Cross Platform Development for Raspberry

Cross-development means developing and compiling programs on another platform then it intended to run upon. It is a common approach for Pi, since it’s also how Raspian is built. For quite some time I did compile Pi applications on the Pi itself. Which works quite well, but has one large drawback: performance. Being able to compile these programs on my 8 core 16GB desktop, improved compilation time enorms. And it is quite simple. I have used 2 different approaches to setup a cross build environment:

1. Manual setup
2. Debian Crossbuild packages

Let's have a look!

GCC and Linux Architectures

GCC is also used to cross compile Linux applications. Applications can be compiled for 32-bit or 64-bit Linux systems.

- For 32-bit:
  - arm-linux-gnueabihf-gcc and
  - arm-linux-gnueabihf-g++.
- For 64-bit:
  - aarch64-linux-gnu-gcc and
  - aarch64-linux-gnu-g++.

Check the current target architecture by logging into the Pi:

```
# gcc -dumpmachine
------------------
arm-linux-gnueabihf
```

Method 1: Debian Crossbuild packages

Required Development sources

That will involve setting up dpkg for armhf architecture:

```
dpkg --add-architecture armhf
apt-get update
```

Installing the development libraries for new arhitecture:

```
apt-get install libssl-dev:armhf
apt-get install gnutls-dev:armhf
```

```
apt-get install libmicrohttpd-dev:armhf
apt-get install libgpiod2:armhf
```

Method 2: Manual Setup

Set Up Cross Build Tools

The first step is to install the development tools on the desktop, or host system. From the command line run the following:

```
# apt-get update
# apt-get upgrade
# apt-get install build-essential
# apt-get install gcc-arm-linux-gnueabihf
# apt-get install g++-arm-linux-gnueabihf
# apt-get install gcc-aarch64-linux-gnu
# apt-get install g++-aarch64-linux-gnu
```

The first line makes sure that the system is up to date. The second instruction installs the general build tools. The third installs the C and C++ compiler and build tools for the Pi's ARM processor. The ARM architecture designation, arm-linux-gnueabihf-, is used as a prefix to distinguish the ARM tools from the host system tools. Note the dash at the end of the string. Test the installation by entering the commant below. This reports the version of the G++ compiler installed and other information:

```
arm-linux-gnueabihf-g++ -v
```

Get required ARM libraries

Depending on the application and required libraries, it is possible that not all libraries are available. Easiest way is to copy these over from an existing Raspberry installation. On the Raspberry these are available in: ***/usr/lib/arm-linux-gnueabihf***. On the target build environment the library path is: ***/usr/arm-linux-gnueabihf/lib/***. We will be placing these retrieved libraries in a separate subdirectory (extra).

```
mkdir /usr/arm-linux-gnueabihf/lib/extra
scp root@192.168.178.xx:/usr/lib/arm-linux-gnueabihf/libxxxx.* /usr/arm-
linux-gnueabihf/lib/extra
```

Now we need to tell the loader (ld) to look for libraries in this extra directory

```
nano /etc/ld.so.conf.d/arm-linux-gnueabihf.conf
Insert the path "/usr/arm-linux-gnueabihf/lib/extra" into this file. Then
save and run:
ldconfig
```

Test Application

The host system should now be ready to build a Raspberry Pi program. Let’s test it with a minimal test application, just to see if everything is working. Create the following test.cpp file:

```
#include <iostream>
using namespace std;

int main()
{
  cout << "This is our first test!" << endl;
  return 0;
}
```

Build the program. The first line compiles the file test.cpp and the second links the compiler output to build the executable.

```
arm-linux-gnueabihf-g++ -O3 -g3 -Wall -c -fPIC -o test.o test.cpp
arm-linux-gnueabihf-g++ -o test test.o
```

Now let's check the architecture of the executable with the 'file' command. If this shows 'ARM'it is compiled for an ARM processor and can transfer it to the Pi.

```
file test
test: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
linked,
interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0,
BuildID[sha1]=3a73dacec4a89fa2c6ea9751d18743ba9bba33ac, not stripped

scp test pi@192.168.178.xx:/tmp
```

That's all. I told you this was simple.

Remote Debugging

We will gdb on host and Pi to debug with the GCC toolset. First install the debug capability on our desktop system that works with target systems regardless of the processor architecture.

```
sudo apt-get install gdb-multiarch
```

Now install a debug server on the Pi system that can be connected to from our desktop and control execution on the target.

```
sudo apt-get install gdbserver
```

Start the server on the Pi. The command starts the server, runs it continuously. The third parameter is the computer name or IP address of the host computer. In my case it is “192.168.178.18”. 2001 is the port that the host and target will use for communication. Be aware that running the gdbserver is a

potential security hole on the Pi. So make sure it can only be accessed from your local network.

```
gdbserver --multi 192.168.178.18:2001
```

The server uses stdout to display error messages and the output from the debugged program.

The program gdb-multiarch is an interactive program. On the host, enter the command:

```
gdb-multiarch
```

This prints a lot of output followed by the prompt (gdb). You might want to add the quiet option (-q) to the command line. At the prompt, using the name of your Pi, enter the following commands and you should see the responses, as shown in italics here:

```
target extended-remote 192.168.178.61:2001
Remote debugging using 192.168.178.61:2001
set remote exec-file remote/test
[[no response]]
file test
Reading symbols from test...done.
```

The target command tells gdb the name of the target system, 192.168.178.61 with port, 2001. If it worked, you'll get the response shown; otherwise a timeout message. The set remote line specifies the file location to use on the Pi target system. Remember that the executable needs to be copied onto the target system every time it is changed. We'll see how to do that from within gdb, below. The last command, file, is the path and filename on the host computer. This can be a directory other than where the debugger is running. Now, type the following and get the response as shown:

```
run
Starting program: /tmp/hello
[[ a bunch more lines, some warnings and stuff to ignore for now ]]
[Inferior 1 (process 234) exited normally]
```

If it works you'll see that last line. Now switch to the SSH terminal connected to the Pi and you'll see a couple of lines of status, the output from the program, a blank line, and a line reporting the exit status of the program. If all this doesn't work check the error message from the server. It should explain what is happening. Often there is a mismatch between what you told gdb and the location of the executable.

The debugger provides a large number of commands for manipulating breakpoints, continuing execution, listing the program, etc. For instance, type in main and the program stops when main() is reached. A 'c' runs the program after a breakpoint. To help with debugging, there is an interesting mode, the Terminal User Interface, or tui. You can access and leave it by typing in Ctrl-x, Ctrl-a. This mode provides windows that show the source, registers, assembly language, and other information.

NEW

Links

- https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads
- https://www.acmesystems.it/arm9_toolchain
- https://jensd.be/1126/linux/cross-compiling-for-arm-or-aarch64-on-debian-or-ubuntu
- https://learn.arm.com/install-guides/gcc/cross/
- https://packages.debian.org/search?keywords=crossbuild-essential
- https://www.get-edi.io/assets/pdfs/DebianCross.pdf
- https://prashanth.entertolearn.in/debian-arm-architecture-and-emulation/using-debian-arm-cross-compiler-for-bare-metal-programming