**Concept**

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,

You can use the same source file in several modules.

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.

**Example**

---------- cut here ------------------

#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).

#!!! **A module should start with LOCAL\_PATH**

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

#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.

# with the exception of LOCAL\_PATH

include $(CLEAR\_VARS)

#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'.

LOCAL\_MODULE := hello-jni

#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.

# Default extention for C++ is “.cpp” You can change it by setting

# LOCAL\_CPP\_EXTENSION=”.cxx”

LOCAL\_SRC\_FILES := hello-jni.c

#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.  
 include $(BUILD\_SHARED\_LIBRARY)  
----------- cut here ------------------

**Naming convention**

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 ------------------

**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\_WHOLE\_STATIC\_LIBRARIES described below).  
 Example usage:  
  
 include $(BUILD\_STATIC\_LIBRARY)  
  
 Note that this will generate a file named lib$(LOCAL\_MODULE).a  
  
**PREBUILT\_SHARED\_LIBRARY**  
 Points to a build script used to specify a prebuilt shared library.  
 Unlike BUILD\_SHARED\_LIBRARY and BUILD\_STATIC\_LIBRARY, the value  
 of LOCAL\_SRC\_FILES must be a single path to a prebuilt shared  
 library (e.g. foo/libfoo.so), instead of a source file.  
  
 You can reference the prebuilt library in another module using  
 the LOCAL\_PREBUILTS variable (see docs/PREBUILTS.html for more  
 information).  
  
**PREBUILT\_STATIC\_LIBRARY**  
 This is the same as PREBUILT\_SHARED\_LIBRARY, but for a static library  
 file instead. See docs/PREBUILTS.html for more.  
  
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 example, 'android-3' correspond to Android 1.5 system images. For  
 a complete list of platform names and corresponding Android system  
 images, read docs/STABLE-APIS.html.  
  
TARGET\_ARCH\_ABI  
 Name of the target CPU+ABI when this Android.mk is parsed.  
 Two values are supported at the moment:  
  
 armeabi  
 For ARMv5TE  
  
 armeabi-v7a  
  
 NOTE: Up to Android NDK 1.6\_r1, this variable was simply defined  
 as 'arm'. However, the value has been redefined to better  
 match what is used internally by the Android platform.  
  
 For more details about architecture ABIs and corresponding  
 compatibility issues, please read docs/CPU-ARCH-ABIS.html  
  
 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-armeabi'  
  
 (Up to Android NDK 1.6\_r1, this used to be 'android-3-arm' by default)

**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 **last included Makefile**, which typically is  
 the current Android.mk's directory. This is useful to define  
 LOCAL\_PATH at the start of your Android.mk as with:  
  
 LOCAL\_PATH := $(call my-dir)  
  
 IMPORTANT NOTE: Due to the way GNU Make works, this really returns  
 the path of the \*last\* \*included\* \*Makefile\* during the parsing of  
 build scripts. **Do 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' will define  
 LOCAL\_PATH to $PATH/foo instead of $PATH, due to the include that  
 was performed before that.  
  
 For this reason, it's better to put additional includes after  
 everything else in an Android.mk, as in:  
  
 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  
 include $(LOCAL\_PATH)/foo/Android.mk  
  
 If this is not convenient, 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 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...  
  
import-module  
 A function that allows you to find and include the Android.mk  
 of another module by name. A typical example is:  
  
 $(call import-module,<name>)  
  
 And this will look for the module tagged <name> in the list of  
 directories referenced by your **NDK\_MODULE\_PATH** environment  
 variable, and include its Android.mk automatically for you.  
  
 Read docs/IMPORT-MODULE.html for more details.

**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 explicitly  
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.  
  
 By default, 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)  
  
 You can override this default with LOCAL\_MODULE\_FILENAME (see below)  
  
LOCAL\_MODULE\_FILENAME  
 This variable is optional, and allows you to redefine the name of  
 generated files. By default, module <foo> will always generate a  
 static library named lib<foo>.a or a shared library named lib<foo>.so,  
 which are standard Unix conventions.  
  
 You can override this by defining LOCAL\_MODULE\_FILENAME, For example:  
  
 LOCAL\_MODULE := foo-version-1  
 LOCAL\_MODULE\_FILENAME := libfoo  
  
 NOTE: You should not put a path or file extension in your  
 LOCAL\_MODULE\_FILENAME, these will be handled automatically by the  
 build system.  
  
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(s) of C++ source files. They must begin with a dot.  
 The default is '.cpp' but you can change it. For example:  
  
 LOCAL\_CPP\_EXTENSION := .cxx  
  
 Since NDK r7, you can list several extensions in this variable, as in:  
  
 LOCAL\_CPP\_EXTENSION := .cxx .cpp .cc  
  
**LOCAL\_CPP\_FEATURES**  
 This is an optional variable that can be defined to indicate  
 that your code relies on specific C++ features. To indicate that  
 your code uses RTTI (RunTime Type Information), use the following:  
  
 LOCAL\_CPP\_FEATURES := rtti  
  
 To indicate that your code uses C++ exceptions, use:  
  
 LOCAL\_CPP\_FEATURES := exceptions  
  
 You can also use both of them with (order is not important):  
  
 LOCAL\_CPP\_FEATURES := rtti features  
  
 The effect of this variable is to enable the right compiler/linker  
 flags when building your modules from sources. For prebuilt binaries,  
 this also helps declare which features the binary relies on to ensure  
 the final link works correctly.  
  
 It is recommended to use this variable instead of enabling -frtti and  
 -fexceptions directly in your LOCAL\_CPPFLAGS definition.  
  
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.

!! --std=c++0x

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.  
  
LOCAL\_ARM\_NEON  
 Defining this variable to 'true' allows the use of ARM Advanced SIMD  
 (a.k.a. NEON) GCC intrinsics in your C and C++ sources, as well as  
 NEON instructions in Assembly files.  
  
 You should only define it when targeting the 'armeabi-v7a' ABI that  
 corresponds to the ARMv7 instruction set. Note that not all ARMv7  
 based CPUs support the NEON instruction set extensions and that you  
 should perform runtime detection to be able to use this code at runtime  
 safely. To learn more about this, please read the documentation at  
 docs/CPU-ARM-NEON.html and docs/CPU-FEATURES.html.  
  
 Alternatively, you can also specify that only specific source files  
 may be compiled with NEON support by using the '.neon' suffix, as  
 in:  
  
 LOCAL\_SRC\_FILES = foo.c.neon bar.c zoo.c.arm.neon  
  
 In this example, 'foo.c' will be compiled in thumb+neon mode,  
 'bar.c' will be compiled in 'thumb' mode, and 'zoo.c' will be  
 compiled in 'arm+neon' mode.  
  
 Note that the '.neon' suffix must appear after the '.arm' suffix  
 if you use both (i.e. foo.c.arm.neon works, but not foo.c.neon.arm !)  
  
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 if you \*really\*  
 need to by setting this variable to 'true'.  
  
 NOTE: This feature does not modify the ABI and is only enabled on  
 kernels targeting ARMv6+ CPU devices. Machine code generated  
 with this feature enabled will run unmodified on devices  
 running earlier CPU architectures.  
  
 For more information, see:  
  
 http://en.wikipedia.org/wiki/NX\_bit  
 http://www.gentoo.org/proj/en/hardened/gnu-stack.xml  
  
LOCAL\_EXPORT\_CFLAGS  
 Define this variable to record a set of C/C++ compiler flags that will  
 be added to the LOCAL\_CFLAGS definition of any other module that uses  
 this one with LOCAL\_STATIC\_LIBRARIES or LOCAL\_SHARED\_LIBRARIES.  
  
 For example, consider the module 'foo' with the following definition:  
  
 include $(CLEAR\_VARS)  
 LOCAL\_MODULE := foo  
 LOCAL\_SRC\_FILES := foo/foo.c  
 **LOCAL\_EXPORT\_CFLAGS** := -DFOO=1  
 include $(BUILD\_STATIC\_LIBRARY)  
  
 And another module, named 'bar' that depends on it as:  
  
 include $(CLEAR\_VARS)  
 LOCAL\_MODULE := bar  
 LOCAL\_SRC\_FILES := bar.c  
 LOCAL\_CFLAGS := -DBAR=2  
 LOCAL\_STATIC\_LIBRARIES := foo  
 include $(BUILD\_SHARED\_LIBRARY)  
  
 Then, the flags '-DFOO=1 -DBAR=2' will be passed to the compiler when  
 building bar.c  
  
 Exported flags are prepended to your module's LOCAL\_CFLAGS so you can  
 easily override them. They are also transitive: if 'zoo' depends on  
 'bar' which depends on 'foo', then 'zoo' will also inherit all flags  
 exported by 'foo'.  
  
 Finally, exported flags are \*not\* used when building the module that  
 exports them. In the above example, -DFOO=1 would not be passed to the  
 compiler when building foo/foo.c.  
  
LOCAL\_EXPORT\_CPPFLAGS  
 Same as LOCAL\_EXPORT\_CFLAGS, but for C++ flags only.  
  
LOCAL\_EXPORT\_C\_INCLUDES  
 Same as LOCAL\_EXPORT\_CFLAGS, but for C include paths.  
 This can be useful if 'bar.c' wants to include headers  
 that are provided by module 'foo'.  
  
LOCAL\_EXPORT\_LDLIBS  
 Same as LOCAL\_EXPORT\_CFLAGS, but for linker flags. Note that the  
 imported linker flags will be appended to your module's LOCAL\_LDLIBS  
 though, due to the way Unix linkers work.  
  
 This is typically useful when module 'foo' is a static library and has  
 code that depends on a system library. LOCAL\_EXPORT\_LDLIBS can then be  
 used 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)  
  
 There, libbar.so will be built with a -llog at the end of the linker  
 command to indicate that it depends on the system logging library,  
 because it depends on 'foo'.  
  
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. This forces the  
 build system to use an intermediate list file, and use it with the  
 library archiver or static linker with the @$(listfile) syntax.  
  
 This can be useful on Windows, where the command-line only accepts  
 a maximum of 8191 characters, which can be too small for complex  
 projects.  
  
 This also impacts the compilation of individual source files, placing  
 nearly all compiler flags inside list files too.  
  
 Note that any other value than 'true' will revert to the default  
 behaviour. You can also define APP\_SHORT\_COMMANDS in your  
 Application.mk to force this behaviour for all modules in your  
 project.  
  
 NOTE: We do not recommend enabling this feature by default, since it  
 makes the build slower.  
  
LOCAL\_FILTER\_ASM  
 Define this variable to a shell command that will be used to filter  
 the assembly files from, or generated from, your LOCAL\_SRC\_FILES.  
  
 When it is defined, the following happens:  
  
 - Any C or C++ source file is generated into a temporary assembly  
 file (instead of being compiled into an object file).  
  
 - Any temporary assembly file, and any assembly file listed in  
 LOCAL\_SRC\_FILES is sent through the LOCAL\_FILTER\_ASM command  
 to generate \_another\_ temporary assembly file.  
  
 - These filtered assembly files are compiled into object file.  
  
 In other words, If you have:  
  
 LOCAL\_SRC\_FILES := foo.c bar.S  
 LOCAL\_FILTER\_ASM := myasmfilter  
  
 foo.c --1--> $OBJS\_DIR/foo.S.original --2--> $OBJS\_DIR/foo.S --3--> $OBJS\_DIR/foo.o  
 bar.S --2--> $OBJS\_DIR/bar.S --3--> $OBJS\_DIR/bar.o  
  
 Were "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, as in:  
  
 myasmfilter $OBJS\_DIR/foo.S.original $OBJS\_DIR/foo.S  
 myasmfilter bar.S $OBJS\_DIR/bar.S