Jensd's I/O buffer

random technotes…

Cross compiling for arm or aarch64 on Debian or Ubuntu

Posted on **26/01/2021**

ARM is gaining more and more traction and is growing a lot in popularity. It's not always possible to build directly on these ARM-based devices, especially when they are limited in resources. The majority of build and developer machines are still on x86 and by using cross compiling, it is possible to build binaries or executables usable on another architecture. For example, to use your standard PC, most likely x86, to build something that is usable on another machine or device that's on another architecture, like ARM. In this post, I'll explain how to do cross compiling for 32bit ARM (arm) or 64bit ARM (aarch64) using Debian 10 or Ubuntu 20.04 LTS.

Youtube video

If you are interested, I also created a YouTube video from this blogpost. If you prefer classic text, you can just follow the rest of this article:

Cross compiling for arm or aarch64 on Debian or Ubuntu

Introduction

The ability to cross compile, for me, is most used to build troubleshooting tools that are not installed or available on Linux-based devices. For example a device like a Raspberry Pi, NAS, router or an access point that has a custom Linux build without or limited option to install additional packages.

Sample output of a random embedded device running Linux…:

```
/ # telnet
/bin/sh: telnet: not found
/ # tcpdump
/bin/sh: telnet: not found
/ # strace
/bin/sh: strace: not found
/ # curl
/bin/sh: curl: not found
```

For the steps below, I will be using Debian 10 (Buster) and I will also test the same steps on Ubuntu 20.04.1 (LTS). All steps are verified to be interchangeable between both. The starting point for both is a minimal installation (standard system utilities + SSH server). This makes sure that, if anyone wants to repeat these steps, all is reproducible and nothing is skipped or missed that would be preinstalled already.

Terminology

In cross compiling, the following (confusing) terminology is used:

- Build platform: Architecture of the build machine
- Host platform: The architecture you are building for
- Target platform: The architecture that will handle the compiled binaries

Build and host are more or less clear but target can be confusing. Simply put, target is only relevant when working on development tools (like the compiler itself).
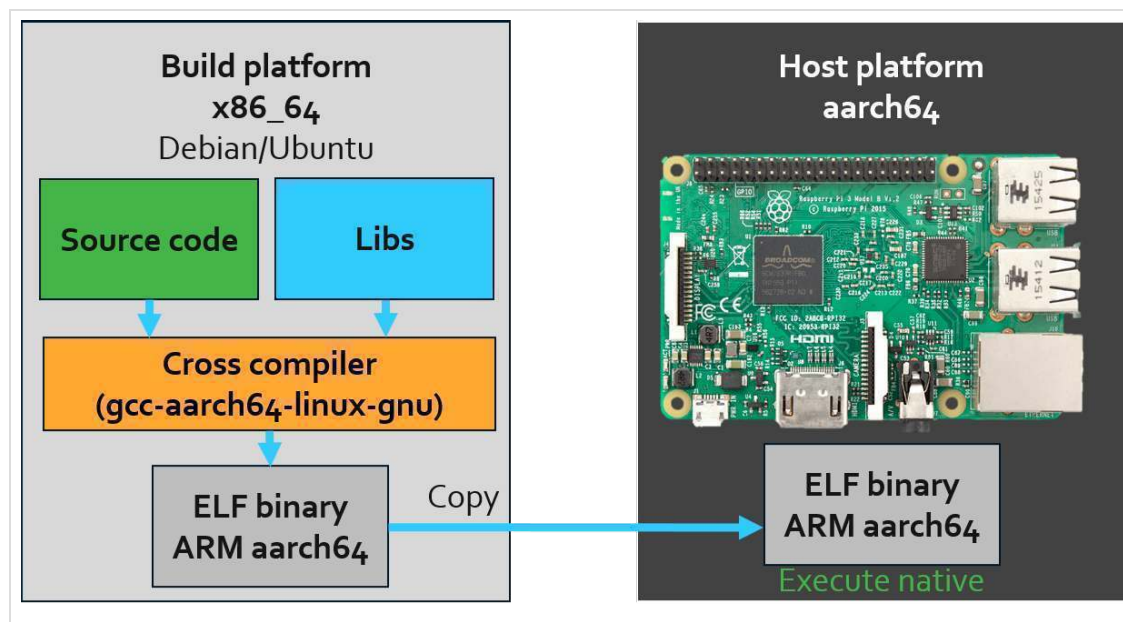
— Cross compiling

When you are building for the same architecture as which you are using, build, host and target are the same. This is called a "native" compilation. If build and target platform are the same, but host is different, then we're talking about cross compilation, which this post is covering. When all three platforms are different, it's called a "canadian". This is used to build a cross compiler for another architecture.

Just to be clear, in this post, the build and target platform are x86_64 (standard PC) and the host is the ARM platform. I will cover both 32bit ARM (armv6, armv7 or simply arm) and 64bit ARM (aarch64).

ARM architectures

To find out for which of these (32 bit or 64 bit ARM) you need to compile, the easiest is to look at the output of uname -m.

For x86_64 (standard PC):

```
jensd@deb10:~$ uname -m
x86_64
```

32 bit ARM:

```
[ap1:~]# uname -m
armv7l
```

64 bit ARM (or aarch64):

```
root@armv8:/ # uname -m
aarch64
```

Prerequisites

Before we can start compiling, we need to install the necessary packages and tools for cross compiling

for ARM. These include the standard tools needed for compiling native:

For 32 bit ARM (arm):

```
jensd@deb10:~$ sudo apt install gcc make gcc-arm-linux-gnueabi binutils-
arm-linux-gnueabi
Reading package lists... Done
Building dependency tree
...
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for libc-bin (2.28-10) ...
```

For 64 bit ARM (aarch64):

```
jensd@deb10:~$ sudo apt install gcc make gcc-aarch64-linux-gnu binutils-
aarch64-linux-gnu
Reading package lists... Done
...
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for libc-bin (2.28-10) ...
```

Of course you can install both the necessary compilers for 32 and 64-bit if you plan to compile for both these architectures.

Compiling a simple C program from source

Once we have installed the prerequisites, we can try and compile a simple C program. Let's first do a so-called native compile for the PC we're compiling from, just to make sure that our program does what we want.

Save the source as helloworld.c:

```
#include<stdio.h>
int main()
{
        printf("Hello World!\n");
        return 0;
}
```

Compile the source "native" and write the binary as helloworld-x86_64

```
jensd@deb10:~$ vi helloworld.c
jensd@deb10:~$ gcc helloworld.c -o helloworld-x86_64
```

To see what type and for which platform the result of our compilation is, we can check the output with the "file"-tool:

```
jensd@deb10:~$ file helloworld.x86_64
helloworld.x86_64: ELF 64-bit LSB pie executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=f55972265cd343c3110a5d339d71caeed755c23e,
not stripped
```

We can execute the binary to check the result:

```
jensd@deb10:~$ ./helloworld-x86_64
Hello World!
jensd@deb10:~$
```

The next step is to compile the same source for ARM. We simply do this by using a different compiler (arm-linux-gnueabi-gcc instead of gcc for 32 bit ARM or gcc-aarch64-linux-gnu for 64 bit ARM or aarch64).

For 32 bit ARM (arm):

```
jensd@deb10:~$ arm-linux-gnueabi-gcc helloworld.c -o helloworld-arm
-static
jensd@deb10:~$ file helloworld-arm
helloworld-arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=971f8b03dcb473de2caa1abbb63990977c6478b3, not stripped
```

For 64 bit ARM (aarch64):

```
jensd@deb10:~$ aarch64-linux-gnu-gcc helloworld.c -o helloworld-aarch64
-static
jensd@deb10:~$ file helloworld-aarch64
helloworld-aarch64: ELF 64-bit LSB executable, ARM aarch64, version 1
(GNU/Linux), statically linked, for GNU/Linux 3.7.0,
BuildID[sha1]=428239fafd78d5fd28dc400e913618817224ea50, not stripped
```

As you can see, file gives us a different result, which we would expect.
Trying to execute these binaries on the build machine (x86_64), as expected, will result in an error:

```
jensd@deb10:~$ ./helloworld-aarch64
-bash: ./helloworld-aarch64: cannot execute binary file: Exec format
error
jensd@deb10:~$ ./helloworld-arm
-bash: ./helloworld-arm: cannot execute binary file: Exec format error
```

To test if this has worked, we need a machine or device running the architecture for which we built:

```
root@armv8:/$ uname -a
Linux armv8 4.4.214-armada-17.10.1 #1 SMP Fri Jul 31 23:23:54 UTC 2020
aarch64 aarch64 aarch64 GNU/Linux
root@armv8:/$ uname -m
aarch64
root@armv8:/$ wget http://192.168.1.1/helloworld-aarch64
...
helloworld-aarch64 100% |*****************************| 557k 0:00:00 ETA
...
root@armv8:/ # chmod +x helloworld-aarch64
root@armv8:/ # ./helloworld-aarch64
Hello World!
```

As you see in the above output, our small program works fine on ARM after cross compiling it!

Cross compiling with configure and make

The above example was pretty simple but when compiling source from larger projects, it's usually done by generating a makefile with configure and then running the compile and other necessary steps with make. To replace gcc with another, target platform specific compiler would be a lot of work. Fortunately most of the times you can just specify the platform which you are compiling for when running configure.

As an example, I'll create a binary for ARM aarch64 of strace. To avoid getting into problems with dependencies on my embedded ARM device, I'll provide the static option (see below for more explanation).

First step is to get the source of strace from: https://github.com/strace/strace/releases/tag/v5.10 and extract it:

```
jensd@deb10:~$ wget https://github.com/strace/strace/releases/download
/v5.10/strace-5.10.tar.xz
...
strace-5.10.tar.xz    100%[===============>]   1.77M  2.49MB/s    in 0.7s
2021-01-26 16:57:20 (2.49 MB/s) - 'strace-5.10.tar.xz' saved
[1859688/1859688]
jensd@deb10:~$ tar -xf strace-5.10.tar.xz
jensd@deb10:~$ cd strace-5.10/
jensd@deb10:~/strace-5.10$
```

The next step is to run configure. But here we need to specify the build and host platform so that we want to end up with a binary (statically linked) for ARM:

```
jensd@deb10:~/strace-5.10$ ./configure --build x86_64-pc-linux-gnu --host
aarch64-linux-gnu LDFLAGS="-static -pthread" --enable-mpers=check
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane… yes
checking for aarch64-linux-gnu-strip... aarch64-linux-gnu-strip
...
config.status: creating strace.spec
config.status: creating debian/changelog
config.status: creating config.h
config.status: executing depfiles commands
jensd@deb10:~/strace-5.10$
```

At this point we're ready to do the actual cross compile by running make:

```
jensd@deb10:~/strace-5.10$ make
```

```
aarch64-linux-gnu-gcc -E -P -DHAVE_CONFIG_H   \
...
jensd@deb10:~/strace-5.10$ file strace
strace: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.7.0,
BuildID[sha1]=db1ce4305df1dac73b81efe99847725d65ac9ab4, with debug_info,
not stripped
```

As you see, I built this one for use on aarch64. If you want to do the same for armv6 or armv7, simply replace –host aarch64-linux-gnu with –host arm-linux-gnueabi when running configure.

Some additional explanation with the flags and arguments I passed to ./configure in order to get this working:

- `checking for library containing timer_create… no`
  `configure: error: failed to find timer_create`
  was fixed by adding LDFLAGS="-pthread"
- `checking for m32 personality compile support… no`
  `checking whether to enable m32 personality support… no`
  `configure: error: Cannot enable m32 personality support`
  was fixed by adding –enable-mpers=check

As with the small C-program, it's time to test the compiled binary on ARM:

```
root@armv8:/data $ uname -a
Linux armv8 4.4.214-armada-17.10.1 #1 SMP Fri Jul 31 23:23:54 UTC 2020
aarch64 aarch64 aarch64 GNU/Linux
root@armv8:/data $ uname -m
aarch64
root@armv8:/data $ wget http://192.168.1.1/strace
...
strace                  100%  |***********************| 5989k  0:00:00 ETA
...
root@armv8:/data $ chmod +x strace
root@armv8:/data $ ./strace -V
strace -- version 5.10
Copyright (c) 1991-2020 The strace developers https://strace.io.
This is free software; see the source for copying conditions.  There is
NO
 warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.

Optional features enabled: no-m32-mpers
```

This gives us the ability to simply copy and use strace on a random aarch64 machine.

About static linking and dependencies

As I mentioned in the beginning, I mainly use cross compilation to build troubleshooting tools. Often, the platform where you are building for, is limited. This could be due to lack of resources, like an embedded device. But also because pre-built packages are either not available or it's not possible to install them. In a lot of cases, this also means that installing dependencies for whatever you are building might be a problem. Obviously, these dependencies also have to be built for that same architecture.

If you have this kind of limitation or you simply want your binary to just run on that architecture. Without worrying on dependencies, or conflicting (older) version of those dependencies that might be already installed, you can use static linking. This means that at build time all necessary dependencies will be included in the binary itself.

Static linking has a few drawback as it is potentially unsecure (the included dependencies will not be updated with the system), could cause incompatibility when libraries that do lower level system calls and the resulting binary file will be larger. These are things I can live with when building troubleshooting tools as they are not intended for long-time use.
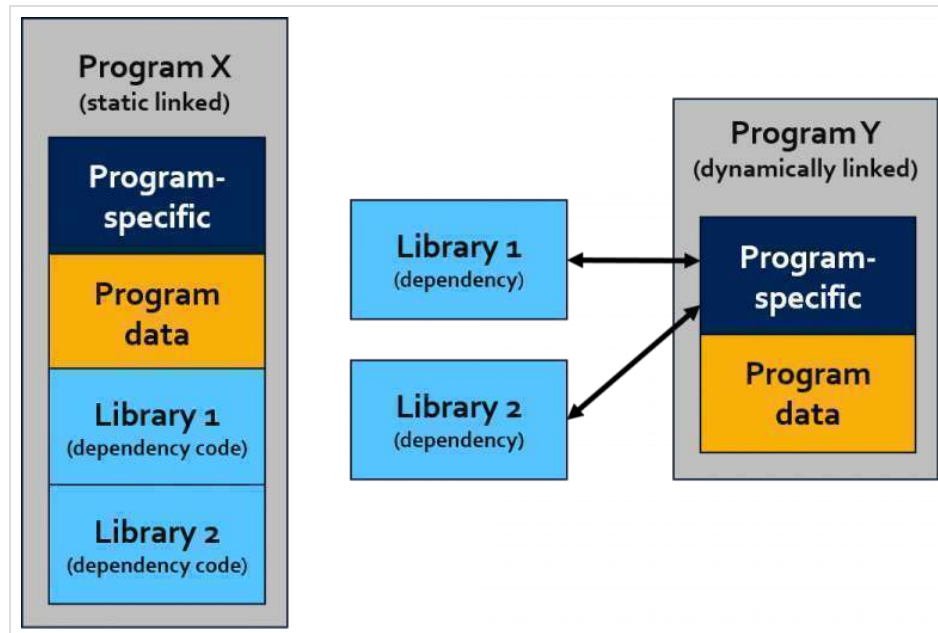
—   Static vs. dynamic linking

While static linking might be what you are looking for, it's not always be easy to accomplish. Especially in combination with cross compilation it can give you a headache. Most tools depend on libc or glibc, which discourages static linking for the good reasons I mentioned in the paragraph above. Fortunately there is a libc implementation that was developed from scratch and that allows proper static linking for libc-dependencies: musl (pronounce as musscle).

Cross compiling with configure and make using the musl libc implementation

To make use of musl, we need to download the correct version for our cross compilation. You can find a full list over here: https://musl.cc/#binaries.

After the download, we can extract the archive and test if this works on our build machine:

```
jensd@deb10:~$ wget https://musl.cc/aarch64-linux-musl-cross.tgz
...
aarch64-linux-musl-cross.tgz   100%[=============>]   103.69M  6.81MB/s
in 16s
2021-01-27 15:19:55 (6.42 MB/s) - 'aarch64-linux-musl-cross.tgz' saved
[108731156/108731156]
jensd@deb10:~$ tar -xvzf aarch64-linux-musl-cross.tgz
aarch64-linux-musl-cross/
aarch64-linux-musl-cross/usr
```

```
...
jensd@deb10:~$ ./aarch64-linux-musl-cross/bin/aarch64-linux-musl-gcc
--version
aarch64-linux-musl-gcc (GCC) 10.2.1 20210116
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Next, I will build a static linked version of TCPdump for aarch64. Always a nice tool to have handy when you need to log, investigate or troubleshoot network connectivity. Unfortunately tcpdump is not always available or in the best case as a limited Busybox-version.

First, we need to install some required tools, these are used by the tcpdump for the build process:

```
jensd@deb10:~$ sudo apt install flex bison
Reading package lists... Done
Building dependency tree
...
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for libc-bin (2.28-10) ...
```

Next, we need to download the source for both tcpdump and libpcap and extract it. You can find the latetst version over here: https://www.tcpdump.org/index.html#latest-releases

```
jensd@deb10:~$ wget https://www.tcpdump.org/release/libpcap-1.10.0.tar.gz
...
libpcap-1.10.0.tar.gz   100%[============>]   912.66K  1.51MB/s    in
0.6s
2021-01-27 15:20:22 (1.51 MB/s) - 'libpcap-1.10.0.tar.gz' saved
[934559/934559]
jensd@deb10:~$ tar -xvzf libpcap-1.10.0.tar.gz
jensd@deb10:~$ wget https://www.tcpdump.org/release/tcpdump-4.99.0.tar.gz
...
tcpdump-4.99.0.tar.gz   100%[============>]   1.92M  2.57MB/s    in 0.7s
2021-01-27 15:21:36 (2.57 MB/s) - 'tcpdump-4.99.0.tar.gz' saved
[2008080/2008080]
jensd@deb10:~$ tar -xvzf tcpdump-4.99.0.tar.gz
```

After downloading and extracting the source code, we need to run the configure-script for libpcap first. Only this time, we need to set our compiler to the musl-compiler for our cross compilation by setting CC to: aarch64-linux-musl-gcc:

```
jensd@deb10:~$ cd libpcap-1.10.0/
jensd@deb10:~/libpcap-1.10.0$ CC=/home/jensd/aarch64-linux-musl-cross/bin
/aarch64-linux-musl-gcc ./configure --build x86_64-pc-linux-gnu --host
aarch64-linux-gnu LDFLAGS="-static"
checking build system type... x86_64-pc-linux-gnu
checking host system type... aarch64-unknown-linux-gnu
...
config.status: creating config.h
config.status: executing default-1 commands
```

If all went fine, we can do the actual compilation of libpcap by issuing make:

```
jensd@deb10:~/libpcap-1.10.0$ make
...
config.status: creating pcap-config.tmp
mv pcap-config.tmp pcap-config
chmod a+x pcap-config
```

Now, we can repeat the same (./configure and make) for tcpdump itself. By doing the static linking, libpcap will be included and the result is a single binary tcpdump:

```
jensd@deb10:~/libpcap-1.10.0$ cd ../tcpdump-4.99.0/
jensd@deb10:~/tcpdump-4.99.0$ CC=/home/jensd/aarch64-linux-musl-cross/bin
/aarch64-linux-musl-gcc ./configure --build x86_64-pc-linux-gnu --host
aarch64-linux-gnu LDFLAGS="-static"
checking build system type... x86_64-pc-linux-gnu
checking host system type... aarch64-unknown-linux-gnu
checking for aarch64-linux-gnu-gcc... /home/jensd/aarch64-linux-musl-
cross/bin/aarch64-linux-musl-gcc
checking whether the C compiler works... yes
...
config.status: creating Makefile
config.status: creating tcpdump.1
config.status: creating config.h
config.status: executing default-1 commands
jensd@deb10:~/tcpdump-4.99.0$ make
...
aarch64-linux-gnu-ranlib libnetdissect.a
 /home/jensd/aarch64-linux-musl-cross/bin/aarch64-linux-musl-gcc
-DHAVE_CONFIG_H   -I. -I../libpcap-1.10.0  -I/usr/inet6/include -g -O2
-static -o tcpdump fptype.o tcpdump.o  libnetdissect.a ../libpcap-1.10.0
/libpcap.a
jensd@deb10:~/tcpdump-4.99.0$ file tcpdump
tcpdump: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
statically linked, with debug_info, not stripped
```

As you can see by the last command, we have our statically linked tcpdump binary. If all goes well, we should be able to execute this, without any further dependencies, on an aarch64-based machine.

```
root@armv8:/$ uname -m
aarch64
root@armv8:/$ wget http://192.168.1.1/tcpdump
...
tcpdump    100%[=============>]   6.73M  --.-KB/s    in 0.1s
2021-01-27 13:55:49 (52.5 MB/s) - 'tcpdump' saved [7061368/7061368]
root@armv8:/$ chmod +x tcpdump
root@armv8:/$ ./tcpdump -i eth0
tcpdump: verbose output suppressed, use -v[v]… for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
13:57:20.452643 IP 192.168.0.90.34020 > 192.168.1.10.9000: Flags [S], seq
3386726277, win 64240, options [mss 1460,sackOK,TS val 3092573417 ecr
0,nop,wscale 7], length 0
13:57:20.452715 IP 192.168.1.10.9000 > 192.168.0.90.34020: Flags [S.],
seq 3761157769, ack 3386726278, win 28960, options [mss 1460,sackOK,TS
```

```
val 4397992 ecr 3092573417,nop,wscale 7], length 0
13:57:20.453325 IP 192.168.0.90.34020 > 192.168.1.10.9000: Flags [.], ack
1, win 502, options [nop,nop,TS val 3092573418 ecr 4397992], length 0
13:57:20.508762 IP 192.168.1.10.46819 > 8.8.8.8.53: 15310+ PTR?
90.0.168.192.in-addr.arpa. (43)
 ^C
4 packets captured
23 packets received by filter
5 packets dropped by kernel
root@armv8:/$
```

As you can see, tcpdump works fine and does not need any dependencies. This is really helpful if you need to work or troubleshoot on platforms that come with a very limited set of tools and no ability to easily install them. You can simply copy the file and it will work.

This entry was posted in **ARM**, **Debian**, **Development**, **Linux**, **Ubuntu** by **jensd**. Bookmark the **permalink [https://jensd.be/1126/linux/cross-compiling-for-arm-or-aarch64-on-debian-or-ubuntu]** .

12 THOUGHTS ON "CROSS COMPILING FOR ARM OR AARCH64 ON DEBIAN OR UBUNTU"

Allen
on **28/01/2021 at 20:52** said:

Very helpful. You've provided a clear explanations of how to do it, and why each of the steps is needed. It's a good tutorial that can be used to extend the concepts to more complicated cross compiling.

jensd
on **28/01/2021 at 21:51** said:

Thanks you! Nice to get some positive feedback :)
Also working on a small YouTube video covering the same.

Pingback: Cross compiling for ARM with Ubuntu 16.04 LTS | Jensd's I/O buffer

MTDzi
on **07/03/2021 at 14:21** said:

Awesome content, thanks! :)

I have a question, slightly related, but more about shared libraries: suppose I'm able to cross compile a shared library with a bunch of dependencies (100+), and I'd need to deploy it to aarch64, and it needs to be a shared library (because of how many dependencies there are; otherwise the binary would be huge). Since I've never done this, my question is pretty basic: what are my next steps? I can probably makes sure that all those dependencies are of the correct version but they'll (in general) have a different path.

Is there a tutorial and/or tool that you could recommend for automatically linking my library on the aarch64 device?

And thanks for your time!

> **jensd**
> on **08/03/2021 at 11:28** said:
>
> The aarch64 machine will need those dependencies. Either you could install them through the package manager (if there is any over there) or cross compile these separately. The versions do not need to be an exact match but too far away will probably give you troubles.
>
> About the path they are in, you do not need to worry. At the time of execution, those dependencies/libraries are being searched in LD_LIBRARY_PATH. You check in advance which libs are missing with the ldd command as well.

zeeshan
on **19/04/2021 at 12:20** said:

hi, Can you help me to cross compile using cmakelist.txt!! i want to compile for ARM on X86

Mickey Mouse
on **13/01/2022 at 16:45** said:

Great article, easy to grasp what is going on.

After reading it, download buildroot on a Linux PC, extract and use it,
for example for RPI 4, buildroot will take care for cross-compiler,
locating all .a, .so, .h … files.

You can use wsl (Windows 10 for linux) subsystem of Microsoft
https://buildroot.org/docs.html

Success!

sara
on **26/01/2022 at 08:10** said:

how did it get in to arm8 directory ?

jensd
on **26/01/2022 at 21:00** said:

Not sure what you mean, which part are you referring to?

Pingback: 一些计算两台主机之间进行socket通信的延迟的小程序 - 算法网

R. Sundararaman
on **17/02/2022 at 08:36** said:

Hi,
Thanks. Can you guide how to cross compile c++ files in build x86_64 platform for host arm64 platform in Centos 8? What glibc package is required for this task?

Goutham
on **24/02/2022 at 09:40** said:

Hey,
Can you please help me to install cross compiler with g++ 10 version in ubuntu for arm64 target?