# NDK PORTING AND ANDROID MAKESYSTEM PORTING

By

Regards

Ashish Tanwer

Masters, Computer Science, Stony Brook University

1063 morse avenue, apt 3-200,Sunnyvale,California-94089

[ashishtanwer@gmail.com](mailto:ashishtanwer@gmail.com), +1-[(631) 551-3060](tel:%28631%29%20551-3060)

<http://ashishtanwer.info>, <http://in.linkedin.com/in/ashishtanwer>

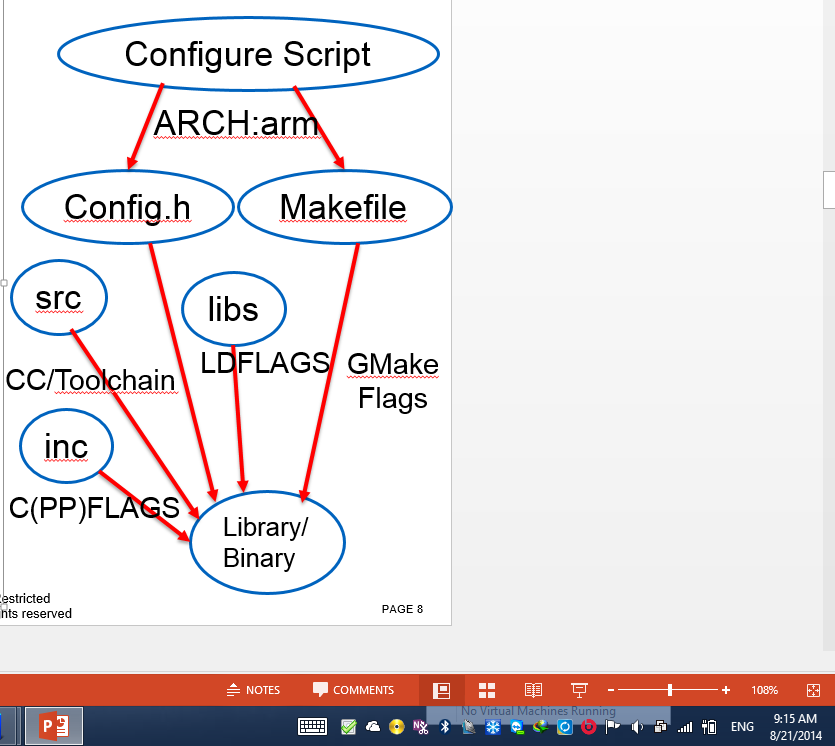
<http://www.youtube.com/user/ashishtanwer> <https://github.com/ashishtanwer>

# NDK PORTING

## **AIM:** Integration of New Tools in Motorola Android for Profiling, Performance/ Power/ Memory/ Code / Core Dump Analysis etc.

## **NDK PORTING**: Configuring to get config.h and cross-compiling with Android Build Parameters.

## **Ported:** Atmel Object Protocol Firmware Upgrade Utility, Busybox, Bootcharts, Strace, libnl, libncurses, lynx,



## **IMPORTANT FLAGS**

export ANDROID\_ROOT=/home/ashishtanwer/wistron/LINUX/android

## CC=arm-eabi-gcc

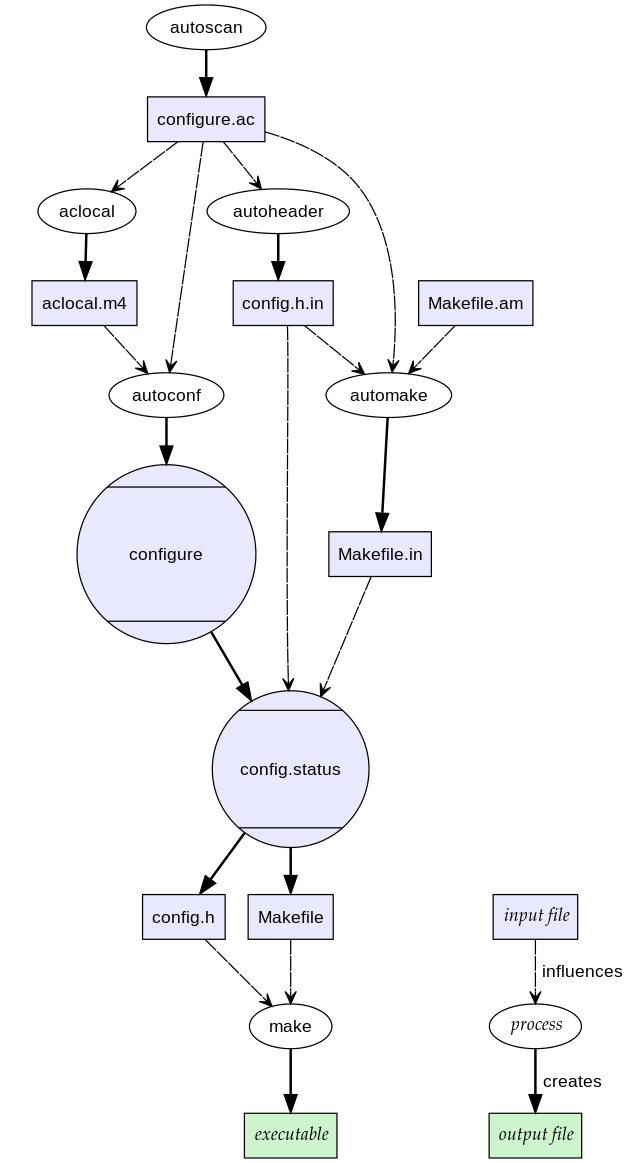
## CPPFLAGS="-I$ANDROID\_ROOT/prebuilts/ndk/9/platforms/android-18/arch-arm/usr/include/“

## CFLAGS="-nostdlib“

## LDFLAGS="-Wl,-L$ANDROID\_ROOT/prebuilts/ndk/9/platforms/android-18/arch-arm/usr/lib/“

## LIBS="-lc“

## **Benefit:** Fast Porting, less efforts.



**DETAILS**

You write a configure.ac file, which is read in by Autoreconf and generates a configure script. Configure script test the build system for headers, libraries etc. and generates a config.h which contains definitions like #define HAVE\_XXX 1 and also generates Makefiles after reading in the Makefile.in files which in turn is generated using Automake by reading in Makefile.am.

**configure.ac —> input to –> autoreconf –> generates -> configure script –> checks host system for headers, libraries etc. -> generates -> config.h**

Also

**Makefile.am –> input to automake –> generates –> Makefile.in –> input to –> configure script –> generates Makefile**

The code uses the generated config.h in the following fashion:

**#ifdef HAVE\_XXXX**

**#include <some-header>**

**#endif**

**….**

**….**

**#ifdef HAVE\_XXXX**

**Call some function which is declared in the header**

**#else**

**Provide some other mechanism or report error to user or do whatever you want**

**#endif**

### **The problem?**

Most Open Source libraries use GNU autotools and its friends for building.

**The first** problem is that the Autools generate some configuration headers based on build time probe of the system. By build time probe I mean checking for things like if a header, library or a tool is present or not. In cross compiling scenario some of these probe should be done on the target system and not on the build system.

**Second,** the build system for most cross compiler tools have their own quirks which need passing some extra flags.

**Third,** in case of Android, it provides its own build system which are essentially some Makefile definitions and rules. This is provided so that people can build their code easily without having to deal with usual cross compiling issues. Thus there is a gap that while your autotools would generate the Makefiles while Android build system requires its own styled Makefiles in the form of Android.mk. One cannot simply generate Android.mk files using autotools.

**Fourth,** even if one gets to write Android specific Makefiles, the build would most probably fail as during the build it would look for a file config.h included in the fashion shown below, while no such file would exist as it is generated by the configure script.

**#ifdef HAVE\_CONFIG\_H**

**#include “config.h”**

**#endif**

Simply copying a config.h file from a run of configure script on another build system wouldn’t really work as the header files and other libraries present on Android may not match with the header and libraries present on the build system. Thus config.h would probably differ if it is somehow generated for Android with the one generated on a build system.

So how does one build an open source library for Android?

### **Solution:**

The way I have managed to work around this trouble is to run the configure script with right cross compilation variables so that a config.h matching my Android system gets built and then writing Android.mk files which would simply use the Android build system.

The way I figured out the right flags was by building the Android source tree which displayed what flags are being used for building and then taking the cues from there, I passed the right flags to the configure script.

It looks something like this for building Android target API on a linux host:

**export ANDROID\_ROOT=/home/ashishtanwer/wistron/LINUX/android**

The command above is sets the path where the Android sources are checked out from git respository.

**NOTE:** It is not necessary to check out the ANDROID sources and you can replace the ANDROID\_ROOT with NDK\_ROOT in all the commands below, along with proper path to the NDK cross compiler.

**export PATH=$PATH:$ANDROID\_ROOT/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/**

The command above is necessary so that configure can find out the path to the cross compiler which is needed to build some test programs during the configure run process. Please note that you can also set the path to the NDK compiler root

**./configure –host=arm-eabi CC=arm-eabi-gcc CPPFLAGS=”-I$ANDROID\_ROOT/build/platforms/android-3/arch-arm/usr/include/” CFLAGS=”-nostdlib” LDFLAGS=”-Wl,-rpath-link=$ANDROID\_ROOT/build/platforms/android-3/arch-arm/usr/lib/ -L$ANDROID\_ROOT/build/platforms/android-3/arch-arm/usr/lib/” LIBS=”-lc “**

The command above has several points that should be well understood.

**–host=arm-eabi –>** This tells the configure script if the cross compilation is being done or not. It is also used as a prefix to some of the cross compiler tools like strip, nm etc.

**CC=arm-eabi-gcc –>** This tells the compiler that should be used for building

**CPPFLAGS –>** This tells the location where the header files should be searched which were specified in configure.ac with AC\_CHECK\_HEADER macro

**CFLAGS=”-nostdlib”** passes the option to build some test programs during configure process run. If you don’t pass this the compiler would link the standard C library of the host system which wouldn’t be compatible with the C library of the target system. You will end up getting error something like this, if you don’t pass this option:

**/home/ashishtanwer/wistron/LINUX/android/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/../lib/gcc/arm-eabi/4.2.1/../../../../arm-eabi/bin/ld: crt0.o: No such file: No such file or directory**

**LIBS=”-lc” –>** This option tells that it should explicitly link to a library called libc.so which is present in the location specified using the -L in the LDFLAGS option. If you are wondering that usually to build a C executable one doesn’t need to provide -lc as libc is automatically linked, then why do we need to specify this here? The answer lies in -nostdlib flag, which instructs not to link with the standard C library on the build system.

You will end up getting error something like this, if you don’t pass this option:

**/home/ashishtanwer/wistron/LINUX/android/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/../lib/gcc/arm-eabi/4.2.1/../../../../arm-eabi/bin/ld: crt0.o: No such file: No such file or directory  
collect2: ld returned 1 exit status**

**LDFLAGS = –>** This option is also passed to build some test programs during configure process run.If you don’t pass the -Wl,-rpath-link option, then linker does not know where do the libraries dependent on the library specific using LIBS reside. If you don’t pass the -L option then the linker doesn’t know where do the libraries specified in LIBS reside.

You will end up getting error something like this, if you don’t pass the -Wl,-rpath-link option:

**/home/ashishtanwer/wistron/LINUX/android/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/../lib/gcc/arm-eabi/4.2.1/../../../../arm-eabi/bin/ld: warning: libdl.so, needed by /home/divkis01/mydroid/development/ndk/build/platforms/android-3/arch-arm/usr/lib//libc.so, not found (try using -rpath or -rpath-link)  
/home/ashishtanwer/wistron/LINUX/android/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/../lib/gcc/arm-eabi/4.2.1/../../../../arm-eabi/bin/ld: warning: cannot find entry symbol \_start; defaulting to 00008184  
/home/ashishtanwer/wistron/LINUX/android/development/ndk/build/platforms/android-3/arch-arm/usr/lib//libc.so: undefined reference to `dl\_unwind\_find\_exidx’**

You will end up getting error something like this, if you don’t pass the -L option:

**/home/ashishtanwer/wistron/LINUX/android/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/../lib/gcc/arm-eabi/4.2.1/../../../../arm-**

**eabi/bin/ld: cannot find -lc**

**collect2: ld returned 1 exit status**

Once you run the configure script with these flags and options, it will generate the appropriate config.h which is compatible / in sync with your target system. Now you can go ahead and start writing the Android.mk files to build your sources.

# ANDROID MAKESYSTEM PORTING

## **AIM:** Integration of New Tools in Motorola Android for Profiling, Performance/ Power/ Memory/ Code / Core Dump Analysis etc.

## **Android MAKE PORTING**: Write fresh Android.mk equivalent to the GNU Makefile.

## Tools Ported: Orientation Detection Service, Oprofile, autologcat, libnl, libncurces.

## Android MAKE Commands: m/mm/mmm

### Android.mk for Building a C Hello world program

**LOCAL PATH $(call my-dir)**

**include $(CLEAR\_VARS)**

**# give module name**

**LOCAL\_MODULE: = hello\_world**

**# list your C files to compile**

**LOCAL\_SRC\_FILES:= helloworld.c**

**# this option will build executables**

**include $(BUILD\_EXECUTABLE)**

### Android.mk for Building an APK

**LOCAL\_PATH:= $(my-dir)**

**include $(CLEAR\_VARS)**

**LOCAL\_MODULE:= <build name>**

**LOCAL\_SRC\_FILES:= main.c**

**LOCAL\_MODULE\_TAGS:= eng development**

**LOCAL\_SHARED\_LIBRARIES:= cutils**

**include $(BUILD\_EXECUTABLE)**

**(HOST\_) EXECUTABLE, (HOST\_) LIBRARY, (HOST\_) PREBUILT, (HOST\_) SHARED\_LIBRARY, (HOST\_) STATIC\_LIBRARY, PACKAGE, JAVADOC, RAW\_EXECUTABLE, RAW\_STATIC\_LIBRARY, COPY\_HEADERS, KEY\_CHAR\_MAP**

### Android.mk for Adding a Static Java Library

**LOCAL\_PATH:= $(call my-dir)**

**include $(CLEAR\_VARS)**

**# Build all java files in the java subdirectory**

**LOCAL\_SRC\_FILES:= $(call all-subdir-java-files)**

**# Any libraries that this library depends on**

**LOCAL\_JAVA\_LIBRARIES:= android.test.runner**

**# The name of the jar file to create**

**LOCAL\_MODULE:= sample**

**# Build a static jar file.**

**include $(BUILD\_STATIC\_JAVA\_LIBRARY)**

### **Android.mk Variables**

These are the variables that you'll commonly see in Android.mk files, listed alphabetically. First, a note on the variable naming:

* **LOCAL\_** - These variables are set per-module. They are cleared by the include $(CLEAR\_VARS) line, so you can rely on them being empty after including that file. Most of the variables you'll use in most modules are LOCAL\_ variables.
* **PRIVATE\_** - These variables are make-target-specific variables. That means they're only usable within the commands for that module.
* **HOST\_** and **TARGET\_** - These contain the directories and definitions that are specific to either the host or the target builds. Do not set variables that start with HOST\_ or TARGET\_ in your makefiles.
* **BUILD\_** and **CLEAR\_VARS** - These contain the names of well-defined template makefiles to include. Some examples are CLEAR\_VARS and BUILD\_HOST\_PACKAGE.

**Benefit:** Auto includes in the Android Source.

**Make Parameters**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| LOCAL\_C\_INCLUDES | Additional directories to instruct the C/C++ compilers to look for header files in. These paths are rooted at the top of the tree. Use LOCAL\_PATH if you have subdirectories of your own that you want in the include paths. For example:  LOCAL\_C\_INCLUDES += extlibs/zlib-1.2.3 LOCAL\_C\_INCLUDES += $(LOCAL\_PATH)/src  You should not add subdirectories of include to LOCAL\_C\_INCLUDES, instead you should reference those files in the #include statement with their subdirectories. For example:  #include <utils/KeyedVector.h> not ~~#include <KeyedVector.h>~~ |
| LOCAL\_CC | If you want to use a different C compiler for this module, set LOCAL\_CC to the path to the compiler. If LOCAL\_CC is blank, the appropriate default compiler is used. |
| LOCAL\_CFLAGS | If you have additional flags to pass into the C or C++ compiler, add them here. For example:  LOCAL\_CFLAGS += -DLIBUTILS\_NATIVE=1 |
| LOCAL\_CPPFLAGS | If you have additional flags to pass into only the C++ compiler, add them here. For example:  LOCAL\_CPPFLAGS += -ffriend-injection  LOCAL\_CPPFLAGS is guaranteed to be after LOCAL\_CFLAGS on the compile line, so you can use it to override flags listed in LOCAL\_CFLAGS |
| LOCAL\_JAVA\_LIBRARIES | When linking Java apps and libraries, LOCAL\_JAVA\_LIBRARIES specifies which sets of java classes to include. Currently there are two of these: core and framework. In most cases, it will look like this:  LOCAL\_JAVA\_LIBRARIES := core framework  Note that setting LOCAL\_JAVA\_LIBRARIES is not necessary (and is not allowed) when building an APK with "include $(BUILD\_PACKAGE)". The appropriate libraries will be included automatically. |
| LOCAL\_LDFLAGS | You can pass additional flags to the linker by setting LOCAL\_LDFLAGS. Keep in mind that the order of parameters is very important to ld, so test whatever you do on all platforms. |
| LOCAL\_LDLIBS | LOCAL\_LDLIBS allows you to specify additional libraries that are not part of the build for your executable or library. Specify the libraries you want in -lxxx format; they're passed directly to the link line. However, keep in mind that there will be no dependency generated for these libraries. It's most useful in simulator builds where you want to use a library preinstalled on the host. The linker (ld) is a particularly fussy beast, so it's sometimes necessary to pass other flags here if you're doing something sneaky. Some examples:  LOCAL\_LDLIBS += -lcurses -lpthread LOCAL\_LDLIBS += -Wl,-z,origin |
| LOCAL\_MODULE | LOCAL\_MODULE is the name of what's supposed to be generated from your Android.mk. For exmample, for libkjs, the LOCAL\_MODULE is "libkjs" (the build system adds the appropriate suffix -- .so .dylib .dll). For app modules, use LOCAL\_PACKAGE\_NAME instead of LOCAL\_MODULE. |
| LOCAL\_MODULE\_PATH | Instructs the build system to put the module somewhere other than what's normal for its type. If you override this, make sure you also set LOCAL\_UNSTRIPPED\_PATH if it's an executable or a shared library so the unstripped binary has somewhere to go. |
| LOCAL\_MODULE\_TAGS | Set LOCAL\_MODULE\_TAGS to any number of whitespace-separated tags.  This variable controls what build flavors the package gets included in. For example:   * user: include this in user/userdebug builds * eng: include this in eng builds * tests: the target is a testing target and makes it available for tests * optional: don't include this |
| LOCAL\_PACKAGE\_NAME | LOCAL\_PACKAGE\_NAME is the name of an app. For example, Dialer, Contacts, etc. |
| LOCAL\_POST\_PROCESS\_COMMAND | For host executables, you can specify a command to run on the module after it's been linked. You might have to go through some contortions to get variables right because of early or late variable evaluation:  module := $(HOST\_OUT\_EXECUTABLES)/$(LOCAL\_MODULE) LOCAL\_POST\_PROCESS\_COMMAND := /Developer/Tools/Rez -d \_\_DARWIN\_\_ -t APPL\        -d \_\_WXMAC\_\_ -o $(module) Carbon.r |
| LOCAL\_PREBUILT\_EXECUTABLES | When including $(BUILD\_PREBUILT) or $(BUILD\_HOST\_PREBUILT), set these to executables that you want copied. They're located automatically into the right bin directory. |
| LOCAL\_PREBUILT\_JAVA\_LIBRARIES |  |
| LOCAL\_PREBUILT\_LIBS | When including $(BUILD\_PREBUILT) or $(BUILD\_HOST\_PREBUILT), set these to libraries that you want copied. They're located automatically into the right lib directory. |
| LOCAL\_SHARED\_LIBRARIES | These are the libraries you directly link against. You don't need to pass transitively included libraries. Specify the name without the suffix:  LOCAL\_SHARED\_LIBRARIES := \     libutils \     libui \     libaudio \     libexpat \     libsgl |
| LOCAL\_SRC\_FILES | The build system looks at LOCAL\_SRC\_FILES to know what source files to compile -- .cpp .c .y .l .java. For lex and yacc files, it knows how to correctly do the intermediate .h and .c/.cpp files automatically. If the files are in a subdirectory of the one containing the Android.mk, prefix them with the directory name:  LOCAL\_SRC\_FILES := \     file1.cpp \     dir/file2.cpp |
| LOCAL\_STATIC\_JAVA\_LIBRARIES |  |
| LOCAL\_STATIC\_LIBRARIES | These are the static libraries that you want to include in your module. Mostly, we use shared libraries, but there are a couple of places, like executables in sbin and host executables where we use static libraries instead.  LOCAL\_STATIC\_LIBRARIES := \     libutils \     libtinyxml |
| LOCAL\_UNINSTALLABLE\_MODULE |  |
| LOCAL\_UNSTRIPPED\_PATH | Instructs the build system to put the unstripped version of the module somewhere other than what's normal for its type. Usually, you override this because you overrode LOCAL\_MODULE\_PATH for an executable or a shared library. If you overrode LOCAL\_MODULE\_PATH, but not LOCAL\_UNSTRIPPED\_PATH, an error will occur. |
| LOCAL\_WHOLE\_STATIC\_LIBRARIES | These are the static libraries that you want to include in your module without allowing the linker to remove dead code from them. This is mostly useful if you want to add a static library to a shared library and have the static library's content exposed from the shared library.  LOCAL\_WHOLE\_STATIC\_LIBRARIES := \  libsqlite3\_android |
| LOCAL\_YACCFLAGS | Any flags to pass to invocations of yacc for your module. A known limitation here is that the flags will be the same for all invocations of YACC for your module. This can be fixed. If you ever need it to be, just ask.  LOCAL\_YACCFLAGS := -p kjsyy |