Hashir's Blog

Ltes Fgirue ti tuo !

Android.mk Documentation

leave a comment »

30 Votes

```
Android.mk file syntax specification

Introduction:
-------------

This document describes the syntax of Android.mk build file
written to describe your C and C++ source files to the Android
NDK. To understand what follows, it is assumed that you have
read the docs/OVERVIEW.TXT file that explains their role and
usage.

Overview:
---------

An Android.mk file is written to describe your sources to the
build system. More specifically:

- The file is really a tiny GNU Makefile fragment that will be
  parsed one or more times by the build system. As such, you
  should try to minimize the variables you declare there and
  do not assume that anything is not defined during parsing.

- The file syntax is designed to allow you to group your
  sources into 'modules'. A module is one of the following:

    - a static library
    - a shared library

  Only shared libraries will be installed/copied to your
  application package. Static libraries can be used to generate
  shared libraries though.

  You can define one or more modules in each Android.mk file,
  and you can use the same source file in several modules.

- The build system handles many details for you. For example, you
  don't need to list header files or explicit dependencies between
  generated files in your Android.mk. The NDK build system will
  compute these automatically for you.

  This also means that, when updating to newer releases of the NDK,
  you should be able to benefit from new toolchain/platform support
  without having to touch your Android.mk files.

Note that the syntax is *very* close to the one used in Android.mk files
distributed with the full open-source Android platform sources. While
the build system implementation that uses them is different, this is
an intentional design decision made to allow reuse of 'external' libraries'
source code easier for application developers.

Simple example:
---------------

Before describing the syntax in details, let's consider the simple
```

"hello JNI" example, i.e. the files under:

    apps/hello-jni/project

Here, we can see:

  - The 'src' directory containing the Java sources for the
    sample Android project.

  - The 'jni' directory containing the native source for
    the sample, i.e. 'jni/hello-jni.c'

    This source file implements a simple shared library that
    implements a native method that returns a string to the
    VM application.

  - The 'jni/Android.mk' file that describes the shared library
    to the NDK build system. Its content is:

```
   ---------- cut here ------------------
   LOCAL_PATH := $(call my-dir)

   include $(CLEAR_VARS)

   LOCAL_MODULE    := hello-jni
   LOCAL_SRC_FILES := hello-jni.c

   include $(BUILD_SHARED_LIBRARY)
   ---------- cut here ------------------
```

Now, let's explain these lines:

```
  LOCAL_PATH := $(call my-dir)
```

An Android.mk file must begin with the definition of the LOCAL_PATH variable.
It is used to locate source files in the development tree. In this example,
the macro function 'my-dir', provided by the build system, is used to return
the path of the current directory (i.e. the directory containing the
Android.mk file itself).

```
  include $(CLEAR_VARS)
```

The CLEAR_VARS variable is provided by the build system and points to a
special GNU Makefile that will clear many LOCAL_XXX variables for you
(e.g. LOCAL_MODULE, LOCAL_SRC_FILES, LOCAL_STATIC_LIBRARIES, etc...),
with the exception of LOCAL_PATH. This is needed because all build
control files are parsed in a single GNU Make execution context where
all variables are global.

```
  LOCAL_MODULE := hello-jni
```

The LOCAL_MODULE variable must be defined to identify each module you
describe in your Android.mk. The name must be *unique* and not contain
any spaces. Note that the build system will automatically add proper
prefix and suffix to the corresponding generated file. In other words,
a shared library module named 'foo' will generate 'libfoo.so'.

IMPORTANT NOTE:
If you name your module 'libfoo', the build system will not
add another 'lib' prefix and will generate libfoo.so as well.
This is to support Android.mk files that originate from the
Android platform sources, would you need to use these.

```
  LOCAL_SRC_FILES := hello-jni.c
```

The LOCAL_SRC_FILES variables must contain a list of C and/or C++ source
files that will be built and assembled into a module. Note that you should
not list header and included files here, because the build system will
compute dependencies automatically for you; just list the source files
that will be passed directly to a compiler, and you should be good.

```
Note that the default extension for C++ source files is '.cpp'. It is
however possible to specify a different one by defining the variable
LOCAL_DEFAULT_CPP_EXTENSION. Don't forget the initial dot (i.e. '.cxx'
will work, but not 'cxx').

  include $(BUILD_SHARED_LIBRARY)

The BUILD_SHARED_LIBRARY is a variable provided by the build system that
points to a GNU Makefile script that is in charge of collecting all the
information you defined in LOCAL_XXX variables since the latest
'include $(CLEAR_VARS)' and determine what to build, and how to do it
exactly. There is also BUILD_STATIC_LIBRARY to generate a static library.

There are more complex examples under apps/, with commented
Android.mk files that you can look at.

Reference:
----------

This is the list of variables you should either rely on or define in
an Android.mk. You can define other variables for your own usage, but
the NDK build system reserves the following variable names:

- names that begin with LOCAL_  (e.g. LOCAL_MODULE)
- names that begin with PRIVATE_, NDK_ or APP_  (used internally)
- lower-case names (used internally, e.g. 'my-dir')

If you need to define your own convenience variables in an Android.mk
file, we recommend using the MY_ prefix, for a trivial example:

   ---------- cut here ------------------
    MY_SOURCES := foo.c
    ifneq ($(MY_CONFIG_BAR),)
      MY_SOURCES += bar.c
    endif

    LOCAL_SRC_FILES += $(MY_SOURCES)
   ---------- cut here ------------------

So, here we go:

NDK-provided variables:
- - - - - - - - - - - -

These GNU Make variables are defined by the build system before
your Android.mk file is parsed. Note that under certain circumstances
the NDK might parse your Android.mk several times, each with different
definition for some of these variables.

CLEAR_VARS
    Points to a build script that undefines nearly all LOCAL_XXX variables
    listed in the "Module-description" section below. You must include
    the script before starting a new module, e.g.:

      include $(CLEAR_VARS)

BUILD_SHARED_LIBRARY
    Points to a build script that collects all the information about the
    module you provided in LOCAL_XXX variables and determines how to build
    a target shared library from the sources you listed. Note that you
    must have LOCAL_MODULE and LOCAL_SRC_FILES defined, at a minimum before
    including this file. Example usage:

      include $(BUILD_SHARED_LIBRARY)

    note that this will generate a file named lib$(LOCAL_MODULE).so

BUILD_STATIC_LIBRARY
    A variant of BUILD_SHARED_LIBRARY that is used to build a target static
```

```
    library instead. Static libraries are not copied into your
    project/packages but can be used to build shared libraries (see
    LOCAL_STATIC_LIBRARIES and LOCAL_STATIC_WHOLE_LIBRARIES described below).
    Example usage:

      include $(BUILD_STATIC_LIBRARY)

    Note that this will generate a file named lib$(LOCAL_MODULE).a

TARGET_ARCH
    Name of the target CPU architecture as it is specified by the
    full Android open-source build. This is 'arm' for any ARM-compatible
    build, independent of the CPU architecture revision.

TARGET_PLATFORM
    Name of the target Android platform when this Android.mk is parsed.
    For now, only 'android-3' is supported, which corresponds to the
    Android 1.5 platform.

TARGET_ARCH_ABI
    Name of the target CPU+ABI when this Android.mk is parsed.
    For now, only 'arm' is supported, which really means the following:

       ARMv5TE or higher CPU, with 'softfloat' floating point support

    Other target ABIs will be introduced in future releases of the NDK
    and will have a different name. Note that all ARM-based ABIs will
    have 'TARGET_ARCH' defined to 'arm', but may have different
    'TARGET_ARCH_ABI'

TARGET_ABI
    The concatenation of target platform and abi, it really is defined
    as $(TARGET_PLATFORM)-$(TARGET_ARCH_ABI) and is useful when you want
    to test against a specific target system image for a real device.

    By default, this will be 'android-3-arm'

NDK-provided function macros:
- - - - - - - - - - - - - - - -

The following are GNU Make 'function' macros, and must be evaluated
by using '$(call <function>)'. They return textual information.

my-dir
    Returns the path of the current Android.mk's directory, relative
    to the top of the NDK build system. This is useful to define
    LOCAL_PATH at the start of your Android.mk as with:

        LOCAL_PATH := $(call my-dir)

all-subdir-makefiles
    Returns a list of Android.mk located in all sub-directories of
    the current 'my-dir' path. For example, consider the following
    hierarchy:

        sources/foo/Android.mk
        sources/foo/lib1/Android.mk
        sources/foo/lib2/Android.mk

    If sources/foo/Android.mk contains the single line:

        include $(call all-subdir-makefiles)

    Then it will include automatically sources/foo/lib1/Android.mk and
    sources/foo/lib2/Android.mk

    This function can be used to provide deep-nested source directory
    hierarchies to the build system. Note that by default, the NDK
    will only look for files in sources/*/Android.mk
```

```
this-makefile
    Returns the path of the current Makefile (i.e. where the function
    is called).

parent-makefile
    Returns the path of the parent Makefile in the inclusion tree,
    i.e. the path of the Makefile that included the current one.

grand-parent-makefile
    Guess what...

Module-description variables:
- - - - - - - - - - - - - - -

The following variables are used to describe your module to the build
system. You should define some of them between an 'include $(CLEAR_VARS)'
and an 'include $(BUILD_XXXXX)'. As written previously, $(CLEAR_VARS) is
a script that will undefine/clear all of these variables, unless explicitely
noted in their description.

LOCAL_PATH
    This variable is used to give the path of the current file.
    You MUST define it at the start of your Android.mk, which can
    be done with:

      LOCAL_PATH := $(call my-dir)

    This variable is *not* cleared by $(CLEAR_VARS) so only one
    definition per Android.mk is needed (in case you define several
    modules in a single file).

LOCAL_MODULE
    This is the name of your module. It must be unique among all
    module names, and shall not contain any space. You MUST define
    it before including any $(BUILD_XXXX) script.

    The module name determines the name of generated files, e.g.
    lib<foo>.so for a shared library module named <foo>. However
    you should only refer to other modules with their 'normal'
    name (e.g. <foo>) in your NDK build files (either Android.mk
    or Application.mk)

LOCAL_SRC_FILES
    This is a list of source files that will be built for your module.
    Only list the files that will be passed to a compiler, since the
    build system automatically computes dependencies for you.

    Note that source files names are all relative to LOCAL_PATH and
    you can use path components, e.g.:

      LOCAL_SRC_FILES := foo.c \
                         toto/bar.c

    NOTE: Always use Unix-style forward slashes (/) in build files.
          Windows-style back-slashes will not be handled properly.

LOCAL_CPP_EXTENSION
    This is an optional variable that can be defined to indicate
    the file extension of C++ source files. The default is '.cpp'
    but you can change it. For example:

        LOCAL_CPP_EXTENSION := .cxx

LOCAL_C_INCLUDES
    An optional list of paths, relative to the NDK *root* directory,
    which will be appended to the include search path when compiling
    all sources (C, C++ and Assembly). For example:

        LOCAL_C_INCLUDES := sources/foo
```

```
    Or even:

        LOCAL_C_INCLUDES := $(LOCAL_PATH)/../foo

    These are placed before any corresponding inclusion flag in
    LOCAL_CFLAGS / LOCAL_CPPFLAGS

LOCAL_CFLAGS
    An optional set of compiler flags that will be passed when building
    C *and* C++ source files.

    This can be useful to specify additionnal macro definitions or
    compile options.

    IMPORTANT: Try not to change the optimization/debugging level in
               your Android.mk, this can be handled automatically for
               you by specifying the appropriate information in
               your Application.mk, and will let the NDK generate
               useful data files used during debugging.

    NOTE: In android-ndk-1.5_r1, the corresponding flags only applied
          to C source files, not C++ ones. This has been corrected to
          match the full Android build system behaviour. (You can use
          LOCAL_CPPFLAGS to specify flags for C++ sources only now).

LOCAL_CXXFLAGS
    An alias for LOCAL_CPPFLAGS. Note that use of this flag is obsolete
    as it may disappear in future releases of the NDK.

LOCAL_CPPFLAGS
    An optional set of compiler flags that will be passed when building
    C++ source files *only*. They will appear after the LOCAL_CFLAGS
    on the compiler's command-line.

    NOTE: In android-ndk-1.5_r1, the corresponding flags applied to
          both C and C++ sources. This has been corrected to match the
          full Android build system. (You can use LOCAL_CFLAGS to specify
          flags for both C and C++ sources now).

LOCAL_STATIC_LIBRARIES
    The list of static libraries modules (built with BUILD_STATIC_LIBRARY)
    that should be linked to this module. This only makes sense in
    shared library modules.

LOCAL_SHARED_LIBRARIES
    The list of shared libraries *modules* this module depends on at runtime.
    This is necessary at link time and to embed the corresponding information
    in the generated file.

    Note that this does not append the listed modules to the build graph,
    i.e. you should still add them to your application's required modules
    in your Application.mk

LOCAL_LDLIBS
    The list of additional linker flags to be used when building your
    module. This is useful to pass the name of specific system libraries
    with the "-l" prefix. For example, the following will tell the linker
    to generate a module that links to /system/lib/libz.so at load time:

      LOCAL_LDLIBS := -lz

    See docs/STABLE-APIS.TXT for the list of exposed system libraries you
    can linked against with this NDK release.

LOCAL_ALLOW_UNDEFINED_SYMBOLS
    By default, any undefined reference encountered when trying to build
    a shared library will result in an "undefined symbol" error. This is a
    great help to catch bugs in your source code.

    However, if for some reason you need to disable this check, set this
```

```
    variable to 'true'. Note that the corresponding shared library may fail
    to load at runtime.

LOCAL_ARM_MODE
    By default, ARM target binaries will be generated in 'thumb' mode, where
    each instruction are 16-bit wide. You can define this variable to 'arm'
    if you want to force the generation of the module's object files in
    'arm' (32-bit instructions) mode. E.g.:

      LOCAL_ARM_MODE := arm

    Note that you can also instruct the build system to only build specific
    sources in arm mode by appending an '.arm' suffix to its source file
    name. For example, with:

       LOCAL_SRC_FILES := foo.c bar.c.arm

    Tells the build system to always compile 'bar.c' in arm mode, and to
    build foo.c according to the value of LOCAL_ARM_MODE.

    NOTE: Setting APP_OPTIM to 'debug' in your Application.mk will also force
          the generation of ARM binaries as well. This is due to bugs in the
          toolchain debugger that don't deal too well with thumb code.
```

Written by Hashir N A

January 27, 2010 at 8:53 am

Posted in Google Android

Tagged with android

« ASE – Android Scripting Environments
3 ways to audio record/capture in android »