### 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. It also means that they're unlikely to change behind your back from modules that are included after yours. This [link to the make documentation](http://www.gnu.org/software/make/manual/make.html#Target_002dspecific) describes more about target-specific variables.
* **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.
* Any other name is fair-game for you to use in your Android.mk. However, remember that this is a non-recursive build system, so it is possible that your variable will be changed by another Android.mk included later, and be different when the commands for your rule / module are executed.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| LOCAL\_AAPT\_FLAGS |  |
| LOCAL\_ACP\_UNAVAILABLE |  |
| LOCAL\_ADDITIONAL\_JAVA\_DIR |  |
| LOCAL\_AIDL\_INCLUDES |  |
| LOCAL\_ALLOW\_UNDEFINED\_SYMBOLS |  |
| LOCAL\_ARM\_MODE |  |
| LOCAL\_ASFLAGS |  |
| LOCAL\_ASSET\_DIR |  |
| LOCAL\_ASSET\_FILES | In Android.mk files that include $(BUILD\_PACKAGE) set this to the set of files you want built into your app. Usually:  LOCAL\_ASSET\_FILES += $(call find-subdir-assets) |
| LOCAL\_BUILT\_MODULE\_STEM |  |
| 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\_CERTIFICATE |  |
| 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\_CLASSPATH |  |
| LOCAL\_COMPRESS\_MODULE\_SYMBOLS |  |
| LOCAL\_COPY\_HEADERS | The set of files to copy to the install include tree. You must also supply LOCAL\_COPY\_HEADERS\_TO.  This is going away because copying headers messes up the error messages, and may lead to people editing those headers instead of the correct ones. It also makes it easier to do bad layering in the system, which we want to avoid. We also aren't doing a C/C++ SDK, so there is no ultimate requirement to copy any headers. |
| LOCAL\_COPY\_HEADERS\_TO | The directory within "include" to copy the headers listed in LOCAL\_COPY\_HEADERS to.  This is going away because copying headers messes up the error messages, and may lead to people editing those headers instead of the correct ones. It also makes it easier to do bad layering in the system, which we want to avoid. We also aren't doing a C/C++ SDK, so there is no ultimate requirement to copy any headers. |
| LOCAL\_CPP\_EXTENSION | If your C++ files end in something other than ".cpp", you can specify the custom extension here. For example:  LOCAL\_CPP\_EXTENSION := .cc  Note that all C++ files for a given module must have the same extension; it is not currently possible to mix different extensions. |
| 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\_CXX | If you want to use a different C++ compiler for this module, set LOCAL\_CXX to the path to the compiler. If LOCAL\_CXX is blank, the appropriate default compiler is used. |
| LOCAL\_DX\_FLAGS |  |
| LOCAL\_EXPORT\_PACKAGE\_RESOURCES |  |
| LOCAL\_FORCE\_STATIC\_EXECUTABLE | If your executable should be linked statically, set LOCAL\_FORCE\_STATIC\_EXECUTABLE:=true. There is a very short list of libraries that we have in static form (currently only libc). This is really only used for executables in /sbin on the root filesystem. |
| LOCAL\_GENERATED\_SOURCES | Files that you add to LOCAL\_GENERATED\_SOURCES will be automatically generated and then linked in when your module is built. See the [Custom Tools](http://android.mk/#custom-tools) template makefile for an example. |
| LOCAL\_INSTRUMENTATION\_FOR |  |
| LOCAL\_INSTRUMENTATION\_FOR\_PACKAGE\_NAME |  |
| LOCAL\_INTERMEDIATE\_SOURCES |  |
| LOCAL\_INTERMEDIATE\_TARGETS |  |
| LOCAL\_IS\_HOST\_MODULE |  |
| LOCAL\_JAR\_MANIFEST |  |
| LOCAL\_JARJAR\_RULES |  |
| 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\_JAVA\_RESOURCE\_DIRS |  |
| LOCAL\_JAVA\_RESOURCE\_FILES |  |
| LOCAL\_JNI\_SHARED\_LIBRARIES |  |
| 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. An error will occur if you forget to.  See [Putting modules elsewhere](http://android.mk/#moving-modules) for more. |
| LOCAL\_MODULE\_STEM |  |
| 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\_NO\_DEFAULT\_COMPILER\_FLAGS |  |
| LOCAL\_NO\_EMMA\_COMPILE |  |
| LOCAL\_NO\_EMMA\_INSTRUMENT |  |
| LOCAL\_NO\_STANDARD\_LIBRARIES |  |
| LOCAL\_OVERRIDES\_PACKAGES |  |
| 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\_PREBUILT\_OBJ\_FILES |  |
| LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES |  |
| LOCAL\_PRELINK\_MODULE |  |
| LOCAL\_REQUIRED\_MODULES | Set LOCAL\_REQUIRED\_MODULES to any number of whitespace-separated module names, like "libblah" or "Email". If this module is installed, all of the modules that it requires will be installed as well. This can be used to, e.g., ensure that necessary shared libraries or providers are installed when a given app is installed. |
| LOCAL\_RESOURCE\_DIR |  |
| LOCAL\_SDK\_VERSION |  |
| 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.  See [Putting modules elsewhere](http://android.mk/#moving-modules) for more. |
| 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 |
| OVERRIDE\_BUILT\_MODULE\_PATH |  |

[↑ Go to top](http://android.mk/#top)

Except as noted, this content is licensed under [Creative Commons Attribution 2.5](http://creativecommons.org/licenses/by/2.5/).