

Linux Training:Introduction to Cross Compiling

Contents

Prerequisite

- SDK
- Lab Files
- Create Directory

Lab 1

Lab 2

Lab 3

- Lab 3a
- Lab 3b
- Lab 3c

Prerequisite

This training assumes you already have Processor Linux SDK 1.0 installed. The instructions and screen shots will be based on the AM335x SDK. The instructions should generally be the same AM335x and AM437x SDK. Although path's may slightly change.

Since everyone's computer is different and the sdk may be installed anywhere the exact commands that need to be used may change depending on each user's setup.

To keep things generic, folder path's placeholders will be used. This makes it clear that the path required is dependent on the user's setup. When you see any place holder make sure you replace it with the proper path based on your local setup.

SDK

The full path to the Processor Linux SDK 1.0 will be referred to as:

<sdk-path>

Lab Files

This lab also requires lab files and is located within a git repository. Choose the location where you want these files to be located and run the below command:

```
git clone git://git.ti.com/sitara-linux/toolchain-lab.git
```

```
cd toolchain-lab
```

```
git checkout processor-linux-sdk-v1.0
```

The full path to the toolchain-lab folder will be referred to as:

<toolchain-lab dir path>

Create Directory

This lab also requires a directory called cross-compiled to be created. Create a directory called cross-compiled.

The full path to this directory will be referred to as:

<cross-compiled dir path>

Lab 1

This first lab will show you how to use the cross compile tool chain to build a simple hello-world program.

The first step to using the tool chain is to include its full path in your environments PATH variable.

This will allow you to call the tool chain's compiler by simply entering arm-linux-gnueabi-hf-gcc instead of "<sdk path>/linux-devkit/sysroots/i686-arago-linux/usr/bin/arm-linux-gnueabi-hf-gcc" every time you want to use the compiler.

The tool chain is located in your sdk's linux-devkit/sysroots/i686-arago-linux/usr/bin directory.

The PATH environment variable determines what programs are available to a user without having to always specify the full path to each program/. Imagine if you had to constantly type /usr/bin/cd or /usr/bin/ls!

Enter the below command to add the tool chain’s path to the PATH environment variable.

```
export PATH=<sdk-path>/linux-devkit/sysroots/i686-arago-linux/usr/bin:$PATH
```

To verify that your tool chain is now in your “path” enter the below characters and then hit tab:

```
arm-  
  
You should see a bunch of programs that start with arm-linux-gnueabi- like below.  
arm-linux-gnueabi-addr2line  arm-linux-gnueabi-gprof  
arm-linux-gnueabi-ar         arm-linux-gnueabi-ld  
arm-linux-gnueabi-as         arm-linux-gnueabi-ld.bfd  
arm-linux-gnueabi-c++filt    arm-linux-gnueabi-ldd  
arm-linux-gnueabi-cpp        arm-linux-gnueabi-ld.gold  
arm-linux-gnueabi-g++        arm-linux-gnueabi-nm  
arm-linux-gnueabi-gcc        arm-linux-gnueabi-objcopy  
arm-linux-gnueabi-gcc-4.7.3  arm-linux-gnueabi-objdump  
arm-linux-gnueabi-gcc-ar     arm-linux-gnueabi-ranlib
```

If you don’t see something similar to the above then your PATH variable was not set properly and you need to verify that the path to the toolchain was set correctly.

Now that your tool chain is in your path you can build the hello-world example.

The command you will use follows the below pattern:

```
arm-linux-gnueabi-gcc <list of source files for a program> -o <name of executable>
```

For this hello-world program run the below command:

```
arm-linux-gnueabi-gcc hello-world.c -o hello-world
```

You should not see anything outputted but if you type ls you should see a program called hello-world inside your current directory.

Lab 2

Now that you know how to build a simple program using the cross compiler it is time to build something a bit more complex.

In the terminal, execute the below command to build an example application based on the libpng library.

```
arm-linux-gnueabi-gcc libpng-short-example.c -o example-png
```

```
Oops this time you should see an error similar to the below image.  
libpng-short-example.c:(.text+0x594): undefined reference to `png_write_end'  
/tmp/ccsAYpqR.o: In function `process_file':  
libpng-short-example.c:(.text+0x626): undefined reference to `png_set_expand'  
libpng-short-example.c:(.text+0x636): undefined reference to `png_set_strip_16'  
collect2: error: ld returned 1 exit status
```

A quick Google search tells us this kind of error is related to libpng.

Usually if you don’t get a complaint about a missing header but instead see an undefined reference error message either something is wrong with the program or the compiler’s linker isn’t able to find the necessary library or doesn’t know what library to link against.

Since the example we are trying to build comes from libpng sources we can give it the benefit of the doubt that that the program has no syntax errors.

If you do a quick search in “<sdk-path>/linux-devkit/sysroots/cortexa8t2hf-vfp-neon-linux-gnueabi/usr/lib” you will find libpng headers and libraries (libpng.so) which means the required files to build this program are available and the linker isn’t linking to it for some reason. You can double check this by looking at the sdk’s manifest located in the sdk’s doc directory.

libpng12	1.2.50-r0
libpng16-16	1.6.8-r0

As you can see in the above image the sdk does include the libpng library so we can be sure the files are there.

By default the compiler knows where to search for all the headers and libraries contained in linux-devkit so the problem must be related to the linker not knowing which library to link against.

The way you do this is by passing the library to link to directly to the compiler.

If you search for libpng in linux-devkit you will see the library file libpng.so.

So how do you tell the linker to link against libpng.so? The process is pretty simple. For the library name drop “lib” from the beginning of the library name and drop the extension from the end of the library name. And then you finally append ‘-l’ to the front and pass that directly to the compiler.

For libpng.so this will result in -lpng being passed to the compiler.

So run the below command:

```
arm-linux-gnueabi-gcc libpng-short-example.c -lpng -o example-png
```

If you received no error then run ls to verify that “example-png” now exist. If so you successfully built a program that linked against an external library.

Note: Usually when building programs you didn’t create you don’t need to manually specify which library the linker should link against since the Makefile will generally handle this. Usually if you have a linking error it is due to the linker not finding the library it was told to link against.

Lab 3

Lab 3a

In this this portion of the lab you will build another example application that depends on the libmad library.

Building libmad should be similar to building example-png so let’s give a shot.

Run the below command in the terminal: arm-linux-gnueabi-gcc minimad.c -o mad-example

```
> arm-linux-gnueabi-gcc minimad.c -o mad-example
minimad.c:27:18: fatal error: mad.h: No such file or directory
compilation terminated.
```

The above message is something that we have never seen before. This message is saying that the compiler is unable to find the header mad.h.

Using a bit of Google magic you will find out that mad.h is a part of the libmad library. So you can either check the sdk’s manifest (docs/software_manifest.htm) or search in “<sdk-path>/linux-devkit/sysroots/cortexa8t2hf-vfp-neon-linux-gnueabi/usr/include” to find references to libmad or mad.h. In both cases no header or references to libmad could be found. What will we do?

Well with our new skills with the toolchain how hard can it be to manually cross compile the library from scratch?

Run the below command to go to the libmad sources directory that was conveniently downloaded for you:

```
cd “<toolchain-lab dir path>/libmad-0.15.1b”
```

Libmad uses auto tools and the first step to building an auto tools based program is to run configure.

Run configure by entering the below command:

```
./configure
```

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
```

Running ./configure may succeed but if you look closely to the output you may notice that it isn’t picking up the tool chain provided in the sdk which is a problem since it means your configuring and compiling the software for your host PC not for your target device.

From the screenshot above you will see the line asking if we are cross compiling which should say yes but since it says no that means we have a big problem.

To fix this we need the help of environment-setup and all the useful environment variables it exports.

When cross compiling user space programs you should always source environment-setup. This file includes a lot of helpful environment variables and best of all it already adds the compiler to your PATH!

Run the below command to source environment-setup:

```
source <sdk-path>/linux-devkit/environment-setup
```

In your console you should see “[linux-devkit]” in lime green colors. This is an indicator that environment-setup has been sourced in your current terminal window.

[linux-devkit]:

Now that environment-setup has been sourced run configure again:

```
./configure
```

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for gcc... arm-linux-gnueabi-gcc
checking for C compiler default output... a.out
checking whether the C compiler works... configure: error: cannot run C compiled programs.
If you meant to cross compile, use `--host'.
See `config.log' for more details.
```

You will now see that it is detecting the correct compiler but it still terminates with an error. This is because when cross compiling configure needs some additional parameters passed to it.

So how do you know what parameters are required and what the value should be? Who cares? Environment-setup already has an environment variable called `CONFIGURE_FLAGS` that provides this information!

Run the below command and notice that `CONFIGURE_FLAGS` is also being passed in.

```
./configure $CONFIGURE_FLAGS
```

```
checking whether to enable experimental code... no
configure: creating ./config.status
config.status: creating Makefile
config.status: creating msvc++/Makefile
config.status: creating libmad.list
config.status: creating config.h
config.status: executing depfiles commands
```

If you see something similar to the above that means configure was able to execute correctly.

Now you simply need to run make to compile the library.

A handy option for make is the `-j` option. This option lets you pass a number to run make with multiple threads.

So to run make with 4 threads you will use `make -j 4`. Since modern computers typically have at least 2 cores and most likely 2 additional virtual cores which means `make -j 4` will speed things up significantly.

Bonus: Try using this option when building your kernel. You will see a significant difference in build speed.

Run make with four threads by executing the below commands:

```
make -j 4
```

```
(cd .libs && rm -f libmad.so.0 && ln -s libmad.so.0.2.1 libmad.so.0)
(cd .libs && rm -f libmad.so && ln -s libmad.so.0.2.1 libmad.so)
arm-linux-gnueabi-ar cru .libs/libmad.a version.o fixed.o bit.o timer.o stream.o frame.o synth.o decoder.o layer12.o layer3.o huffman.o imdct_l_arm.o
arm-linux-gnueabi-ranlib .libs/libmad.a
creating libmad.la
(cd .libs && rm -f libmad.la && ln -s ../libmad.la libmad.la)
```

If make ran successfully you should see something like the above.

Now let's go back to our "toolchain lab" directory and build our example again.

Run the below commands:

```
cd ..
```

```
cd
```

Hmm for some reason you're getting the same exact error message even though libmad has been cross compiled and the headers exist.

The problem is environment-setup only knows about the libraries and headers located in linux-devkit.

So you have two options

Install all the headers and libraries into linux-devkit but that can make things messy and will also make it difficult to determine what came with the sdk and what didn't.

Create a new directory that will contain all the files you manually cross compiled.

Lab 3b

To install headers and libraries you simply run `make install` and it will install all the necessary headers, libraries and misc. files needed by libmad.

So run the below command:

```
make install
```

```
make[3]: Entering directory `/home/sitara/toolchain-lab/libmad-0.15.1b'
mkdir -p -- . /usr/local/lib
/bin/sh ./libtool --mode=install /usr/bin/install -c libmad.la /usr/local/lib/libmad.la
/usr/bin/install -c .libs/libmad.so.0.2.1 /usr/local/lib/libmad.so.0.2.1
/usr/bin/install: cannot create regular file `/usr/local/lib/libmad.so.0.2.1': Permission denied
```

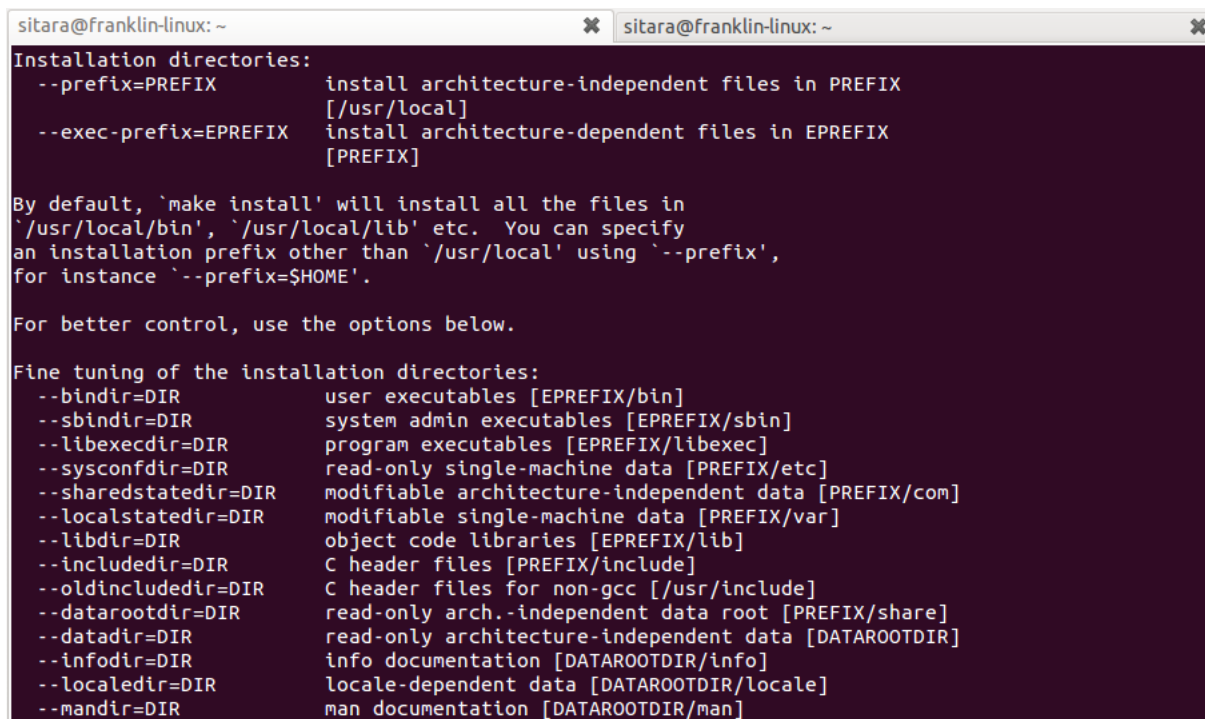
This is not a good sign. Running `make install` caused the makefile to try to install our cross compiled libraries on the host (`/usr/local/`) which is definitely something we don't want.

So how can you change the install location?

Configuration changes such as this is usually handled by `configure` so looking at the `configure` help will hopefully provide a solution

Run:

```
./configure --help
```



```
sitara@franklin-linux: ~
Installation directories:
  --prefix=PREFIX      install architecture-independent files in PREFIX
                        [/usr/local]
  --exec-prefix=EPREFIX install architecture-dependent files in EPREFIX
                        [PREFIX]

By default, 'make install' will install all the files in
'usr/local/bin', 'usr/local/lib' etc. You can specify
an installation prefix other than 'usr/local' using '--prefix',
for instance '--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR          user executables [EPREFIX/bin]
  --sbindir=DIR         system admin executables [EPREFIX/sbin]
  --libexecdir=DIR      program executables [EPREFIX/libexec]
  --sysconfdir=DIR      read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR  modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR   modifiable single-machine data [PREFIX/var]
  --libdir=DIR          object code libraries [EPREFIX/lib]
  --includedir=DIR      C header files [PREFIX/include]
  --oldincludedir=DIR   C header files for non-gcc [/usr/include]
  --datarootdir=DIR     read-only arch.-independent data root [PREFIX/share]
  --datadir=DIR         read-only architecture-independent data [DATAROOTDIR]
  --infodir=DIR         info documentation [DATAROOTDIR/info]
  --localedir=DIR       locale-dependent data [DATAROOTDIR/locale]
  --mandir=DIR         man documentation [DATAROOTDIR/man]
```

Awesome changing the location that libmad is installed to is as simple as using the `--prefix` option.

First we want to remove all the configuration and compiled files that were generated from the last time we built libmad.

To do this we use `make distclean`. For some software running `make distclean` isn't a valid command so you will use `make clean` instead.

So first run:

```
make distclean
```

We want the files to be located at `/home/sitara/cross-compiled/usr/` so rerun `configure` using the below command:

```
./configure $CONFIGURE_FLAGS --prefix=<cross-compiled dir path>/usr/
```

If you don't see any errors then run:

```
make -j 4
```

Finally run:

```
make install
```

```
mkdir -p -- . /home/sitara/cross-compiled/usr/include
/usr/bin/install -c -m 644 mad.h /home/sitara/cross-compiled/usr/include/mad.h
make[3]: Leaving directory `/home/sitara/toolchain-lab/libmad-0.15.1b'
make[2]: Leaving directory `/home/sitara/toolchain-lab/libmad-0.15.1b'
make[1]: Leaving directory `/home/sitara/toolchain-lab/libmad-0.15.1b'
```

If you see something similar to the above with no permission denied errors then make install was successful. Typically make install will create any necessary folders if they don't exist already so there is no need to manually create the "cross-compiled" directory

Lab 3c

Now that libmad has been cross compiled and installed in a separate directory the compiler still needs a way to know where to find this new location for headers and libraries.

This will require a change to environment-setup.

In the terminal run:

```
gedit <sdk-path>/linux-devkit/environment-setup
```

In environment setup find both the lines with LDFLAGS and CPPFLAGS.

LDFLAGS determines where to search for the target libraries while CPPFLAGS determines where to search for target headers.

To add to the library search path, the -L<directory> option should be added to the LDFLAGS environment variable. For the headers search path, the -I<directory> option should be added to the CPPFLAGS environment variable.

So in environment-setup change the lines:

```
export LDFLAGS="--sysroot=$SDK_PATH_TARGET"
```

```
...
```

```
export CPPFLAGS="-march=armv7-a -marm -mthumb-interwork -mfloat-abi=hard -mfpu=neon -mtune=cortex-a8 --sysroot=$SDK_PATH_TARGET"
```

To

```
export LDFLAGS="-L<cross-compiled dir path>/usr/lib/ --sysroot=$SDK_PATH_TARGET"
```

```
...
```

```
export CPPFLAGS="-I<cross-compiled dir path>/usr/include/ -march=armv7-a -marm -mthumb-interwork -mfloat-abi=hard -mfpu=neon -mtune=cortex-a8 --sysroot=$SDK_PATH_TARGET"
```

Note: -I is a dash with a capital i.

You should only add the bolded text so leave everything else as is.

Now that you have changed environment-setup before you can make use of this change you need to source environment-setup again.

So run:

```
source <sdk-path>/linux-devkit/environment-setup
```

Now in the terminal go back to the "toolchain lab" dir if you are no longer there by running the below command:

```
cd "<toolchain-lab dir path>"
```

Let's try building the program again by running the below command:

```
arm-linux-gnueabi-gcc minimad.c -o mad-example
```

```
/tmp/ccmEZnBM.o: In function `input':
minimad.c:(.text+0x12c): undefined reference to `mad_stream_buffer'
/tmp/ccmEZnBM.o: In function `error':
minimad.c:(.text+0x300): undefined reference to `mad_stream_errorstr'
/tmp/ccmEZnBM.o: In function `decode':
minimad.c:(.text+0x3b0): undefined reference to `mad_decoder_init'
minimad.c:(.text+0x3c0): undefined reference to `mad_decoder_run'
minimad.c:(.text+0x3d0): undefined reference to `mad_decoder_finish'
collect2: error: ld returned 1 exit status
```

Unfortunately the same error as before is seen again. The reason is we need to explicitly pass in the CPPFLAGS and LDFLAGS environment-variable to properly tell the compiler the

additional locations to search. A makefile typically handles this for you if it exist for the program or library your building.

So run the below command and also pass the CPPFLAGS and LDFLAGS environment variable:

```
arm-linux-gnueabi-gcc minimad.c $CPPFLAGS $LDFLAGS -o mad-example
```

Ahh progress but our old friend is back.

Once again the linker needs to have the library name passed to it.

The shared library for libmad is called libmad.so. So like before the library name needs to be converted which would be "-lmad"

So let's hopefully try building this one last time

```
arm-linux-gnueabi-gcc minimad.c $CPPFLAGS $LDFLAGS -lmad -o mad-example
```

No error this time!

If you run ls you will see mad-example has been created! Congrats!

Now unlike hello-world and example-png you can't simply copy mad-example to your sdk's target file system and expect it to work since your filesystem doesn't include libmad! So how can you copy the necessary files for libmad onto your file system?

Simply copy the contents of cross-compiled dir path directly to the root of your target file system (ex /media/rootfs when using a sd card reader). Being able to simply copy the manually compiled files to your target file system is another benefit of installing the files in a separate directory. Once the files are copied you can copy the "mad-example" program to your target file system and you will be good to go.

Keystone=

{{
1. switchcategory:MultiCore=

▪ For technical support on MultiCore devices, please post your questions in the [C6000 MultiCore Forum](#)

▪ For questions related to the BIOS MultiCore SDK (MCSDK), please use the [BIOS Forum](#)

Please post only comments related to the article [Linux Training:Introduction to Cross Compiling](#) here.

▪ For technical support on MultiCore devices, please post your questions in the [C6000 MultiCore Forum](#)

▪ For questions related to the BIOS MultiCore SDK (MCSDK), please use the [BIOS Forum](#)

Please post only comments related to the article [Linux Training:Introduction to Cross Compiling](#) here.


C2000=For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article [Linux Training:Introduction to Cross Compiling](#) here.

DaVinci=For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article [Linux Training:Introduction to Cross Compiling](#) here.

MSP430=For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article [Linux Training:Introduction to Cross Compiling](#) here.

OMAP35x=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article [Linux Training:Introduction to Cross Compiling](#) here.

OMAP=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article [Linux Training:Introduction to Cross Compiling](#) here.



Amplifiers & Linear

Audio

Broadband RF/IF & Digital Radio

Clocks & Timers

Data Converters

DLP & MEMS

High-Reliability Interface

Logic

Power Management

Processors

▪ [ARM Processors](#)

▪ [Digital Signal Processors \(DSP\)](#)

▪ [Microcontrollers \(MCU\)](#)

▪ [OMAP Applications Processors](#)

Switches & Multiplexers

Temperature Sensors & Control ICs

Wireless Connectivity

Retrieved from "https://processors.wiki.ti.com/index.php?title=Linux_Training:Introduction_to_Cross_Compiling&oldid=200679"

This page was last edited on 26 May 2015, at 10:52.
Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.