

ABI Management

On this page

Supported ABIs

Generating Code for a Specific ABI


ABI Management on the Android Platform

Different Android handsets use different CPUs, which in turn support different instruction sets. Each combination of CPU and instruction sets has its own Application Binary Interface, or *ABI*. The ABI defines, with great precision, how an application's machine code is supposed to interact with the system at runtime. You must specify an ABI for each CPU architecture you want your app to work with.

A typical ABI includes the following information:

- The CPU instruction set(s) that the machine code should use.
- The endianness of memory stores and loads at runtime.
- The format of executable binaries, such as programs and shared libraries, and the types of content they support.
- Various conventions for passing data between your code and the system. These conventions include alignment constraints, as well as how the system uses the stack and registers when it calls functions.
- The list of function symbols available to your machine code at runtime, generally from very specific sets of libraries.

This page enumerates the ABIs that the NDK supports, and provides information about how each ABI works. For a list of ABI issues for 32-bit systems, see [32-bit ABI bugs](#)

(https://android.googlesource.com/platform/bionic/+/master/README.md#32_bit-ABI-bugs) 

Supported ABIs

Each ABI supports one or more instruction sets. Table 1 provides an at-a-glance overview of the instruction sets each ABI supports.

Table 1. ABIs and supported instruction sets.

ABI	Supported Instruction Set(s)	Notes
armeabi (#armeabi)	<ul style="list-style-type: none">• ARMV5TE and later• Thumb-1	No hard float.
armeabi-v7a (#v7a)	<ul style="list-style-type: none">• armeabi• Thumb-2• VFPv3-D16• Other, optional	Incompatible with ARMv5, v6 devices.
arm64-v8a (#arm64-v8a)	<ul style="list-style-type: none">• AArch-64	
x86 (#x86)	<ul style="list-style-type: none">• x86 (IA-32)• MMX• SSE/2/3• SSSE3	No support for MOVBE or SSE4.
x86_64 (#86-64)	<ul style="list-style-type: none">• x86-64• MMX• SSE/2/3• SSSE3• SSE4.1, 4.2• POPCNT	
mips (#mips)	<ul style="list-style-type: none">• MIPS32r1 and later	Uses hard-float, and assumes a CPU:FPU clock ratio of 2:1 for maximum compatibility. Provides neither micromips nor MIPS16.
mips64 (#mips64)	<ul style="list-style-type: none">• MIPS64r6	

More detailed information about each ABI appears below.

armeabi

This ABI is for ARM-based CPUs that support at least the ARMv5TE instruction set. Please refer to the following documentation for more details:

- ARM Architecture Reference Manual (https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf)
- Procedure Call Standard for the ARM Architecture (http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042e/IHL0042E_aapcs.pdf)
- ARM ELF File Format (http://infocenter.arm.com/help/topic/com.arm.doc.dui0101a/DUI0101A_Elf.pdf)
- Application Binary Interface (ABI) for the ARM Architecture (<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html>)
- Base Platform ABI for the ARM Architecture (http://infocenter.arm.com/help/topic/com.arm.doc.ihl0037c/IHL0037C_bpabi.pdf)
- C Library ABI for the ARM Architecture (http://infocenter.arm.com/help/topic/com.arm.doc.ihl0039c/IHL0039C_clibabi.pdf)
- C++ ABI for the ARM Architecture (<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihl0041d/index.html>)
- Run-time ABI for the ARM Architecture (http://infocenter.arm.com/help/topic/com.arm.doc.ihl0043d/IHL0043D_rtabi.pdf)
- ELF System V Application Binary Interface (<http://www.sco.com/developers/gabi/2001-04-24/contents.html>)
- Generic/Itanium C++ ABI (<http://mentorembedded.github.com/cxx-abi/abi.html>)

The AAPCS standard defines EABI as a family of similar but distinct ABIs. Also, Android follows the little-endian ARM GNU/Linux ABI (http://sourcery.mentor.com/sgpp/lite/arm/portal/kbattach142/arm_gnu_linux_abi.pdf).

This ABI does not support hardware-assisted floating point computations. Instead, all floating-point operations use software helper functions from the compiler's `libgcc.a` static library.

The armeabi ABI supports ARM's Thumb (a.k.a. Thumb-1) instruction set (<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>). The NDK generates Thumb code by default unless you specify different behavior using the `LOCAL_ARM_MODE` variable in your `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) file.

armeabi-v7a

This ABI extends armeabi to include several CPU instruction set extensions (<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html>). The instruction extensions that this Android-specific ABI supports are:

- The Thumb-2 instruction set extension, which provides performance comparable to 32-bit ARM instructions

with similar compactness to Thumb-1.

- The VFP hardware-FPU instructions. More specifically, VFPv3-D16, which includes 16 dedicated 64-bit floating point registers, in addition to another 16 32-bit registers from the ARM core.

Other extensions that the v7-a ARM spec describes, including Advanced SIMD (<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Beijfcja.html>) (a.k.a. NEON), VFPv3-D32, and ThumbEE, are optional to this ABI. Since their presence is not guaranteed, the system should check at runtime whether the extensions are available. If they are not, you must use alternative code paths. This check is similar to the one that the system typically performs to check or use MMX (http://en.wikipedia.org/wiki/MMX_%28instruction_set%29), SSE2 (<http://en.wikipedia.org/wiki/SSE2>), and other specialized instruction sets on x86 CPUs.

For information about how to perform these runtime checks, refer to The `cpufeatures` Library (<https://developer.android.com/ndk/guides/cpu-features.html>). Also, for information about the NDK's support for building machine code for NEON, see NEON Support (<https://developer.android.com/ndk/guides/cpu-arm-neon.html>).

The `armeabi-v7a` ABI uses the `-mfloat-abi=softfp` switch to enforce the rule that the compiler must pass all double values in core register pairs during function calls, instead of dedicated floating-point ones. The system can perform all internal computations using the FP registers. Doing so speeds up the computations greatly.

arm64-v8a

This ABI is for ARMv8-based CPUs that support AArch64. It also includes the NEON and VFPv4 instruction sets.

For more information, see the ARMv8 Technology Preview (http://www.arm.com/files/downloads/ARMv8_Architecture.pdf), and contact ARM for further details.

x86

This ABI is for CPUs supporting the instruction set commonly referred to as "x86" or "IA-32". Characteristics of this ABI include:

- Instructions normally generated by GCC with compiler flags such as the following:

```
-march=i686 -mtune=intel -mssse3 -mfpmath=sse -m32
```

These flags target the the Pentium Pro instruction set, along with the the MMX

(http://en.wikipedia.org/wiki/MMX_%28instruction_set%29), SSE (http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), SSE2 (<http://en.wikipedia.org/wiki/SSE2>), SSE3 (<http://en.wikipedia.org/wiki/SSE3>), and SSSE3 (<http://en.wikipedia.org/wiki/SSSE3>)

instruction set extensions. The generated code is an optimization balanced across the top Intel 32-bit CPUs.

For more information on compiler flags, particularly related to performance optimization, refer to GCC x86 performance hints (<http://software.intel.com/blogs/2012/09/26/gcc-x86-performance-hints>).

- Use of the standard Linux x86 32-bit calling convention, as opposed to the one for SVR. For more information,

see section 6, "Register Usage", of Calling conventions for different C++ compilers and operating systems (http://www.agner.org/optimize/calling_conventions.pdf).

The ABI does not include any other optional IA-32 instruction set extensions, such as:

- MOVBE
- Any variant of SSE4.

You can still use these extensions, as long as you use runtime feature-probing to enable them, and provide fallbacks for devices that do not support them.

The NDK toolchain assumes 16-byte stack alignment before a function call. The default tools and options enforce this rule. If you are writing assembly code, you must make sure to maintain stack alignment, and ensure that other compilers also obey this rule.

Refer to the following documents for more details:

- GCC online documentation: Intel 386 and AMD x86-64 Options (<https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/i386-and-x86-64-Options.html>)
- Calling conventions for different C++ compilers and operating systems (http://www.agner.org/optimize/calling_conventions.pdf)
- Intel IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>)
- Intel IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide (<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf>)
- System V Application Binary Interface: Intel386 Processor Architecture Supplement (<http://www.sco.com/developers/devspecs/abi386-4.pdf>)

x86_64

This ABI is for CPUs supporting the instruction set commonly referred to as "x86-64." It supports instructions that GCC typically generates with the following compiler flags:

```
-march=x86-64 -msse4.2 -mpopcnt -m64 -mtune=intel
```

These flags target the x86-64 instruction set, according to the GCC documentation. along with the MMX (http://en.wikipedia.org/wiki/MMX_%28instruction_set%29), SSE (http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), SSE2 (<http://en.wikipedia.org/wiki/SSE2>), SSE3 (<http://en.wikipedia.org/wiki/SSE3>), SSSE3 (<http://en.wikipedia.org/wiki/SSSE3>), SSE4.1

(<http://en.wikipedia.org/wiki/SSE4#SSE4.1>), SSE4.2 (<http://en.wikipedia.org/wiki/SSE4#SSE4.2>), and POPCNT (<https://software.intel.com/en-us/node/512035>) instruction-set extensions. The generated code is an optimization balanced across the top Intel 64-bit CPUs.

For more information on compiler flags, particularly related to performance optimization, refer to GCC x86 Performance (<http://software.intel.com/blogs/2012/09/26/gcc-x86-performance-hints>).

This ABI does not include any other optional x86-64 instruction set extensions, such as:

- MOVBE
- SHA
- AVX
- AVX2

You can still use these extensions, as long as you use runtime feature probing to enable them, and provide fallbacks for devices that do not support them.

Refer to the following documents for more details:

- Calling conventions for different C++ compilers and operating systems
(http://www.agner.org/optimize/calling_conventions.pdf)
- Intel64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference
(http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals)
- Intel64 and IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming
(http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals)

mips

This ABI is for MIPS-based CPUs that support at least the MIPS32r1 instruction set. It includes the following features:

- MIPS32 revision 1 ISA
- Little-endian
- O32
- Hard-float
- No DSP application-specific extensions

For more information, please refer to the following documentation:

- Architecture for Programmers ("MIPSARCH")

- [ELF System V Application Binary Interface \(https://refspecs.linuxbase.org/elf/gabi4+/contents.html\)](https://refspecs.linuxbase.org/elf/gabi4+/contents.html)

- [Itanium/Generic C++ ABI \(http://sourcery.mentor.com/public/cxx-abi/abi.html\)](http://sourcery.mentor.com/public/cxx-abi/abi.html)

For more specific details, see [MIPS32 Architecture \(http://www.imgtec.com/mips/architectures/mips32/\)](http://www.imgtec.com/mips/architectures/mips32/). Answers to common questions are in the [MIPS FAQ \(https://sourcery.mentor.com/sgpp/lite/mips/portal/target_arch?@action=faq&target_arch=MIPS\)](https://sourcery.mentor.com/sgpp/lite/mips/portal/target_arch?@action=faq&target_arch=MIPS).

mips64

This ABI is for MIPS64 R6. For more information, see [MIPS64 Architecture \(http://www.imgtec.com/mips/architectures/mips64/\)](http://www.imgtec.com/mips/architectures/mips64/).

Generating Code for a Specific ABI

By default, the NDK generates machine code for the armeabi ABI. You can generate ARMv7-a-compatible machine code, instead, by adding the following line to your `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file.

```
APP_ABI := armeabi-v7a
```

To build machine code for two or more distinct ABIs, using spaces as delimiters. For example:

```
APP_ABI := armeabi armeabi-v7a
```

This setting tells the NDK to build two versions of your machine code: one for each ABI listed on this line. For more information on the values you can specify for the `APP_ABI` variable, see [Android.mk \(https://developer.android.com/ndk/guides/android_mk.html\)](https://developer.android.com/ndk/guides/android_mk.html).

When you build multiple machine-code versions, the build system copies the libraries to your application project path, and ultimately packages them into your APK, so creating a *fat binary* (http://en.wikipedia.org/wiki/Fat_binary). A fat binary is larger than one containing only the machine code for a single system; the tradeoff is gaining wider compatibility, but at the expense of a larger APK.

At installation time, the package manager unpacks only the most appropriate machine code for the target device. For details, see [Automatic extraction of native code at install time \(#aen\)](#).

ABI Management on the Android Platform

This section provides details about how the Android platform manages native code in APKs.

Native code in app packages

Both the Play Store and Package Manager expect to find NDK-generated libraries on filepaths inside the APK matching the following pattern:

```
/lib/<abi>/lib<name>.so
```

Here, `<abi>` is one of the ABI names listed under Supported ABIs (#sa), and `<name>` is the name of the library as you defined it for the `LOCAL_MODULE` variable in the `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) file. Since APK files are just zip files, it is trivial to open them and confirm that the shared native libraries are where they belong.

If the system does not find the native shared libraries where it expects them, it cannot use them. In such a case, the app itself has to copy the libraries over, and then perform `dlopen()`.

In a fat binary, each library resides under a directory whose name matches a corresponding ABI. For example, a fat binary may contain:

```
/lib/armeabi/libfoo.so
/lib/armeabi-v7a/libfoo.so
/lib/arm64-v8a/libfoo.so
/lib/x86/libfoo.so
/lib/x86_64/libfoo.so
/lib/mips/libfoo.so
/lib/mips64/libfoo.so
```

Note: ARMv7-based Android devices running 4.0.3 or earlier install native libraries from the `armeabi` directory instead of the `armeabi-v7a` directory if both directories exist. This is because `/lib/armeabi/` comes after `/lib/armeabi-v7a/` in the APK. This issue is fixed from 4.0.4.

Android Platform ABI support

The Android system knows at runtime which ABI(s) it supports, because build-specific system properties indicate:

- The primary ABI for the device, corresponding to the machine code used in the system image itself.
- An optional, secondary ABI, corresponding to another ABI that the system image also supports.

This mechanism ensures that the system extracts the best machine code from the package at installation time.

For best performance, you should compile directly for the primary ABI. For example, a typical ARMv5TE-based

device would only define the primary ABI: `armeabi`. By contrast, a typical, ARMv7-based device would define the primary ABI as `armeabi-v7a` and the secondary one as `armeabi`, since it can run application native binaries generated for each of them.

Many x86-based devices can also run `armeabi-v7a` and `armeabi` NDK binaries. For such devices, the primary ABI would be `x86`, and the second one, `armeabi-v7a`.

A typical MIPS-based device only defines a primary abi: `mips`.

Automatic extraction of native code at install time

When installing an application, the package manager service scans the APK, and looks for any shared libraries of the form:

```
lib/<primary-abi>/lib<name>.so
```

If none is found, and you have defined a secondary ABI, the service scans for shared libraries of the form:

```
lib/<secondary-abi>/lib<name>.so
```

When it finds the libraries that it's looking for, the package manager copies them to `/lib/lib<name>.so`, under the application's `data` directory (`data/data/<package_name>/lib/`).

If there is no shared-object file at all, the application builds and installs, but crashes at runtime.

