Android ABIs

Different Android devices use different CPUs, which in turn support different instruction sets. Each combination of CPU and instruction set has its own Application Binary Interface (ABI). An ABI includes the following information:

- The CPU instruction set (and extensions) that can be used.
- The endianness of memory stores and loads at runtime. Android is always little-endian.
- Conventions for passing data between applications and the system, including alignment constraints, and how the system uses the stack and registers when it calls functions.
- The format of executable binaries, such as programs and shared libraries, and the types of content they support. Android always uses ELF. For more information, see ELF System V Application Binary Interface (https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html).
- How C++ names are mangled. For more information, see Generic/Itanium C++ ABI (http://itanium-cxx-abi.github.io/cxx-abi/).

This page enumerates the ABIs that the NDK supports, and provides information about how each ABI works.

ABI can also refer to the native API supported by the platform. For a list of those kinds of ABI issues affecting 32-bit systems, see 32-bit ABI bugs (https://android.googlesource.com/platform/bionic/+/master/docs/32-bit-abi.md).

Supported ABIs

Table 1. ABIs and supported instruction sets.

ABI	Supported Instruction Sets	Notes
`armeabi-v7a` (#v7a)	• armeabi • Thumb-2	Incompatible with ARMv5/v6 devices.

	• VFPv3-D16	
arm64-v8a (#arm64-v8a)	• AArch64	
x86 (#x86)	• x86 (IA-32) • MMX • SSE/2/3 • SSSE3	No support for MOVBE or SSE4.
x86_64 (#86-64)	• x86-64 • MMX • SSE/2/3 • SSSE3 • SSE4.1, 4.2 • POPCNT	

Note: Historically the NDK supported ARMv5 (armeabi), and 32-bit and 64-bit MIPS, but support for these ABIs was removed in NDK r17.

armeabi-v7a

This ABI is for 32-bit ARM-based CPUs. The Android variant includes Thumb-2 and the VFP hardware floating point instructions, specifically VFPv3-D16, which includes 16 dedicated 64-bit floating point registers.

For information about the parts of the ABI that aren't Android-specific, see Application Binary Interface (ABI) for the ARM Architecture (https://developer.arm.com/architectures/system-architectures/software-standards/abi)

The NDK's build systems generate Thumb-2 code by default unless you use `LOCAL_ARM_MODE` in your `Android.mk` (/ndk/guides/android_mk) for ndk-build or `ANDROID_ARM_MODE` when configuring CMake (/ndk/guides/cmake).

Other extensions including Advanced SIMD (Neon) and VFPv3-D32 are optional. For more information, see Neon Support (/ndk/guides/cpu-arm-neon).

The `armeabi-v7a` ABI uses `-mfloat-abi=softfp` to enforce the rule that, although system can execute floating-point code, the compiler must pass all `float` values in integer registers and all `double` values in integer register pairs when making function calls.

arm64-v8a

This ABI is for ARMv8-A based CPUs, which support the 64-bit AArch64 architecture. It includes the Advanced SIMD (Neon) architecture extensions.

You can use Neon intrinsics (https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics) in C and C++ code to take advantage of the Advanced SIMD extension. The Neon Programmer's Guide for Armv8-A (https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/neon-programmers-guide-for-armv8-a) provides more information about Neon intrinsics and Neon programming in general.

See Arm's Learn the Architecture (https://developer.arm.com/architectures/learn-the-architecture) for complete details of the parts of the ABI that aren't Android-specific. Arm also offers some porting advice in 64-bit Android Development (https://developer.arm.com/64bit).

On Android, the platform-specific x18 register is reserved for ShadowCallStack (https://source.android.com/devices/tech/debug/shadow-call-stack) and should not be touched by your code. Current versions of Clang default to using the `-ffixed-x18` option on Android, so unless you have hand-written assembler (or a very old compiler) you shouldn't need to worry about this.

x86

This ABI is for CPUs supporting the instruction set commonly known as "x86", "i386", 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 Pentium Pro instruction set, along with the the MMX (https://en.wikipedia.org/wiki/MMX_%28instruction_set%29), SSE (https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), SSE2 (https://en.wikipedia.org/wiki/SSE2), SSE3 (https://en.wikipedia.org/wiki/SSE3), and SSSE3 (https://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 (https://software.intel.com/content/www/us/en/develop/blogs/gcc-x86-performance-hints.html)
  .

- 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:

- 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 (https://en.wikipedia.org/wiki/MMX_%28instruction_set%29), SSE (https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), SSE2 (https://en.wikipedia.org/wiki/SSE2), SSE3 (https://en.wikipedia.org/wiki/SSE3), SSSE3 (https://en.wikipedia.org/wiki/SSSE3), SSE4.1 (https://en.wikipedia.org/wiki/SSE4#SSE4.1), SSE4.2 (https://en.wikipedia.org/wiki/SSE4#SSE4.2), and POPCNT 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 Hints (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)

Generate code for a specific ABI

Gradle (#gradle) ndk-build (#ndk-build) CMake (#cmake)

Gradle (whether used via Android Studio or from the command line) builds for all non-deprecated ABIs by default. To restrict the set of ABIs that your application supports, use `abiFilters`. For example, to build for only 64-bit ABIs, set the following configuration in your `build.gradle`:

```
android {
    defaultConfig {
        ndk {
            abiFilters 'arm64-v8a', 'x86_64'
        }
    }
}
```

The default behavior of the build system is to include the binaries for each ABI in a single APK, also known as a fat APK (https://en.wikipedia.org/wiki/Fat_binary). A fat APK is significantly larger than one containing only the binaries for a single ABI; the tradeoff is gaining wider compatibility, but at the expense of a larger APK. It is strongly recommended that you take advantage of either App Bundles (/guide/app-bundle) or APK Splits (/studio/build/configure-apk-splits) to reduce the size of your APKs while still maintaining maximum device compatibility.

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` (/ndk/guides/android_mk) 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 APK, each library resides under a directory whose name matches a corresponding ABI. For example, a fat APK 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
```

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.
- Optionally, secondary ABIs, corresponding to other 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.

64-bit devices also support their 32-bit variants. Using arm64-v8a devices as an example, the device can also run armeabi and armeabi-v7a code. Note, however, that your application will perform much better on 64-bit devices if it targets arm64-v8a rather than relying on the device running the armeabi-v7a version of your application.

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`.

You can force install an apk for a specific ABI (#sa). This is useful for testing. Use the following command:

```
adb install --abi abi-identifier path_to_apk
```

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 native library directory (`<nativeLibraryDir>/`). The following snippets retrieve the `nativeLibraryDir`:

Kotlin (#kotlin) Java (#java)

```
import android.content.pm.PackageInfo
import android.content.pm.ApplicationInfo
import android.content.pm.PackageManager
...
val ainfo = this.applicationContext.packageManager.getApplication
        "com.domain.app",
        PackageManager.GET_SHARED_LIBRARY_FILES
)
Log.v(TAG, "native library dir ${ainfo.nativeLibraryDir}")
```

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