 where possible. Usually any Kernel configuration change requests will be included within the official Slackware Kernel package.

The Slackware ARM/AArch64 build system

The Slackware ARM build system was built to abstract the common functionality of the build scripts into a library and configuration file, enabling global changes to be made in a single location.

The Slackware ARM source tree works as an 'overlay' to the Slackware x86/64 upstream source tree. This way all assets (sources, patches, etc.) are *referenced* from the Slackware ARM build system, but are never duplicated. The Slackware ARM build scripts (called `<packagename>.SlackBuild`) are however customised to operate within the Slackware ARM build system, but produce identical packages to the x86 upstream versions. This approach enables architecture customisations to be easily made, whilst minimising effort required to synchronize source assets.

Assumptions

This document assumes that you're building on a full Slackware system, because the Slackware Linux Kernel package build process requires a number of development tools to be available.

This document provides some basic instructions using the `slackpkg` tool to ensure that your system has the latest versions of these packages available.

You'll find how to setup slackpkg here

If you don't have `slackpkg` set up or would prefer to do this manually, you may use the `upgradepkg`

–`install-new` tool instead.

This document covers working on Slackware `-current` (the development branch), but you can use the same steps for a stable release by switching the name of the tree - e.g. `-current` → `-15.0`.

Preparing the environment

Download the Slackware AArch64 source tree

Within your home directory, create a directory to contain the Slackware AArch64 source tree:

```
$ cd # return to the root of your home directory (not the 'root' user):
$ mkdir slackware ; cd slackware
```

Determine where you are in the file system, as you will need this shortly:

```
$ pwd
/home/mozes/slackware
```

```
rsync \
   --delete -Pavv \
   ftp.arm.slackware.com::slackwarearm/slackwareaarch64-current .
```

Note the full stop/period on the end of the command - this denotes the present working directory

The full 'slackware' tree is required because the Kernel build script requires some of the package files to be available during build time, as certain tools and libraries are extracted from them to build the OS InitRD (Operating System Initial RAM Disk).

Install the prerequisite packages

Before the Kernel can be built, you need to ensure that you have certain packages installed. This list isn't fully inclusive because you should build the Kernel on a full system, but particularly for Slackware -current, you need to ensure that you have the latest version of the Kernel packages installed so that it's aligned with the Slackware AArch64 source tree.

If the packages aren't available for installation, use the `upgrade` option (`slackpkg upgrade <pkgname>`) to slackpkg instead to ensure that they're up to date.

Update slackpkg's cache

As **root**:

```
$ slackpkg update
```

Install the Slackware ARM development kit

The Slackware ARM build system uses a package named `slackkit` that contains the library functions to build the packages. This package is installed by default, but if you don't have it :

As **root**, install/upgrade the package:

```
$ slackpkg install slackkit
```

Ensure that the Kernel source package is up to date

As **root**, install/upgrade the package:

```
$ slackpkg install kernel-source
```

Install other dependencies

This list isn't exhaustive but should cover it (if more dependencies are required, please drop mozes@slackware.com a mail with them and I'll update this doc!)

```
$ slackpkg install eudev mdadm lvm2 device-tree-compiler rsync
```

Using an alternative Kernel source tree - e.g. Raspberry Pi Kernel fork

If you want to use an alternative Kernel source tree rather than the mainline Kernel that Slackware uses, this is easy to do.

If you want to use the mainline standard Kernel, skip this section and move to the next (`Configure the Slackware ARM build system`).

For this example we'll use the Raspberry Pi Kernel fork but you can easily see how to adjust this for any other Kernel fork.

As your normal (not root) user, download the RPi Kernel fork:

```
$ cd /tmp ; mkdir rpikernel ; cd rpikernel
$ git clone --depth=1 --branch rpi-6.1.y
```

```
https://github.com/raspberrypi/linux.git linux-rpi-6.1.y
```

The Raspberry Pi Kernel fork is classified as 'dirty' because it contains code that isn't committed. Remove the git structure to prevent the RPi Kernel being labeled as such (because it breaks the Slackware ARM naming scheme for /lib/modules):

```
$ find . -name '.git*' -print0 | xargs -0 rm -rf
```

Rename the Kernel source directory back to the standard mainline naming convention:

```
$ pushd linux-rpi-*
$ kver=$( echo "$(sed -ne's/^VERSION *= *//p' Makefile).$(sed -
ne's/^PATCHLEVEL *= *//p' Makefile).$(sed -ne's/^SUBLEVEL *= *//p'
Makefile)" )
$ popd
$ mv -fv linux-rpi-* linux-$kver
```

As **root**, move this directory into the standard location `/usr/src`

```
$ mv -fv /tmp/rpikernel /usr/src/
```

Make a note of this Kernel version as you'll need it later to ensure the version is aligned with the Slackware ARM build system

Configure the Slackware ARM build system

In this example the directory into which the Slackware AArch64 source tree was downloaded is `/home/mozes/slackware` - change this to what you noted down earlier

As **root**, you need to set up some symlinks to the location of your Slackware AArch64 source tree.

```
$ cd # return to root's home directory
$ ln -vfs /home/mozes/slackware/slackwareaarch64-current/slackware tgzstash
$ ln -vfs /home/mozes/slackware/slackwareaarch64-current
```

Preparing and Customising the Linux Kernel source

The Slackware Linux Kernel package `kernel-source` installs the source code for the official Slackware AArch64 Kernel package within versioned directory `/usr/src/linux-<x>.<y>.<z>`, and sets up a symlink to that versioned directory as `/usr/src/linux`.

The Linux source directory **must** be named `linux-<version>` - this is the mainline

Kernel naming convention which is hard coded within the `kernel.SlackBuild`. If you are using a Kernel fork with a different naming convention, you should rename it to match the standard (as performed in the earlier example when using the Raspberry Pi Kernel fork).

Ensure version numbers align

The version of the Linux Kernel contained within the Slackware AArch64 source tree must match that within `/usr/src`. If you're using Slackware -current and have followed the instructions above, the versions should align but it's worth checking.

Check the version configured within the Slackware AArch64 source tree:

```
$ egrep '^export VERSION.*' /home/mozes/slackware/slackwareaarch64-
current/source/k/arm/build
export VERSION=${VERSION:-6.1.27}
```

Remember to use the correct path to your Slackware AArch64 source tree

When using an alternate Kernel source (such as the Raspberry Pi's) these versions will usually be out of sync. In which case, you should adjust the `arm/build` script to match the Kernel source.

Check the version of the Linux source directory:

```
$ cd /usr/src/
$ ls -d linux-* -la
drwxr-xr-x 24 root root 4096 Jun 30 10:41 linux-6.1.27/
```

The versions in this example are aligned, but if not you should make them so - usually by changing the version within the 'arm/build' script.

Make your changes

The Slackware `kernel-source` package has a number of patches (see `slackwareaarch64-current/source/k/patches/`) applied to it already.

The Slackware `kernel-source` package contains the Kernel configuration file (named `.config`) used to build the official Kernel packages. This configuration file can be found within the Slackware AArch64 source tree at: `slackwareaarch64-current/source/k/configs/config-armv8`

At this point you can use the Linux Kernel configuration tools to change the configuration, or perhaps

apply a patch.

Example:

```
$ cd /usr/src/linux-*
$ make menuconfig
```

There are other configuration tools:

```
$ make nconfig # another curses-based tool
$ make xconfig # an X configuration tool
```

Using an alternative Kernel source tree - e.g. Raspberry Pi Kernel fork

If you want to use the mainline standard Kernel, skip this section and move to the next (“`Preserve the modified Kernel configuration`”).

If you're using an alternative Kernel source such as the Raspberry Pi's, you need to make a few adjustments.

As **root**:

```
$ cd /usr/src/rpikernel/linux-*/
$ make bcm2711_defconfig # this uses the default configuration for the SoC
the RPi4 uses
$ make menuconfig # make any changes you want
```

Switch the 'local version' to the standard Slackware ARM uses:

```
$ sed -i 's?^CONFIG_LOCALVERSION=.*?CONFIG_LOCALVERSION="-armv8"?g' .config
$ sed -i 's?^# CONFIG_LOCALVERSION_AUTO.*?CONFIG_LOCALVERSION_AUTO=y?g'
.config
```

Preserve the modified Kernel configuration

When you next upgrade the Slackware `kernel-source` package, it will overwrite your Kernel configuration, so it's best to preserve it.

This is also relevant if you're using an alternative Kernel source or the standard Slackware mainline Kernel:

```
$ cp .config ../custom-config
```

Building Kernel Packages

Now that you have customised the Kernel, it's time to build the Kernel packages.

As **root**:

Create a directory to receive the new packahges

To avoid overwriting the official packages within the Slackware tree, we will store them in an alternate location.

```
$ mkdir -p /tmp/testpkgs
```

Enter the Slackware ARM Kernel source directory

```
$ cd ~/slackwareaarch64-current/source/k
```

Build the Kernel packages

The `kernel.SlackBuild` Kernel build system replaces /lib/modules/<Kernel version you're building>. If the Kernel version number you're building is identical to the version of the running Kernel (use `uname -r` to check), you **must** reinstall the `a/kernel-modules` package belonging to your running Kernel. If not your machine will most likely fail to boot properly, with the Kernel complaining about magic numbers being out of alignment when loading a variety of Kernel modules.

For the standard/mainline Slackware Kernel source:

```
$ ./arm/build --srcdir /usr/src --pkgstoreoverride /tmp/testpkgs --noconfig
--nopatches
```

For an alternate Kernel source tree such as the Raspberry Pi's, you should do this:

```
$ ./arm/build --srcdir /usr/src/rpikernel --pkgstoreoverride /tmp/testpkgs -
-noconfig --nopatches
```

The meaning of the command line options: `–srcdir` instructs the kernel.SlackBuild to use the source tree `linux-<version>` found within `/usr/src/linux` rather than unpacking the source archive contained within the Slackware AArch64 source tree.

`–noconfig` instructs the kernel.SlackBuild not to install the Slackware Kernel

configuration (as this would overwrite your customised version).

`–nopatches` instructs the kernel.SlackBuild not to apply its patch set, since it's already been applied. In the case of using the Raspberry Pi Kernel fork, there's no need to apply the patches for the mainline Kernel.

`–pkgstoreoverride` instructs the kernel.SlackBuild to store the new packages in the directory specified.

The Kernel packages will build.

Build times

Building the standard Slackware AArch64 Linux Kernel Natively on a RockPro64 takes approximately 4.5 hours.

Building the Raspberry Pi Kernel fork (with the default configuration) natively on a Raspberry Pi 4 takes approximately 3 hours.

Building the standard Slackware 32bit ARM 15.0 Kernel natively on an Orange Pi Plus2E takes approximately 11 hours.

Loading Kernel modules at boot

If you'd like to load some Kernel modules at boot, there are two options:

1. Load them from the Operating System Initial RAM Disk ('OS InitRD'). This is generally only required if you need some specific hardware support early in the boot process. For this, you can create a loader script:

`/boot/local/load_kernel_modules.post`

See `/boot/local/README.txt` for more information.

The Slackware Kernel build script (`kernel.SlackBuild`) packages a generous collection of Kernel modules for the major sub systems, but it's possible that the module you want to load isn't present. If this is the case, you'll need to prevent the build script from deleting the Kernel modules. Within `kernel.SlackBuild` find the section that begins with the comment `# Slim down the modules within the initrd by removing` and edit as necessary. Typically this shouldn't be necessary.

2. Add them to load within the Operating System proper by editing `/etc/rc.d/rc.modules`.

Installing/upgrading to the new Kernel packages

If the build process worked, your Kernel packages will have been installed into the directory specified (`/tmp/testpkgs`). As **root**:

```
$ cd /tmp/testpkgs
```

The Kernel packages will have been installed within their respective 'package series' directories:

`a series` - base packages

```
a/kernel_armv8   -- base Kernel package: Kernel, OS InitRD, DTBs
a/kernel-modules   -- software drivers for hardware support and Kernel
features
```

`d series` - development:

```
d/kernel-headers
```

`k series` - Kernel source:

```
k/kernel-source
```

Generally you'll only want to upgrade the runtime packages:

```
$ upgradepkg a/kernel{-modules,_armv8}*-*.t?z
```

Any customisations you made to the OS InitRD (Operating System Initial RAM Disk) within `/boot/local` will be applied when upgrading to your new packages, because `os-initrd-mgr` is called from the post installation script of the `kernel_armv8` package.

You may receive warnings or errors from `modinfo` whilst upgrading the packages. These are from `os-initrd-mgr` as it processes the OS Initial RAM Disk and occurs if your new Kernel doesn't contain the same modules as the running one (which is particularly the case for the RPi fork). You can ignore these errors - they shouldn't appear during any subsequent runs.

You can now reboot into your new Kernel.

Further Customisations

When you next want to customise the Kernel configuration, you can refresh your previously customised Kernel configuration:

```
$ cd /usr/src/linux
$ cp ../custom-config .config
$ make oldconfig
$ make menuconfig # edit the configuration, or perhaps apply a patch
$ cp .config ../custom-config # preserving it again
```

When using an alternative Kernel source such as the Raspberry Pi's, change the directory name accordingly

Then follow the build process again to build new Kernel packages.

Further notes about using the Raspberry Pi Kernel fork

If you are familiar with the Raspberry Pi Kernel fork, you may be aware that the Device Tree Overlays are not built from the Slackware AArch64 Kernel build script (`kernel.SlackBuild`). This is because these aren't present within the mainline Kernel. In Slackware these overlays are maintained within a separate package.

On the same system upon which you have built the Kernel, you can change into the `hwm-bw-raspberrypi` package source directory, make changes and build a new package:

```
$ cd ~/slackwareaarch64-current/source/a/hwm-bw-raspberrypi/
```

You will find the sources and source download scripts within this directory. Once you have made your changes, build a new package.

```
$ mkdir -p /tmp/testpkgs
$ ./arm/build --pkgstoreoverride /tmp/testpkgs
```

As with the Kernel packages, your new `hwm-bw-raspberrypi` package will be stored within `/tmp/testpkgs` from where you can install it with `upgradepkg`.

```
$ cd /tmp/testpkgs/a
$ upgradepkg hwm-bw-raspberrypi-*.t?z
```

Only the `source/k` and `source/a/hwm-bw-raspberrypi` build scripts support the `–pkgstoreoverride` command line operator. To adjust the output directory for other packages, you should edit their `arm/build` scripts.