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

The LOCAL\_C\_INCLUDES path are also used automatically when launching native debugging with ndk-gdb.

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 additional 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).

It is possible to specify additional include paths with

LOCAL\_CFLAGS += -I<path>, however, it is better to use LOCAL\_C\_INCLUDES

for this, since the paths will then also be used during native debugging with ndk-gdb.

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 Note that use of this flag is obsolete as it may disappear in future releases of the NDK.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.

LOCAL\_WHOLE\_STATIC\_LIBRARIES

A variant of LOCAL\_STATIC\_LIBRARIES used to express that the corresponding library module should be used as "whole archives" to the linker. See the

GNU linker's documentation for the --whole-archive flag.

This is generally useful when there are circular dependencies between several static libraries. Note that when used to build a shared library,

this will force all object files from your whole static libraries to be added to the final binary. This is not true when generating executables

though.

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.html 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.