Android Developers

Android.mk

This page describes the syntax of the `Android.mk` build file, which glues your C and C++ source files to the Android NDK.

On this page

Overview
Basics
Variables and Macros
Module-Description Variables

Overview

The `Android.mk` file resides in a subdirectory of your project's `jni/` directory, and describes your sources and shared libraries to the build system. It is really a tiny GNU makefile fragment that the build system parses once or more. The `Android.mk` file is useful for defining project-wide settings that `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html), the build system, and your environment variables leave undefined. It can also override project-wide settings for specific *modules*.

The syntax of the `Android.mk` allows you to group your sources into *modules*. A module is either a static library, a shared library, or a standalone executable. You can define one or more modules in each `Android.mk` file, and you can use the same source file in multiple modules. The build system only places shared libraries into your application package. In addition, static libraries can generate shared libraries.

In addition to packaging libraries, the build system handles a variety of other details for you. For example, you don't need to list header files or explicit dependencies between generated files in your `Android.mk` file. The NDK build system computes these relationships automatically for you. As a result, you should be able to benefit from new toolchain/platform support in future NDK releases without having to touch your `Android.mk` file.

The syntax of this file is very close to that used in the `Android.mk` files distributed with the full Android Open Source Project (https://source.android.com). While the build system implementation that uses them is different, their similarity is an intentional design decision aimed at making it easier for application developers to reuse source code for external libraries.

Basics

Before exploring the syntax in detail, it is useful to start by understanding the basics of what a

`Android.mk` file contains. This section uses the `Android.mk` file in the Hello-JNI sample toward that end, explaining the role that each line in the file plays.

An `Android.mk` file must begin by defining the `LOCAL_PATH` variable:

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

This variable indicates the location of the source files in the development tree. Here, the macro function `my-dir`, provided by the build system, returns the path of the current directory (the directory containing the `Android.mk` file itself).

The next line declares the `CLEAR_VARS` variable, whose value the build system provides.

```
include $(CLEAR_VARS)
```

The `CLEAR_VARS` variable points to a special GNU Makefile that clears many `LOCAL_XXX` variables for you, such as `LOCAL_MODULE`, `LOCAL_SRC_FILES`, and `LOCAL_STATIC_LIBRARIES`. Note that it does not clear `LOCAL_PATH`. This variable must retain its value because the system parses all build control files in a single GNU Make execution context where all variables are global. You must (re-)declare this variable before describing each module.

Next, the `LOCAL_MODULE` variable stores the name of the module that you wish to build. Use this variable once per module in your application.

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

Each module name must be unique and not contain any spaces. The build system, when it generates the final shared-library file, automatically adds the proper prefix and suffix to the name that you assign to `LOCAL_MODULE`. For example, the example that appears above results in generation of a library called `libhello-jni.so`.

> **Note:** If your module's name already starts with `lib`, the build system does not prepend an additional `lib` prefix; it takes the module name as-is, and adds the `.so` extension. So a source file originally called, for example, `libfoo.c` still produces a shared-object file called `libfoo.so`. This behavior is to support libraries that the Android platform sources generate from `Android.mk` files; the names of all such libraries start with `lib`.

The next line enumerates the source files, with spaces delimiting multiple files:

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

The `LOCAL_SRC_FILES` variable must contain a list of C and/or C++ source files to build into a module.

The last line helps the system tie everything together:

```
include $(BUILD_SHARED_LIBRARY)
```

The `BUILD_SHARED_LIBRARY` variable points to a GNU Makefile script that collects all the information you defined in `LOCAL_XXX` variables since the most recent `include`. This script determines what to build, and how to do it.

There are more complex examples in the samples directories, with commented `Android.mk` files that you can look at. In addition, Sample: native-activity (https://developer.android.com/ndk/samples/sample_na.html) provides a detailed explanation of that sample's `Android.mk` file. Finally, Variables and Macros (#var) provides further information on the variables from this section.

Variables and Macros

The build system provides many possible variables for use in the the `Android.mk` file. Many of these variables come with preassigned values. Others, you assign.

In addition to these variables, you can also define your own arbitrary ones. If you do so, keep in mind that the NDK build system reserves the following variable names:

- Names that begin with `LOCAL_`, such as `LOCAL_MODULE`.
- Names that begin with `PRIVATE_`, `NDK_`, or `APP`. The build system uses these internally.
- Lower-case names, such as `my-dir`. The build system uses these internally, as well.

If you need to define your own convenience variables in an `Android.mk` file, we recommend prepending `MY_` to their names.

NDK-defined variables

This section discusses the GNU Make variables that the build system defines before parsing your `Android.mk` file. Under certain circumstances, the NDK might parse your `Android.mk` file several times, using a different definition for some of these variables each time.

CLEAR_VARS

This variable points to a build script that undefines nearly all `LOCAL_XXX` variables listed in the "Developer-defined variables" section below. Use this variable to include this script before describing a new module. The syntax for using it is:

```
include $(CLEAR_VARS)
```

BUILD_SHARED_LIBRARY

This variable points to a build script that collects all the information about the module you provided in your `LOCAL_XXX` variables, and determines how to build a target shared library from the sources you listed. Note that using this script requires that you have already assigned values to `LOCAL_MODULE` and `LOCAL_SRC_FILES`, at a minimum (for more information about these variables, see Module-Description Variables (#mdv)).

The syntax for using this variable is:

```
include $(BUILD_SHARED_LIBRARY)
```

A shared-library variable causes the build system to generate a library file with a `.so` extension.

BUILD_STATIC_LIBRARY

A variant of `BUILD_SHARED_LIBRARY` that is used to build a static library. The build system does not copy static libraries into your project/packages, but it can use them to build shared libraries (see `LOCAL_STATIC_LIBRARIES` and `LOCAL_WHOLE_STATIC_LIBRARIES`, below). The syntax for using this variable is:

```
include $(BUILD_STATIC_LIBRARY)
```

A static-library variable causes the build system to generate a library with a `.a` extension.

PREBUILT_SHARED_LIBRARY

Points to a build script used to specify a prebuilt shared library. Unlike in the case of `BUILD_SHARED_LIBRARY` and `BUILD_STATIC_LIBRARY`, here the value of `LOCAL_SRC_FILES` cannot be a source file. Instead, it must be a single path to a prebuilt shared library, such as `foo/libfoo.so`. The syntax for using this variable is:

```
include $(PREBUILT_SHARED_LIBRARY)
```

You can also reference a prebuilt library in another module by using the `LOCAL_PREBUILTS` variable. For more information about using prebuilts, see Using Prebuilt Libraries (https://developer.android.com/ndk/guides/prebuilts.html).

PREBUILT_STATIC_LIBRARY

The same as `PREBUILT_SHARED_LIBRARY`, but for a prebuilt static library. For more information about using prebuilts, see Using Prebuilt Libraries (https://developer.android.com/ndk/guides/prebuilts.html).

TARGET_ARCH

The name of the target CPU architecture as the Android Open Source Project specifies it. For any ARM-compatible build, use `arm`, independent of the CPU architecture revision or ABI (see TARGET_ARCH_ABI, below).

The value of this variable is taken from the APP_ABI variable that you define in the `Android.mk` file, which the system reads ahead of parsing the `Android.mk` file.

TARGET_PLATFORM

The Android API level number for the build system to target. For example, the Android 5.1 system images correspond to Android API level 22: `android-22`. For a complete list of platform names and corresponding Android system images, see Android NDK Native APIs (https://developer.android.com/ndk/guides /stable_apis.html). The following example shows the syntax for using this variable:

```
TARGET_PLATFORM := android-22
```

TARGET_ARCH_ABI

This variable stores the name of the CPU and architecture to target when the build system parses this `Android.mk` file. You can specify one or more of the following values, using a space as a delimiter between multiple targets. Table 1 shows the ABI setting to use for each supported CPU and architecture.

Table 1. ABI settings for different CPUs and architectures.

CPU and architecture	Setting
ARMv5TE	`armeabi`
ARMv7	`armeabi-v7a`
ARMv8 AArch64	`arm64-v8a`
i686	`x86`
x86-64	`x86_64`
mips32 (r1)	`mips`
mips64 (r6)	`mips64`
All	`all`

The following example shows how to set ARMv8 AArch64 as the target CPU-and-ABI combination:

```
TARGET_ARCH_ABI := arm64-v8a
```

> **Note:** Up to Android NDK 1.6_r1, this variable is defined as `arm`.

For more details about architecture ABIs and associated compatibility issues, refer to ABI Management (https://developer.android.com/ndk/guides/abis.html).

New target ABIs in the future will have different values.

TARGET_ABI

A concatenation of target Android API level and ABI, it is especially useful when you want to test against a specific target system image for a real device. For example, to specify a 64-bit ARM device running on Android API level 22:

```
TARGET_ABI := android-22-arm64-v8a
```

> **Note:** Up to Android NDK 1.6_r1, the default value was `android-3-arm`.

Module-Description Variables

The variables in this section describe your module to the build system. Each module description should follow this basic flow:

1. Initialize or undefine the variables associated with the module, using the `CLEAR_VARS` variable.
2. Assign values to the variables used to describe the module.
3. Set the NDK build system to use the appropriate build script for the module, using the `BUILD_XXX` variable.

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` file. The following example shows how to do so:

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

The script to which `CLEAR_VARS` points does not clear this variable. Therefore, you only need to define it a single time, even if your Android.mk file describes multiple modules.

LOCAL_MODULE

This variable stores the name of your module. It must be unique among all module names, and must not contain any spaces. You must define it before including any scripts (other than the one for CLEAR_VARS). You need not add either the lib prefix or the .so or .a file extension; the build system makes these modifications automatically. Throughout your Android.mk and Application.mk (https://developer.android.com/ndk/guides/application_mk.html) files, refer to your module by its unmodified name. For example, the following line results in the generation of a shared library module called libfoo.so:

```
LOCAL_MODULE := "foo"
```

If you want the generated module to have a name other than lib + the value of LOCAL_MODULE, you can use the LOCAL_MODULE_FILENAME variable to give the generated module a name of your own choosing, instead.

LOCAL_MODULE_FILENAME

This optional variable allows you to override the names that the build system uses by default for files that it generates. For example, if the name of your LOCAL_MODULE is foo, you can force the system to call the file it generates libnewfoo. The following example shows how to accomplish this:

```
LOCAL_MODULE := foo
LOCAL_MODULE_FILENAME := libnewfoo
```

For a shared library module, this example would generate a file called libnewfoo.so.

> **Note:** You cannot override filepath or file extension.

LOCAL_SRC_FILES

This variable contains the list of source files that the build system uses to generate the module. Only list the files that the build system actually passes to the compiler, since the build system automatically computes any associated depencies.

Note that you can use both relative (to LOCAL_PATH) and absolute file paths.

We recommend avoiding absolute file paths; relative paths make your Android.mk file more portable.

> **Note:** Always use Unix-style forward slashes (/) in build files. The build system does not handle Windows-style backslashes (\) properly.

LOCAL_CPP_EXTENSION

You can use this optional variable to indicate a file extension other than `.cpp` for your C++ source files. For example, the following line changes the extension to `.cxx`. (The setting must include the dot.)

```
LOCAL_CPP_EXTENSION := .cxx
```

From NDK r7, you can use this variable to specify multiple extensions. For instance:

```
LOCAL_CPP_EXTENSION := .cxx .cpp .cc
```

LOCAL_CPP_FEATURES

You can use this optional variable to indicate that your code relies on specific C++ features. It enables the right compiler and linker flags during the build process. For prebuilt binaries, this variable also declares which features the binary depends on, thus helping ensure the final linking works correctly. We recommend that you use this variable instead of enabling `-frtti` and `-fexceptions` directly in your `LOCAL_CPPFLAGS` definition.

Using this variable allows the build system to use the appropriate flags for each module. Using `LOCAL_CPPFLAGS` causes the compiler to use all specified flags for all modules, regardless of actual need.

For example, to indicate that your code uses RTTI (RunTime Type Information), write:

```
LOCAL_CPP_FEATURES := rtti
```

To indicate that your code uses C++ exceptions, write:

```
LOCAL_CPP_FEATURES := exceptions
```

You can also specify multiple values for this variable. For example:

```
LOCAL_CPP_FEATURES := rtti features
```

The order in which you describe the values does not matter.

LOCAL_C_INCLUDES

You can use this optional variable to specify a list of paths, relative to the NDK `root` directory, to add 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
```

Define this variable before setting any corresponding inclusion flags via `LOCAL_CFLAGS` or `LOCAL_CPPFLAGS`.

The build system also uses `LOCAL_C_INCLUDES` paths automatically when launching native debugging with ndk-gdb.

LOCAL_CFLAGS

This optional variable sets compiler flags for the build system to pass when building C *and* C++ source files. The ability to do so can be useful for specifying additional macro definitions or compile options.

Try not to change the optimization/debugging level in your `Android.mk` file. The build system can handle this setting automatically for you, using the relevant information in the `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file. Doing it this way allows the build system to 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. They now match the full Android build system behavior. (You can now use `LOCAL_CPPFLAGS` to specify flags for C++ sources only.)

It is possible to specify additional include paths by writing:

```
LOCAL_CFLAGS += -I<path>,
```

It is better, however, to use `LOCAL_C_INCLUDES` for this purpose, since doing so also makes it possible to use the paths available for native debugging with ndk-gdb.

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. To specify flags for both C and C++ sources, use `LOCAL_CFLAGS`.

LOCAL_STATIC_LIBRARIES

This variable stores the list of static libraries modules on which the current module depends.

If the current module is a shared library or an executable, this variable will force these libraries to be linked into the resulting binary.

If the current module is a static library, this variable simply indicates that other modules depending on the current one will also depend on the listed libraries.

LOCAL_SHARED_LIBRARIES

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

LOCAL_WHOLE_STATIC_LIBRARIES

This variable is a variant of `LOCAL_STATIC_LIBRARIES`, and expresses that the linker should treat the associated library modules as *whole archives*. For more information on whole archives, see the GNU linker's documentation (http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html) for the `--whole-archive` flag.

This variable is useful when there are circular dependencies among several static libraries. When you use this variable to build a shared library, it will force the build system to add all object files from your static libraries to the final binary. The same is not true, however, when generating executables.

LOCAL_LDLIBS

This variable contains the list of additional linker flags for use in building your shared library or executable. It enables you to use the `-l` prefix to pass the name of specific system libraries. For example, the following example tells the linker to generate a module that links to `/system/lib/libz.so` at load time:

```
LOCAL_LDLIBS := -lz
```

For the list of exposed system libraries against which you can link in this NDK release, see Android NDK Native APIs (https://developer.android.com/ndk/guides/stable_apis.html).

> **Note:** If you define this variable for a static library, the build system ignores it, and `ndk-build` prints a warning.

LOCAL_LDFLAGS

The list of other linker flags for the build system to use when building your shared library or executable. For example, the following example uses the `ld.bfd` linker on ARM/X86 GCC 4.6+, on which `ld.gold` is the default

```
LOCAL_LDFLAGS += -fuse-ld=bfd
```

> **Note:** If you define this variable for a static library, the build system ignores it, and ndk-build prints a warning.

LOCAL_ALLOW_UNDEFINED_SYMBOLS

By default, when the build system encounters an undefined reference encountered while trying to build a shared, it will throw an *undefined symbol* error. This error can help you catch catch bugs in your source code.

To disable this check, set this variable to `true`. Note that this setting may cause the shared library to load at runtime.

> **Note:** If you define this variable for a static library, the build system ignores it, and ndk-build prints a warning.

LOCAL_ARM_MODE

By default, the build system generates ARM target binaries in *thumb* mode, where each instruction is 16 bits wide and linked with the STL libraries in the `thumb/` directory. Defining this variable as `arm` forces the build system to generate the module's object files in 32-bit `arm` mode. The following example shows how to do this:

```
LOCAL_ARM_MODE := arm
```

You can also instruct the build system to only build specific sources in `arm` mode by appending `.arm` suffix to the the source filenames. For example, the following example tells the build system to always compile `bar.c` in ARM mode, but to build `foo.c` according to the value of `LOCAL_ARM_MODE`.

```
LOCAL_SRC_FILES := foo.c bar.c.arm
```

> **Note:** You can also force the build system to generate ARM binaries by setting `APP_OPTIM` in your `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file to `debug`. Specifying `debug` forces an ARM build because the toolchain debugger does not handle Thumb code properly.

LOCAL_ARM_NEON

This variable only matters when you are targeting the `armeabi-v7a` ABI. It allows the use of ARM Advanced SIMD (NEON) GCC intrinsics in your C and C++ sources, as well as NEON instructions in

Assembly files.

Note that not all ARMv7-based CPUs support the NEON instruction set extensions. For this reason, you must perform runtime detection to be able to safely use this code at runtime. For more information, see NEON Support (https://developer.android.com/ndk/guides/cpu-arm-neon.html) and The `cpufeatures` Library (https://developer.android.com/ndk/guides/cpu-features.html).

Alternatively, you can use the `.neon` suffix to specify that the build system only compile specific source files with NEON support. In the following example, the build system compiles `foo.c` with thumb and neon support, `bar.c` with thumb support, and `zoo.c` with support for ARM and NEON:

```
LOCAL_SRC_FILES = foo.c.neon bar.c zoo.c.arm.neon
```

If you use both suffixes, `.arm` must precede `.neon`.

LOCAL_DISABLE_NO_EXECUTE

Android NDK r4 added support for the "NX bit" security feature. It is enabled by default, but you can disable it by setting this variable to `true`. We do not recommend doing so without a compelling reason.

This feature does not modify the ABI, and is only enabled on kernels targeting ARMv6+ CPU devices. Machine code with this feature enabled will run unmodified on devices running earlier CPU architectures.

For more information, see Wikipedia: NX bit (http://en.wikipedia.org/wiki/NX_bit) and The GNU stack kickstart (http://www.gentoo.org/proj/en/hardened/gnu-stack.xml).

LOCAL_DISABLE_RELRO

By default, the NDK compiles code with read-only relocations and GOT protection. This variable instructs the runtime linker to mark certain regions of memory as read-only after relocation, making certain security exploits (such as GOT overwrites) more difficult. Note that these protections are only effective on Android API level 16 and higher. On lower API levels, the code will still run, but without memory protections.

This variable is turned on by default, but you can disable it by setting its value to `true`. We do not recommend doing so without a compelling reason.

For more information, see RELRO: RELocation Read-Only (http://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/) and Security enhancements in RedHat Enterprise Linux (section 6) (http://www.akkadia.org/drepper/nonselsec.pdf).

LOCAL_DISABLE_FORMAT_STRING_CHECKS

By default, the build system compiles code with format string protection. Doing so forces a compiler error if a non-constant format string is used in a `printf`-style function.

This protection is on by default, but you can disable it by setting the value of this variable to `true`. We do not recommend doing so without a compelling reason.

LOCAL_EXPORT_CFLAGS

This variable records a set of C/C++ compiler flags to add to the `LOCAL_CFLAGS` definition of any other module that uses this one via the `LOCAL_STATIC_LIBRARIES` or `LOCAL_SHARED_LIBRARIES` variables.

For example, consider the following pair of modules: `foo` and `bar`, which depends on `foo`:

```
include $(CLEAR_VARS)
LOCAL_MODULE := foo
LOCAL_SRC_FILES := foo/foo.c
LOCAL_EXPORT_CFLAGS := -DFOO=1
include $(BUILD_STATIC_LIBRARY)


include $(CLEAR_VARS)
LOCAL_MODULE := bar
LOCAL_SRC_FILES := bar.c
LOCAL_CFLAGS := -DBAR=2
LOCAL_STATIC_LIBRARIES := foo
include $(BUILD_SHARED_LIBRARY)
```

Here, the build system passes the flags `-DFOO=1` and `-DBAR=2` to the compiler when building `bar.c`. It also prepends exported flags to your your module's `LOCAL_CFLAGS` so you can easily override them.

In addition, the relationship among modules is transitive: If `zoo` depends on `bar`, which in turn depends on `foo`, then `zoo` also inherits all flags exported from `foo`.

Finally, the build system does not use exported flags when building locally (i.e., building the module whose flags it is exporting). Thus, in the example above, it does not pass `-DFOO=1` to the compiler when building `foo/foo.c`. To build locally, use `LOCAL_CFLAGS` instead.

LOCAL_EXPORT_CPPFLAGS

This variable is the same as `LOCAL_EXPORT_CFLAGS`, but for C++ flags only.

LOCAL_EXPORT_C_INCLUDES

This variable is the same as `LOCAL_EXPORT_CFLAGS`, but for C include paths. It is useful in cases where, for example, `bar.c` needs to include headers from module `foo`.

LOCAL_EXPORT_LDFLAGS

This variable is the same as `LOCAL_EXPORT_CFLAGS`, but for linker flags.

LOCAL_EXPORT_LDLIBS

This variable is the same as LOCAL_EXPORT_CFLAGS, telling the build system to pass names of specific system libraries to the compiler. Prepend -l to the name of each library you specify.

Note that the build system appends imported linker flags to the value of your module's LOCAL_LDLIBS variable. It does this due to the way Unix linkers work.

This variable is typically useful when module foo is a static library and has code that depends on a system library. You can then use LOCAL_EXPORT_LDLIBS to to export the dependency. For example:

```
include $(CLEAR_VARS)
LOCAL_MODULE := foo
LOCAL_SRC_FILES := foo/foo.c
LOCAL_EXPORT_LDLIBS := -llog
include $(BUILD_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := bar
LOCAL_SRC_FILES := bar.c
LOCAL_STATIC_LIBRARIES := foo
include $(BUILD_SHARED_LIBRARY)
```

In this example, the build system puts -llog at the end of the linker command when it builds libbar.so. Doing so tells the linker that, because libbar.so depends on foo, it also depends on the system logging library.

LOCAL_SHORT_COMMANDS

Set this variable to true when your module has a very high number of sources and/or dependent static or shared libraries. Doing so forces the build system to use @ syntax for archives containing intermediate object files or linking libraries.

This feature can be useful on Windows, where the command line accepts a maximum of only of 8191 characters, which can be too small for complex projects. It also impacts the compilation of individual source files, placing nearly all compiler flags inside list files, too.

Note that any value other than true will revert to the default behaviour. You can also define APP_SHORT_COMMANDS in your Application.mk (https://developer.android.com/ndk/guides/application_mk.html) file to force this behavior for all modules in your project.

We do not recommend enabling this feature by default, since it makes the build slower.

LOCAL_THIN_ARCHIVE

Set this variable to true when building static libraries. Doing so will generate a **thin archive**, a library file that does not contain object files, but instead just file paths to the actual objects that it would normally contain.

This is useful to reduce the size of your build output. The drawback is that such libraries *cannot* be moved to a different location (all paths inside them are relative).

Valid values are `true`, `false` or empty. A default value can be set in your `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file through the `APP_THIN_ARCHIVE` variable.

> **Note:** This is ignored for non-static library modules, or prebuilt static library ones.

LOCAL_FILTER_ASM

Define this variable as a shell command that the build system will use to filter the assembly files extracted or generated from the files you specified for `LOCAL_SRC_FILES`.

Defining this variable causes the following things to occur:

1. The build system generates a temporary assembly file from any C or C++ source file, instead of compiling them into an object file.
2. The build system executes the shell command in `LOCAL_FILTER_ASM` on any temporary assembly file and on any assembly file listed in `LOCAL_SRC_FILES`, thus generating another temporary assembly file.
3. The build system compiles these filtered assembly files into an object file.

For example:

```
LOCAL_SRC_FILES  := foo.c bar.S
LOCAL_FILTER_ASM :=

foo.c --1--> $OBJS_DIR/foo.S.original --2--> $OBJS_DIR/foo.S --3--> $OBJS_DIR/foo
bar.S                                 --2--> $OBJS_DIR/bar.S --3--> $OBJS_DIR/bar
```

"1" corresponds to the compiler, "2" to the filter, and "3" to the assembler. The filter must be a standalone shell command that takes the name of the input file as its first argument, and the name of the output file as the second one. For example:

```
myasmfilter $OBJS_DIR/foo.S.original $OBJS_DIR/foo.S
myasmfilter bar.S $OBJS_DIR/bar.S
```

NDK-provided function macros

This section explains GNU Make function macros that the NDK provides. Use `$(call <function>)` to evaluate them; they return textual information.

my-dir

This macro returns the path of the last included makefile, which typically is the current `Android.mk`'s directory. `my-dir` is useful for defining `LOCAL_PATH` at the start of your `Android.mk` file. For example:

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

Due to the way GNU Make works, what this macro really returns is the path of the last makefile that the build system included when parsing the build scripts. For this reason, you should not call `my-dir` after including another file.

For example, consider the following example:

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

# ... declare one module

include $(LOCAL_PATH)/foo/`Android.mk`

LOCAL_PATH := $(call my-dir)

# ... declare another module
```

The problem here is that the second call to `my-dir` defines `LOCAL_PATH` as `$PATH/foo` instead of `$PATH`, because that was where its most recent include pointed.

You can avoid this problem by putting additional includes after everything else in the `Android.mk` file. For example:

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

# ... declare one module

LOCAL_PATH := $(call my-dir)

# ... declare another module

# extra includes at the end of the Android.mk file
include $(LOCAL_PATH)/foo/Android.mk
```

If it is not feasible to structure the file in this way, save the value of the first `my-dir` call into another variable. For example:

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

LOCAL_PATH := $(MY_LOCAL_PATH)

# ... declare one module

include $(LOCAL_PATH)/foo/`Android.mk`

LOCAL_PATH := $(MY_LOCAL_PATH)

# ... declare another module
```

all-subdir-makefiles

Returns the list of `Android.mk` files located in all subdirectories of the current `my-dir` path.

You can use this function to provide deep-nested source directory hierarchies to the build system. By default, the NDK only looks for files in the directory containing the `Android.mk` file.

this-makefile

Returns the path of the current makefile (from which the build system called the function).

parent-makefile

Returns the path of the parent makefile in the inclusion tree (the path of the makefile that included the current one).

grand-parent-makefile

Returns the path of the grandparent makefile in the inclusion tree (the path of the makefile that included the current one).

import-module

A function that allows you to find and include a module's `Android.mk` file by the name of the module. A typical example is as follows:

```
$(call import-module,<name>)
```

In this example, the build system looks for the module tagged `<name>` in the list of directories referenced that your `NDK_MODULE_PATH` environment variable references, and includes its `Android.mk` file automatically for you.