Slackware ARM GCC aarch64-linux cross-compiler for the Raspberry Pi

Preface

I was thinking about the Cortex-A53 64-bit CPU on my Raspberry Pi 3 and why I'm mainly using Slackware ARM 32 bit operating system on it. Then I started to wonder if it would be possible to build an arm64 kernel and modules to run with Slackware ARM. After reading about how this could be achieved it seemed clear that some cross-compiling would be required. Although I have some experience in building Linux kernels, especially for the Raspberry Pi platform, I'd never done any cross-compiling before a week ago (2016-12-15). So, this whole concept was brand new to me.

For my first attempt (and largely based on reading about how other users were doing it) I used an Ubuntu 16.04.1 LTS (64 bit) system to cross-compile an arm64 kernel for the Raspberry Pi 3. However, the results of doing things by this method were somewhat lacking and created many unforeseen errors. Then I remembered something Mozes had published on his Slackware ARM FAQ page about packages being built natively. Further investigation lead me to realise that assured success would most likely be found by cross-compiling on an ARM device using Slackware ARM. So, that's exactly what I did! Very successfully, might I add. Thanks again, Mozes. <3

As things have turned out, it wasn't *that* difficult. Investing some time into reading about toolchains and how to build cross-compilers was required, as well as testing the results of cross-compiling, but on the whole it's been a relatively simple process. Using Slackware ARM current to cross-compile aarch64 architecture was the key to success here. I'm now very aware that, in comparison, trying to cross-compile aarch64 on an Ubuntu x86_64 system was less than productive.

Notes

Slackware ARM current was used on a Raspberry Pi 3 to build and install the GCC aarch64-linux cross-compiler, and build the arm64 Linux kernel, modules, and device tree blob(s). That's not to say Slackware ARM 14.2 won't work too, but I just didn't do any cross-compiling on the soft float port. The same applies to the Raspberry Pi 1 and 2. Even though it *should* be possible to carry out aarch64 cross-compiler builds on these devices, I didn't do any testing with them. Also bear in mind that configuration options and settings will need to be considered first.

Requirements

As a pre-requisite, you should have;

- a Raspberry Pi 3 running Slackware ARM current with approx. 8GB unused space on your system.
- gawk, git, bison and flex, already installed on your system.
- a USB microSD card reader to connect with your Raspberry Pi 3.
- a (spare) microSD card with Slackware ARM current installed on it. *# not essential but it is advised.*

What's involved

This tutorial will enable you to;

- download the required package source(s) in order to build a GCC cross-compiler on Slackware ARM.
- download the Raspberry Pi Linux kernel GitHub tree and switch to the 64 bit development branch.
- configure, and install, a GCC aarch64-linux (arm64) cross-compiler on your Raspberry Pi 3.
- build an aarch64 (arm64) Linux kernel, modules, and device tree blob(s), and install them on your (spare) Slackware ARM current microSD card.
- successfully boot Slackware ARM current on your Raspberry Pi 3 running an aarch64 (arm64) kernel.

In order not to risk messing up the Slackware ARM system which you use for cross-compiling, a (spare) microSD card containing a working Slackware ARM system should be used to install the arm64 Linux kernel, modules, and device tree blob(s). You'll need a recent (i.e. post-September 2016) version of the Raspberry Pi bootloader/GPU firmware installed on this (spare) microSD card to avoid any problems. To be sure, boot your RPi3 with it and update the firmware. The Linux kernel version on this (spare) microSD card doesn't matter as you'll be replacing it with the aarch64 (arm64) kernel.

Downloading required source and configuration

First of all, as a normal user (i.e. not 'root') create a working directory. For example, I usually work from the /tmp directory:

```
cd /tmp
mkdir build-dir
cd build-dir
```

This is the directory you will download all the required packages and RPi Linux source to.

Downloading RPi Linux kernel source

Use the following 'git' command to download the Raspberry Pi Linux kernel source into a directory named 'linux'.

```
git clone https://github.com/raspberrypi/linux linux
```

This may take a while, depending on the speed of your Internet connection and other factors. Once it has completed you need to switch to the 64 bit kernel development branch.

```
cd linux
git checkout rpi-4.8.y
```

When that's done you should see a message that 'origin/rpi-4.8.y' is the current branch.

Downloading required package source

Before downloading the package source needed to build the GCC cross-compiler, be aware that more recent package versions may exist than the ones shown here. You may wish to install newer versions. It's always a good idea to check. To keep things simple, you might consider downloading a version of GCC which matches the one you currently have installed. I've read lots of articles about this and most advise to install the *latest and greatest* version of GCC available. However, if you're running Slackware ARM current you'll have gcc-5.4.0 installed and this is adequate for what you need.

So, first move back into the 'build-dir' directory and then download the packages below.

```
cd ../
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-2.27
wget -nc ftp://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1
wget -nc https://ftp.gnu.org/gnu/gcc/gcc-5.4.0
wget -nc https://ftp.gnu.org/gnu/glibc/glibc-2.24
wget -nc https://ftp.gnu.org/gnu/gmp/gmp-6.1.1
wget -nc ftp://gcc.gnu.org/pub/gcc/infrastructure/isl-0.16.1
wget -nc https://ftp.gnu.org/gnu/mpc/mpfr-3.1.5
wget -nc https://ftp.gnu.org/gnu/mpfr/mpc-1.0.3
```

Unpacking downloaded tarballs

Now unpack all the downloaded tarballs. You can do this easily with a 'for loop' command.

```
for t in *.tar*; do tar -xvf $t; done
```

Once this has completed you can us the '**ls**' command to verify that the directories are present.

Creating GCC dependency symlinks

Now you are going to create some symbolic links in the gcc-5.4.0 directory. These will point to some of the source directories you have just unpacked, which are dependencies of GCC, and when these symbolic links are present GCC will build them automatically.

```
cd gcc-5.4.0
ln -sf ../cloog-0.18.1 cloog
ln -sf ../gmp-6.1.1 gmp
ln -sf ../isl-0.16.1 isl
ln -sf ../mpfr-3.1.5 mpc
ln -sf ../mpfr-3.1.5 mpfr
```

Alternatively, some articles will advise you to use the following command in order to achieve the same thing.

```
cd gcc-5.4.0
./contrib/download_prerequisites
```

Personally, I always prefer the manual method because then I know what's being downloaded/installed and what to expect. It's up to you which method you use.

Creating GCC cross-compiler install directory

The next thing to do is create an installation directory. This is the directory where the toolchain will be installed. Again, I like to work in /tmp so the install directory will be created there.

```
cd /tmp
mkdir gcc-cross
```

Exporting install directory PATH

You need to export the installation directory's /bin folder to your user's $PATH.

```
export PATH=/tmp/gcc-cross/bin:$PATH
```

To check that this has worked, use the following command:

```
echo $PATH
```

You should see the first $PATH entry is to your installation directory's /bin folder. It's important that your installation directory's /bin folder appears before any other entry in the $PATH.

```
/tmp/gcc-
cross/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/kde4/libexec:/usr
/lib/qt/bin
```

Building the GCC aarch64 cross-compiler

Now with all that in place, concentration focuses on building the cross-assembler, cross-disassembler, cross-linker, and other useful tools.

Building binutils

First move back into the 'build-dir' directory and then create a build directory for binutils. You'll notice

the various build options but as a quick explanation; '–with-sysroot' basically tells binutils to enable 'sysroot' support in the cross-compiler by pointing it to a default empty directory, '–target=aarch64-linux' is the target system type (arm64), and '–disable-multilib' means that we only want binutils to work with the aarch64 instruction set and nothing else.

```
cd build-dir
mkdir build-binutils
cd build-binutils
../binutils-2.27/configure --prefix=/tmp/gcc-cross --with-sysroot --
target=aarch64-linux --disable-multilib
make -j4
make install
```

Installing Linux kernel headers

Here you need to install the Linux kernel headers. Note the 'ARCH=arm64' option for the make process. GCC uses 'aarch64' where the Linux kernel uses 'arm64'. The two separate open source projects identify the same CPU architecture differently.

```
cd ../linux
make ARCH=arm64 INSTALL_HDR_PATH=/tmp/gcc-cross/aarch64-linux
headers_install
```

Build GCC

First move into the 'build-dir' directory and create a build directory for GCC before building it. Notice that only C and C++ have been specified as build languages. That's all you will need here. Incidentally, the available build language options allow just one, or a selection, or all, of the following '–enable-languages=all,ada,c,c++,fortran,go,jit,lto,objc,obj-c++'.

```
cd ../
mkdir build-gcc
cd build-gcc
../gcc-5.4.0/configure --prefix=/tmp/gcc-cross --target=aarch64-linux --
enable-languages=c,c++ --disable-multilib
make -j4 all-gcc
make install-gcc
```

Build and install glibc

First move into the 'build-dir' directory and create a 'build-glibc' directory. Then move into the 'build-glibc' directory before building it. '–build=$MACHTYPE' is a predefined environment variable which describes the Raspberry Pi 3 (in this case) and it's required to compile some additional tools which are utilised during the build process. Notice that you're installing the C library startup files to the installation directory (csu/crt1.o, csu/crti.o, and csu/crtn.o) separately because there doesn’t seem to a 'make' rule that does this without creating other problems.

```
cd ../
mkdir -p build-glibc
cd build-glibc
../glibc-2.24/configure --prefix=/tmp/gcc-cross/aarch64-linux --
build=$MACHTYPE --host=aarch64-linux --target=aarch64-linux --with-
headers=/tmp/gcc-cross/aarch64-linux/include --disable-multilib
libc_cv_forced_unwind=yes
make install-bootstrap-headers=yes install-headers
make -j4 csu/subdir_lib
install csu/crt1.o csu/crti.o csu/crtn.o /tmp/gcc-cross/aarch64-linux/lib
$ARCH_TARGET-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /tmp/gcc-
cross/aarch64-linux/lib/libc.so
touch /tmp/gcc-cross/aarch64-linux/include/gnu/stubs.h
```

Building glibc support library

Now move into the 'build-gcc' directory once again and build the GCC cross-compiler support library.

```
cd ../build-gcc
make -j4 all-target-libgcc
make install-target-libgcc
```

Finish building glibc C library

Move into the 'build-glibc' directory to finish building glibc C library and then install it.

```
cd ../build-glibc
make -j4
make install
```

Finish building GCC C++ library

Move into the 'build-gcc' directory to finish building GCC C++ library and then install it.

```
cd ../build-gcc
make -j4
make install
```

Testing the cross-compiler

To test/check that your GCC aarch64-linux cross-compiler is working properly run the following command.

```
aarch64-linux-gcc -v
```

You should get a response similar to the following.

```
Using built-in specs.
COLLECT_GCC=aarch64-linux-gcc
COLLECT_LTO_WRAPPER=/tmp/gcc-cross/libexec/gcc/aarch64-linux/5.4.0/lto-
wrapper
Target: aarch64-linux
Configured with: ../gcc-5.4.0/configure --prefix=/tmp/gcc-cross --
target=aarch64-linux --enable-languages=c,c++ --disable-multilib
Thread model: posix
gcc version 5.4.0 (GCC)
```

Once this process has been completed, export the GCC cross-compiler PATH on your normal user. If/when you're wanting to cross-compile do this each time after you've (re)booted your system so that the GCC cross-compiler can be located via your user's $PATH. You also have the option to add this command to your **~/.profile** as a permanent setting. Whether or not you decide to permanently add the GCC cross-compiler PATH to your ~/.profile is entirely up to you. If you are using your Slackware ARM current system for exclusively building aarch64 (arm64) packages then it would make sense to do so.

Example export command:

```
export PATH=/tmp/gcc-cross/bin:$PATH
```

The GCC aarch64-linux cross-compiler on your Slackware ARM system is now ready to *rock-n-roll!*

Building the arm64 kernel, modules, and device tree blob (DTB)

To build the aarch64 kernel, modules and device tree blob(s) is exactly the same method as you would carry it out under normal circumstances. Commands such as 'make bzImage && make modules && make modules_install' may be all too familiar to you. The major difference when cross-compiling is that you'll use certain Makefile options/variables/arguments/switches, commonly known as CFLAGS. In our case, CFLAGS will be used to instruct the GCC cross-compiler to build for the aarch64 (arm64) architecture specifically.

Creating the arm64 kernel .config

First of all, as always, you need to be in the Raspberry Pi Linux kernel source directory which is in the 'build-dir' folder. Then you need to create a kernel .config file, based on Raspberry Pi 3 parameters. To keep it simple you can generate a default .config (**defconfig**) file. This file holds the Linux kernel configuration for the arm64 kernel you are going to build. To achieve this run the following commands:

```
cd /tmp/build-dir/linux
make -j4 ARCH=arm64 CROSS_COMPILE=aarch64-linux- bcmrpi3_defconfig
```

Make a note here of the CFLAGS which have been specified. They should be self-explanatory by now. Pay special attention to the trailing '-' of 'CROSS_COMPILE=aarch64-linux-' because that's **NOT** a typo. It needs to be like that!

Building the arm64 kernel

Next up is building the kernel, based on the .config file you have just created. Again, you'll use the same CFLAGS as before. You can even set a 'LOCALVERSION' here which appends whatever you set to the end of the kernel version (e.g. LOCALVERSION=“-arm64” would eventually give you 4.8.13-v8–arm64) once the kernel and modules have been built. Just as an example we'll use it here. Run the following command to start building the arm64 Linux kernel:

```
make -j4 Image ARCH=arm64 CROSS_COMPILE=aarch64-linux- LOCALVERSION="-arm64"
```

So, here you 'make' the kernel which will be saved with the name '**Image**'. The rest you should be familiar with. This process will take a while. Maybe an hour or so.

Building the arm64 device tree blob(s)

Device tree is a means of describing hardware which is read by the kernel at boot time to tell it what hardware exists on the system. In our case it relates to the Raspberry Pi 3 and is the method by which the systems knows which drivers to load for the hardware. On ARM-based devices the use of device trees has become mandatory for all new SOCs, including the Raspberry Pi. The device tree blobs you will be building are '**bcm2710-rpi-3-b.dtb**' and '**bcm2837-rpi-3-b.dtb**'

To build the Raspberry Pi device tree blob(s) run the following command:

```
make -j4 dtbs ARCH=arm64 CROSS_COMPILE=aarch64-linux- LOCALVERSION="-arm64"
```

It's basically the same as you did to build the kernel, only where 'Image' is substituted for '**dtbs**'.

Building the arm64 modules

To build the kernel modules you do it in much the same way as before. Run the following command:

```
make -j4 modules ARCH=arm64 CROSS_COMPILE=aarch64-linux- LOCALVERSION="-arm64"
```

Notice how each time the command is the same except when specifying what you're building. If you have set a 'LOCALVERSION' then it must be kept the same for building the kernel and modules. This process will probably take a while longer than building the arm64 kernel.

Installing the arm64 modules

Once the modules have been built, you have to 'make modules_install'. The process will install your kernel modules to '/tmp/lib/modules/4.8.13-v8'.

You could build *out-of-tree* kernel modules but, to keep things simple, you're going to install them to the usual location. Again, you will use the same CFLAGS as before but without any 'LOCALVERSION' set.

First you need become '**root**' user and enter a passwd when prompted. To install the aarch64 modules run the following commands:

```
su -
make -j4 modules_install ARCH=arm64 CROSS_COMPILE=aarch64-linux-
```

You need to be '**root**' user to install the arm64 modules. A normal user does not have the rights to do so!

So, as I'm a great believer in being thorough, I always verify things at every opportunity. Just to be sure, if nothing else, because it's always a good policy. Make sure the files and directories you have just spent quite a bit of time compiling actually do exist on your system and that they're in the right place. If this is the first time you have installed the GCC cross-compiler on your system and/or built the kernel, modules, and device tree blob(s), then it goes without saying. You could actually do this after each build process, which I often do as well.

```
ls -lah arch/arm64/boot/Image
la -lah arch/arm64/boot/dts/broadcom/bcm*-rpi-3-b.dtb
ls -lah /lib/modules/4.8.13-v8*
```

If you can see that they all exist, then everything has worked as planned.

Copying the arm64 kernel, modules, and device tree blobs (DTB)

Connect the (spare) microSD card containing a working Slackware ARM current system to your Raspberry Pi 3 using a USB microSD card reader. You'll need to mount the partitions first, before copying the arm64 kernel, modules, and device tree blob(s) onto it.

You should still be logged in as '**root**' user. If not, type the following command and enter the passwd for the 'root' user when prompted:

```
su -
```

You need to be '**root**' user to carry out any mount procedures. A normal user does not have the rights to do so!

As 'root' user type the following command:

```
fdisk -l
```

This should show you which device the (spare) microSD card is using on your system. In our case it's a 32GB card and has been identified as '**/dev/sda**', as shown below. This tells us that '**/dev/sda1**' is our /boot partition and '**/dev/sda3**' is our root filesystem partition. Yours may be allocated differently so bear that in mind.

```
Device     Boot  Start      End  Sectors  Size Id Type
/dev/sda1  *        32   195327   195296 95.4M  c W95 FAT32 (LBA)
/dev/sda2       196608   720895   524288  256M 82 Linux swap
/dev/sda3       720896 61145087 60424192 28.8G 83 Linux
```

In order to mount the partitions you first need to create mount directories. Working in the /tmp directory you can do it like this:

```
cd /tmp
mkdir rpi-boot
mkdir rpi-root
mount /dev/sda1 rpi-boot
mount /dev/sda3 rpi-root
```

To check that you've done this correctly, use the '**mount**' command. The output from this should give you something similar to the following:

```
/dev/mmcblk0p3 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/mmcblk0p1 on /boot type vfat (rw,fmask=177,dmask=077)
/dev/sda1 on /tmp/rpi-boot type vfat (rw)
/dev/sda3 on /tmp/rpi-root type ext4 (rw)
```

The next thing to do is copy the arm64 kernel, modules, and device tree blobs to your newly mounted directories. It's important to get this right. After all the hard work you've done it would be a shame to mess it up at this stage.

To copy these files, run the following commands, as '**root**' user:

```
cp build-dir/linux/arch/arm64/boot/Image rpi-boot/boot/kernel8.img
cp build-dir/linux/arch/arm64/boot/dts/broadcom/bcm*-rpi-3-b.dtb rpi-
boot/boot
cp -rv /lib/modules/4.8.13-v8* rpi-root/lib/modules/
```

Once you have done that, check that the files you have copied are present and in the right place.

```
ls -lah rpi-boot/boot/kernel*
```

```
ls -lah rpi-boot/boot/bcm*-rpi-3-b.dtb
ls -lah rpi-root/lib/modules/4.8.13-v8*
```

If it all looks fine and dandy then the next thing you need to do is delete the old armv7 kernel in the rpi-boot/boot directory. This old kernel is named '**kernel7.img**' and to avoid any conflicts with the new arm64 'kernel8.img' you should remove it.

```
rm -rf rpi-boot/kernel7.img
```

No changes to the config.txt or cmdline.txt file(s) should be necessary. If you are using a recent blootloader/GPU firmware version (i.e. post-September 2016) then nothing else needs to be changed or deleted. The system should boot using all your existing settings.

Now you can unmount the previously mounted directories.

```
umount rpi-boot
umount rpi-root
```

Booting Slackware ARM aarch64

Power off your Raspberry Pi.

```
poweroff
```

Remove the USB microSD card reader and swap microSD cards. Power on the Raspberry Pi and boot the microSD card on which you copied the arm64 kernel, modules, and device tree blobs.

The end result

After booting the system with the arm64 kernel, I logged in remotely via SSH as 'root' user. Then I ran the following commands:

```
login as: root
root@192.168.10.33's password:
Last login: Sat Dec 17 20:32:50 2016 from 192.168.10.10
Linux 4.8.13-v8-arm64.
root@drie:~# cat /proc/version
Linux version 4.8.13-v8-arm64 (exaga@drie) (gcc version 5.4.0 (GCC) ) #2 SMP
Fri Dec 16 18:43:38 GMT 2016
root@drie:~# uname -a
Linux drie 4.8.13-v8-arm64 #2 SMP Fri Dec 16 18:43:38 GMT 2016 aarch64
GNU/Linux
root@drie:~# cat /etc/slackware-version
Slackware 14.2
root@drie:~# cat /proc/device-tree/model
Raspberry Pi 3 Model B Rev 1.2
root@drie:~# cat /proc/cmdline | awk -v RS=" " -F= '/serial/ { print $2 }'
0x4135b94e
```

```
root@drie:~#
```

Although I've already come across a few things which need some work, and attention, it's a start. I hope to find more time to devote towards Slackware arm64 over Christmas and the New Year 2017.

Thanks for being interested. <3

Sources

ftp://ftp.arm.slackware.com/slackwarearm/slackwarearm-current/slackware/a/gawk-4.1.4-arm-1.txz # Slackware ARM current - gawk package.
ftp://ftp.arm.slackware.com/slackwarearm/slackwarearm-current/slackware/d/git-2.11.0-arm-1.txz # Slackware ARM current - git package.
ftp://ftp.arm.slackware.com/slackwarearm/slackwarearm-current/slackware/d/bison-3.0.4-arm-1.txz # Slackware ARM current - bison package.
ftp://ftp.arm.slackware.com/slackwarearm/slackwarearm-current/slackware/d/flex-2.6.0-arm-1.txz # Slackware ARM current - flex package.
http://arm.slackware.com/FAQs # Slackware ARM Linux Project Frequently Asked Questions.
http://wiki.osdev.org/GCC_Cross-Compiler # GCC cross-compiler documentation.
https://www.raspberrypi.org/documentation/linux/kernel # Raspberry Pi Linux kernel documentation.
https://www.github.com/raspberrypi/ # Raspberry Pi Foundation GitHub repository Linux kernel, bootloader/GPU firmware.
https://ftp.gnu.org/gnu/ # GCC, binutils, glibc, gmp, mpc, mpfr package source
ftp://gcc.gnu.org/pub/gcc/infrastructure # cloog, isl package source

- Originally written by Exaga

howtos, hardware, aarch64, cross-compile, author exaga