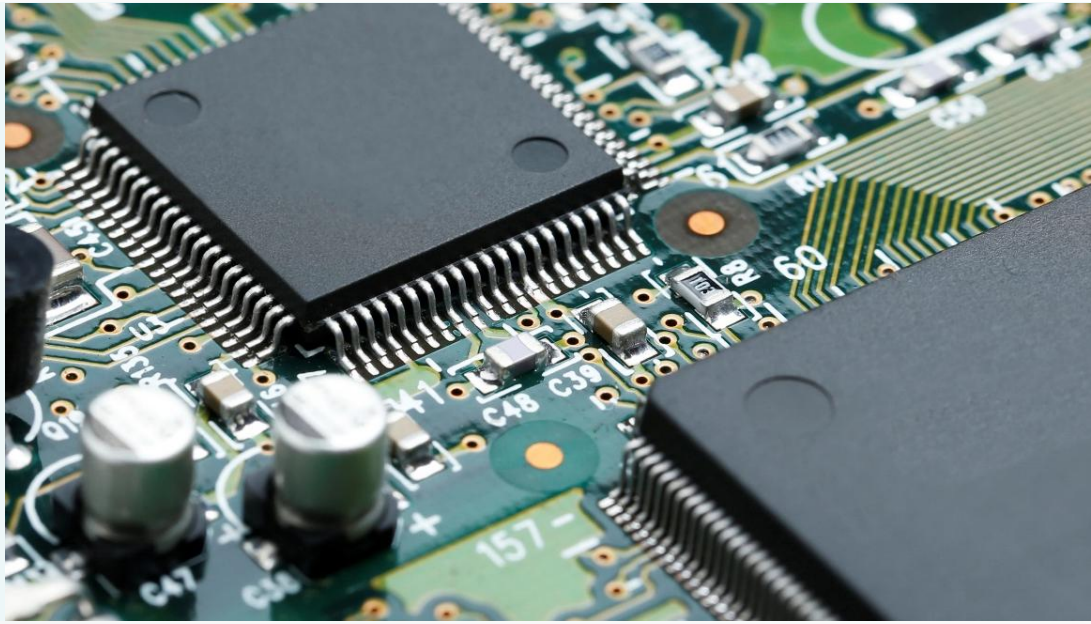
Cursul #8

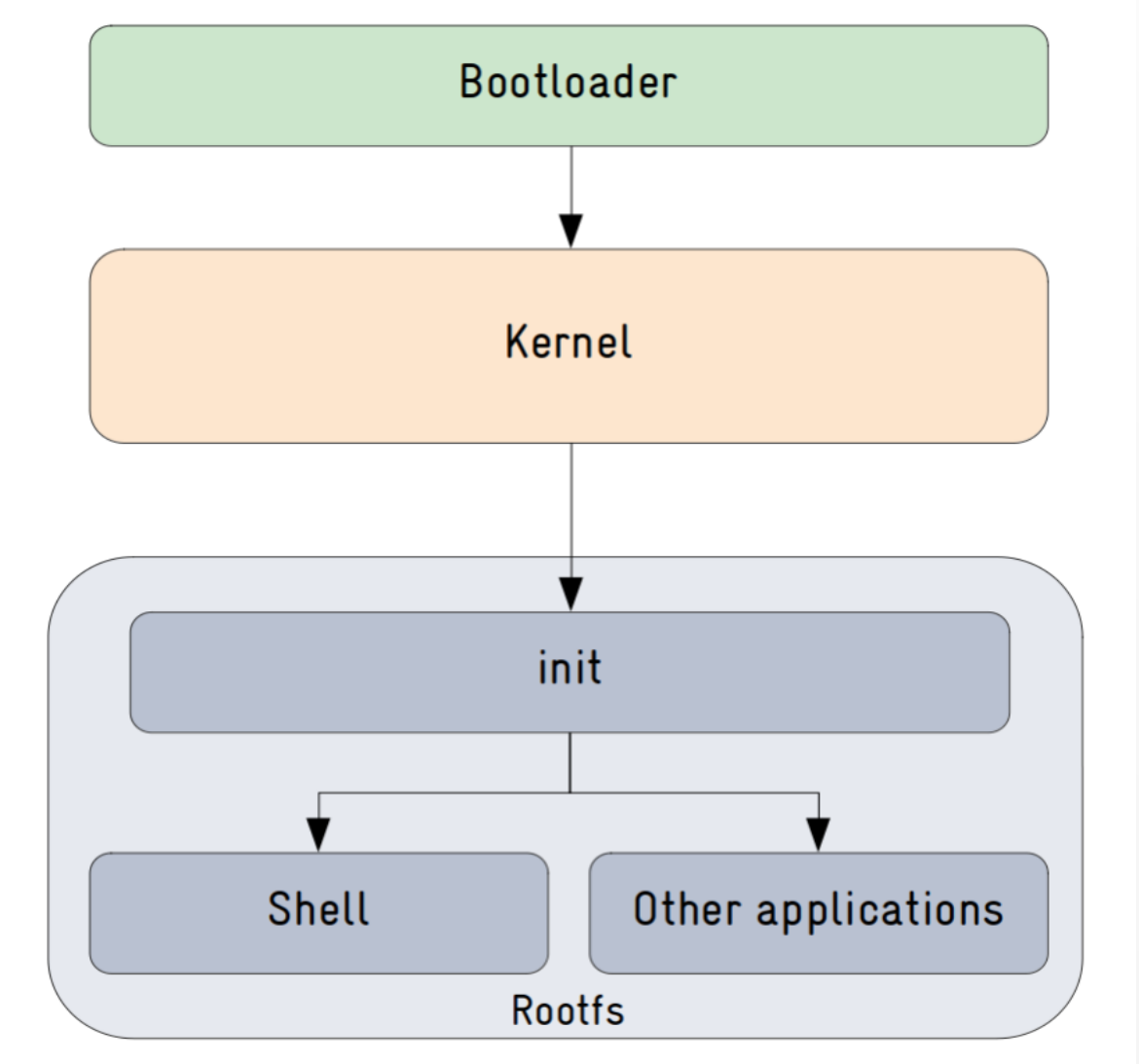
Intro & De ce

automotive crunch

Android Startup Process



Boot overview



Booting in Linux

- After mounting the rootfs, the kernel will run a boot application, also called an init program.
- The init program can be passed as a parameter on the kernel command line, through the init parameter (e.g. init=/sbin/init).
- If the init parameter is not passed, the kernel tries to run the following programs:/sbin/init, /etc/init, /bin/init and /bin/sh.
- If no boot program is found, the kernel panics!

  Kernel panic - not syncing: No init found.

The init program

- There are different implementations of the init program, including:
  - systemd: popular, used in most Linux distributions.
  - System V Init (sysvinit): inherited from Unix systems, used extensively in desktop/server systems, still popular in embedded systems.
  - openrc: used by default in some distributions such as Gentoo.
  - upstart: developed by Canonical for Ubuntu, discontinued.
- Android has its own init mechanism!

Android init

- In Android, the init program is in /system/bin/init:

  ```
  # cat /proc/cmdline
  ... init=/init ...

  # ls -l /init
  lrwxr-x--- 1 root shell 16 2023-05-04 18:59 /init -> /system/bin/init
  ```

- It has 3 main responsibilities:
  - Initialize the operating system's execution environment.
  - Start daemons (init services) and manage their execution.
  - Manage system properties.
- The behavior of the init process is defined in files with the .rc extension.

/system/etc/init/hw/init.rc

```
import /init.environ.rc
import /system/etc/init/hw/init.usb.rc
import /init.${ro.hardware}.rc
import /vendor/etc/init/hw/init.${ro.hardware}.rc
import /system/etc/init/hw/init.usb.configfs.rc
import /system/etc/init/hw/init.${ro.zygote}.rc

# Cgroups are mounted right before early-init using list from /etc/cgroups.json
on early-init
    # Disable sysrq from keyboard
    write /proc/sys/kernel/sysrq 0

    # Android doesn't need kernel module autoloading, and it causes SELinux
    # denials.  So disable it by setting modprobe to the empty string.  Note: to
    # explicitly set a sysctl to an empty string, a trailing newline is needed.
    write /proc/sys/kernel/modprobe \n

    # Set the security context of /adb_keys if present.
    restorecon /adb_keys

    # Set the security context of /postinstall if present.
    restorecon /postinstall
    ...
```

.RC files

- The init configuration is divided into several .rc files, distributed across the various partitions of the operating system and read in the following order:
  /system/etc/init/hw/init.rc
  /system/etc/init/*.rc
  /system_ext/etc/init/*.rc
  /vendor/etc/init/*.rc*
  /odm/etc/init/*.rc*
  /product/etc/init/*.rc*
- In addition, a .rc file can include other .rc files through the import directive:
  import /init.environ.rc
  import /system/etc/init/hw/init.usb.rc
  import /init.${ro.hardware}.rc
  ...

/system/core/init/init.cpp

```
static void LoadBootScripts(ActionManager& action_manager, ServiceList& service_list) {
    Parser parser = CreateParser(action_manager, service_list);

    std::string bootscript = GetProperty("ro.boot.init_rc", "");
    if (bootscript.empty()) {
        parser.ParseConfig("/system/etc/init/hw/init.rc");
        if (!parser.ParseConfig("/system/etc/init")) {
            late_import_paths.emplace_back("/system/etc/init");
        }
        parser.ParseConfig("/system_ext/etc/init");
        if (!parser.ParseConfig("/vendor/etc/init")) {
            late_import_paths.emplace_back("/vendor/etc/init");
        }
        if (!parser.ParseConfig("/odm/etc/init")) {
            late_import_paths.emplace_back("/odm/etc/init");
        }
        if (!parser.ParseConfig("/product/etc/init")) {
            late_import_paths.emplace_back("/product/etc/init");
        }
    } else {
        parser.ParseConfig(bootscript);
    }
}
```

RC file syntax

- Init configuration files have basically two types of declarations: action and service.
- An action declaration has the following format:

  on <trigger> [&& <trigger>]
    <command>
    <command>
    ...
- A service declaration has the following format:

  service <name> <pathname> [<argument>]*
    <option>
    <option>
    ...

Actions and services

- Only the action declaration results in the execution of commands by the init process.
- The service declaration serves only to describe the service.
- To execute it, it is necessary to use the start or class_start commands within an action declaration.
- There are two types of triggers that can trigger an action:
  - Event triggers: triggered via the trigger command or QueueEventTrigger() function in the init code.
  - Property triggers: triggered when the value of a property is changed.

Triggers fired at boot

- After interpreting the rc files, the init process (via the QueueEventTrigger() function) will fire three boot triggers:
  - early-init
  - Init
  - late-init
- Firing these triggers will cause the execution of the commands defined in the corresponding action declaration, starting the Android boot process.
- Within an action declaration, the trigger command can be used to trigger other actions during the boot process.

/system/core/init/init.cpp

```
int SecondStageMain(int argc, char** argv) {
    ...
    LoadBootScripts(am, sm);
    ...

    am.QueueEventTrigger("early-init");
    ...

    am.QueueEventTrigger("init");
    ...

    std::string bootmode = GetProperty("ro.bootmode", "");
    if (bootmode == "charger") {
        am.QueueEventTrigger("charger");
    } else {
        am.QueueEventTrigger("late-init");
    }
    ...
}
```

/system/etc/init/hw/init.rc

```
on early-init
    # Disable sysrq from keyboard
    write /proc/sys/kernel/sysrq 0

    # Android doesn't need kernel module autoloading, and it causes SELinux
    # denials.  So disable it by setting modprobe to the empty string.  Note: to
    # explicitly set a sysctl to an empty string, a trailing newline is needed.
    write /proc/sys/kernel/modprobe \n

    # Set the security context of /adb_keys if present.
    restorecon /adb_keys
    ...

on init
    sysclktz 0

    # Mix device-specific information into the entropy pool
    copy /proc/cmdline /dev/urandom
    copy /system/etc/prop.default /dev/urandom

    symlink /proc/self/fd/0 /dev/stdin
    symlink /proc/self/fd/1 /dev/stdout
    symlink /proc/self/fd/2 /dev/stderr
```

/system/etc/init/hw/init.rc

```
on late-init
    trigger early-fs

    # Mount fstab in init.{$device}.rc by mount_all command. Optional parameter
    # '--early' can be specified to skip entries with 'latemount'.
    # /system and /vendor must be mounted by the end of the fs stage,
    # while /data is optional.
    trigger fs
    trigger post-fs

    # Mount fstab in init.{$device}.rc by mount_all with '--late' parameter
    # to only mount entries with 'latemount'. This is needed if '--early' is
    # specified in the previous mount_all command on the fs stage.
    # With /system mounted and properties form /system + /factory available,
    # some services can be started.
    trigger late-fs

    ...

    trigger early-boot
    trigger boot
```

Commands

- There are several commands that can be used in the configuration file. init, including chdir, chmod, chown, copy, exec, export, hostname, ifup, insmod, loglevel, mkdir, symlink, restart, mount, trigger, etc.
- Note that, despite the similarity, these are not shell commands!
- The init documentation, including a description of the main commands, is available in the Android source code.

  system/core/init/README.md

Init services

- The service directive allows you to declare init services (e.g. programs that will be automatically executed at Android boot).
- When declaring a service, several options can be used to parameterize the execution of the service.
- The service directive only describes the service, but does not automatically start it.
- The execution of a service is controlled using the commands start, stop, class_start, class_reset and class_stop.

Declaration of services

```
service surfaceflinger /system/bin/surfaceflinger
    class core animation
    user system
    group graphics drmrpc readproc
    capabilities SYS_NICE
    onrestart restart --only-if-running zygote
    task_profiles HighPerformance
    socket pdx/system/vr/display/client
    socket pdx/system/vr/display/manager
    socket pdx/system/vr/display/vsync

service bootanim /system/bin/bootanimation
    class core animation
    user graphics
    group graphics audio
    disabled
    oneshot
    ioprio rt 0
    task_profiles MaxPerformance
```

Service options

Option	Description
class	service class
user	service user
group	service group(s)
socket	creates a UNIX domain socket and passes the descriptor to the service
disabled	does not automatically start the service by class
critical	restarts if the service terminates more than 4 times in 4 minutes
onrestart	executes a command when restarting the service
oneshot	executes the service only once

Starting services

```
on early-init
        ...
        start ueventd
on init
        ...
        start servicemanager
        start hwservicemanager
        start vndservicemanager
on boot
        ...
        class_start hal
        class_start core
on nonencrypted
        class_start main
        class_start late_start
```

Properties

- Properties in Android are global variables shared and accessible by the entire operating system.
  - In practice, they are a global dictionary of strings in the key=value format.
  - Used to store operating system information and settings.
- Properties are initialized when Android boots via:
  - Reading files with the .prop extension.
  - Executing setprop commands.
- Properties are kept in RAM while the operating system is running.

.Prop files

- Most properties are declared in files generated during the Android build and loaded at system boot:

  /system/etc/build.prop
  /system/default.prop
  /system/build.prop
  /system_ext/etc/build.prop
  /system_dlkm/etc/build.prop
  /vendor/etc/build.prop
  /vendor/default.prop
  /vendor/build.prop
  /vendor_dlkm/etc/build.prop
  /odm/etc/build.prop
  /odm_dlkm/etc/build.prop
  /product/etc/build.prop
  /data/local.prop

- See the init code in property_service.cpp.

Manipulation properties

- Properties can be manipulated in the terminal via getprop/setprop commands,and in applications via APIs provided by the operating system.
- Properties are read through an intelligent memory-sharing mechanism.
- Writing properties is controlled by the init process via a UNIXdomain socket:

```
# ls -l /dev/socket/property_services
rw-rw-rw- 1 root root 0 2023-05-18 12:29 /dev/socket/property_service
```

Getprop

- The getprop command can be used to list existing properties:

```
# getprop
[apexd.status]: [ready]
[bluetooth.device.class_of_device]: [90,2,12]
[bluetooth.profile.a2dp.source.enabled]: [true]
[bluetooth.profile.asha.central.enabled]: [true]
[bluetooth.profile.avrcp.target.enabled]: [true]
[bluetooth.profile.bap.broadcast.assist.enabled]: [true]
[bluetooth.profile.bap.unicast.client.enabled]: [true]
[bluetooth.profile.bas.client.enabled]: [true]
[bluetooth.profile.ccp.server.enabled]: [true]
[bluetooth.profile.csip.set_coordinator.enabled]: [true]
[bluetooth.profile.gatt.enabled]: [true]
[bluetooth.profile.hap.client.enabled]: [true]
[bluetooth.profile.hfp.ag.enabled]: [true]
[bluetooth.profile.hid.device.enabled]: [true]
[bootreceiver.enable]: [1]
[bpf.progs_loaded]: [1]
[build.version.extensions.r]: [3]
[build.version.extensions.s]: [3]
[build.version.extensions.t]: [3]
...
```

Setprop

- The setprop command can be used to change a property:

  # setprop labworks.teste 1

- By default, properties are created in memory. If you want to create a persistent property, put persist in the property name.

  # setprop persist.labworks.teste 1

- Persistent properties are stored in /data/property/ and are loaded automatically when Android boots:

  # ls -l /data/property/

  -rw------- 1 root root 617 2023-05-18 12:29 persistent_properties

Special properties

- Some properties have a special meaning:
  - *persist.**: persistent properties stored in /data/property/.
  - *ro.**: read-only properties (once created, cannot be changed).
  - *ro.boottime.**: boot time statistics.
  - *init.svc.<name>*: properties indicating the state of an init service.
  - *ctl.start*: special property used to start a service.
  - *ctl.stop*: special property used to stop a service.

Ownership properties

- The init process is capable of taking actions based on ownership changes:

```
on property:sys.boot_completed=1
    bootchart stop
    exec - system system -- /bin/rm -rf /data/per_boot
    mkdir /data/per_boot 0700 system system encryption=Require key=per_boot_ref

on property:persist.vendor.net.doxlat=true
    setprop net.464xlat.cellular.enabled true

on property:sys.usb.config=none && property:sys.usb.configfs=0
    stop adbd
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/bDeviceClass 0
    setprop sys.usb.state ${sys.usb.config}

on property:sys.init_log_level=*
    loglevel ${sys.init_log_level}
```

The framework boot

- The init program is responsible for initializing the Android native layer, but does not manage the framework.
- The framework is initialized by a service called zygote, which is run by init through a program called app_process.
  - The app_process is a program capable of executing DEX bytecode in a Dalvik or ART virtual machine environment.
  - Zygote is the program responsible for initializing the framework and the Android application execution environment.

App_process and zygote

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote stream 660 root system
    socket usap_pool_primary stream 660 root system
    onrestart exec_background - system system -- /system/bin/vdc volume abort_fuse
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart cameraserver
    onrestart restart media
    onrestart restart media.tuner
    onrestart restart netd
    onrestart restart wificond
    task_profiles ProcessCapacityHigh
    critical window=${zygote.critical_window.minute:-off} target=zygote-fatal
```

Zygote

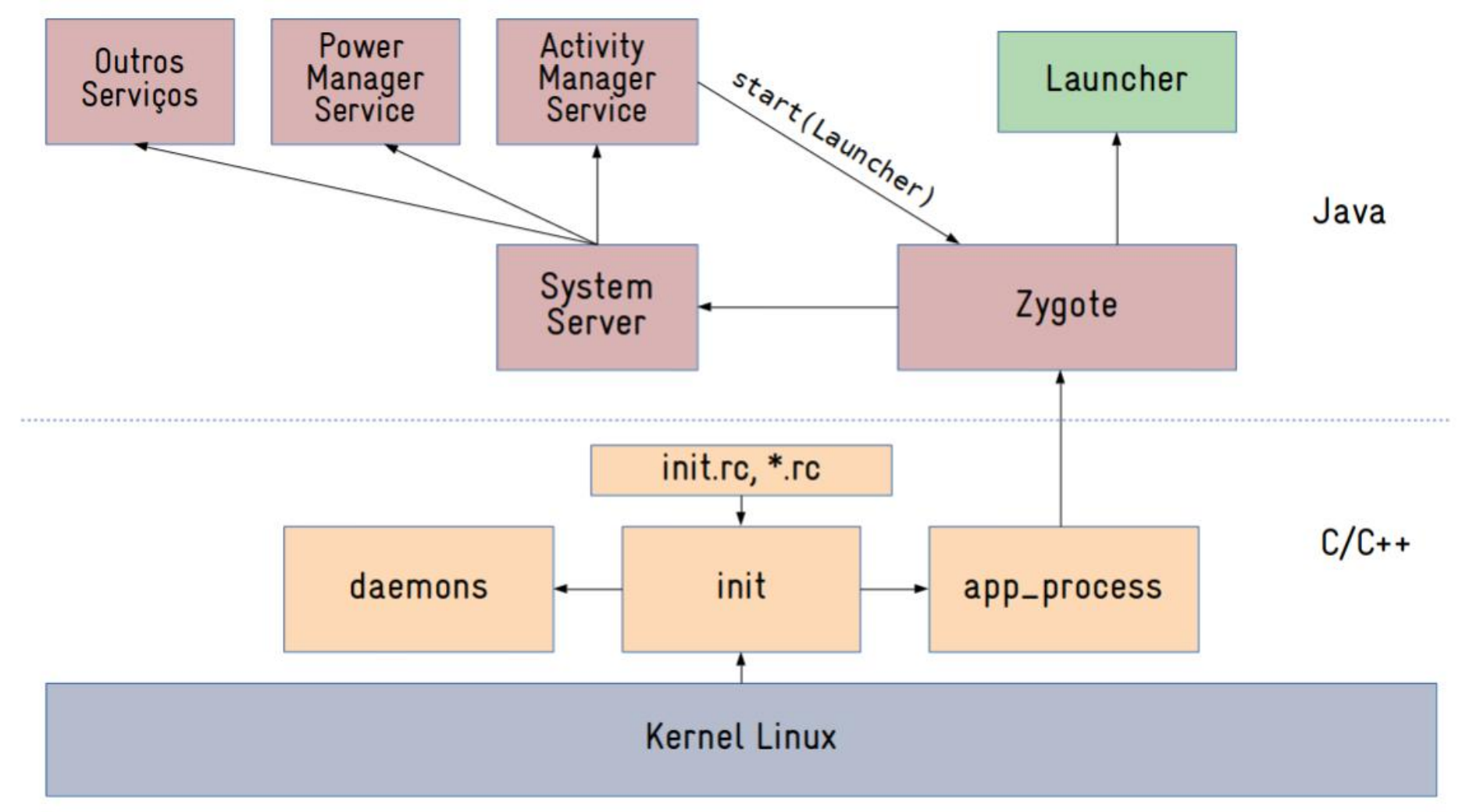
- Zygote is one of the main Android daemons, responsible for:
  - Preloading all Java classes into memory.
  - Starting the system server (framework service daemon).
  - Waiting for connections on a UNIX domain socket (/dev/socket/zygote) to start new Android applications.
- If the system is compiled to support multiple ABIs (e.g. 32 and 64 bits), it is possible to have more than one zygote process running:

```
# ps -A | grep zygote
root          395     1 14083636    165372 do_sys_poll              0 S zygote64
nobody        396     1 1630100     119300 futex_wait_queue_me      0 S zygote
```

System server

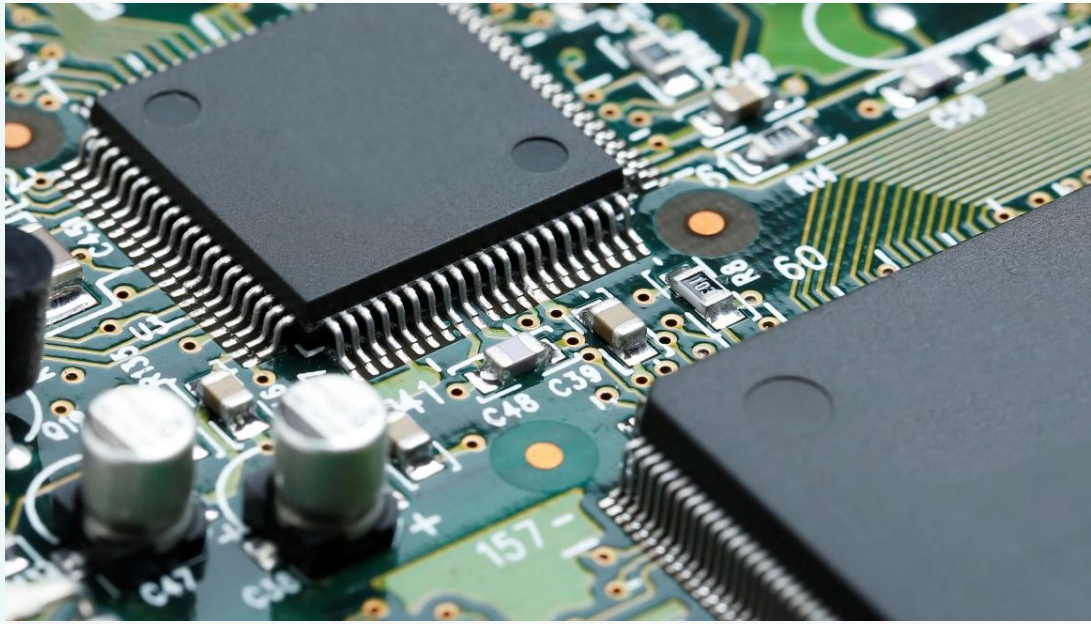
- The system server is a separate process executed by Zygote during its initialization.
  - This process is responsible for registering several framework services, which can be used by applications through the Android API.
    - One of these registered services is the Activity Manager Service.
- At the end of its initialization, the Activity Manager Service sends an Intent of the type CATEGORY_HOME, causing the Launcher application to be executed.

Diagram: Android Startup



Processes at startup

```
USER            PID  PPID     VSZ    RSS WCHAN              ADDR S NAME
root              1     0 10885820  10628 do_epoll_wait       0 S init
root            395     1 14078208 162648 do_sys_poll         0 S zygote
system          637   395 14888480 319412 do_epoll_wait       0 S system_server
u0_a78          868   395 13674004 217768 do_epoll_wait       0 S com.android.systemui
webview_zygote  919   396  1903184  89456 do_sys_poll         0 S webview_zygote
network_stack   931   395 13520300 116112 do_epoll_wait       0 S com.android.networkstack.process
bluetooth       936   395 13618384 132192 do_epoll_wait       0 S com.android.bluetooth
secure_element  955   395 13507784  96464 do_epoll_wait       0 S com.android.se
radio           957   395 13581824 139320 do_epoll_wait       0 S com.android.phone
system          968   395 13586948 147756 do_epoll_wait       0 S com.android.settings
u0_a88         1011   395 13507692  99448 do_epoll_wait       0 S android.ext.services
u0_a83         1186   395 13517028 112732 do_epoll_wait       0 S com.android.cellbroadcastreceiver
u0_a56         1222   395 13525044 108024 do_epoll_wait       0 S com.android.dialer
u0_a74         1311   395 13551964 160108 do_epoll_wait       0 S com.android.launcher3
u0_a85         1355   395 13527612 109628 do_epoll_wait       0 S com.android.permissioncontroller
u0_a68         1356   395 13573372 115272 do_epoll_wait       0 S com.android.inputmethod.latin
u0_a79         1403   395 13555368 123836 do_epoll_wait       0 S com.android.providers.media.module
u0_a67         1447   395 13513964 106812 do_epoll_wait       0 S com.android.deskclock
u0_a76         1466   395 13506396  94124 do_epoll_wait       0 S com.android.carrierconfig
...
```



Compilation Modules

Modules

- Every AOSP software component (native application, Java application, library, etc.) is called a module by the Android build system.
- The instructions for compiling a module can be defined in a makefile (Android.mk) or in a Blueprint file (Android.bp).
- The use of makefiles has been discontinued in the latest versions of Android, and Blueprint files have become the standard.
- In any case, in this section of the training we will study both compilation file formats.

Makefiles

- Android is capable of compiling modules through makefiles.
- To do this, simply create a file called Android.mk with the compilation parameters in the root of the module's source code.
- It is not necessary to define compilation commands in this file, only variables that will be used to parameterize the module's compilation.
- The kati tool is responsible for processing Android.mk and generating a .ninja compilation file.
- An (outdated) documentation about Android.mk is available in the Android source code at build/core/build-system.html.

Example: Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := myapp
LOCAL_SRC_FILES := myapp.c
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_CFLAGS := -O0 -Wall
LOCAL_VENDOR_MODULE := true
include $(BUILD_EXECUTABLE)
```

Android.mk: main variables

Variable	Description
LOCAL_SRC_FILES	List of source files that will be compiled
LOCAL_MODULE	Name of the module in the build system
LOCAL_PACKAGE_NAME	Name of the Android application (.apk)
LOCAL_SHARED_LIBRARIES	List of native libraries that the module depends on
LOCAL_C_INCLUDES	Additional paths of header files usedby the module
LOCAL_JAVA_LIBRARIES	List of Java libraries that the module depends on

Android.mk: main variables (cont.)

Variable	Description
LOCAL_CFLAGS	Extra flags for the C and C++ compilers
LOCAL_CPPFLAGS	Extra flags for the C++ compiler
LOCAL_LDFLAGS	Extra flags for the linker
LOCAL_MODULE_CLASS	Indicates the module type (ETC, APPS, EXECUTABLES, etc.)
LOCAL_MODULE_PATH	Module installation location
LOCAL_VENDOR_MODULE	Forces the module to be installed in the /vendor partition

Android.mk: build types

Macro	Description
BUILD_EXECUTABLE	Compiles and generates an executable in ELF format
BUILD_SHARED_LIBRARY	Compiles and generates a dynamic library (.so)
BUILD_STATIC_LIBRARY	Compiles and generates a static library (.a)
BUILD_JAVA_LIBRARY	Compiles and generates a Java library (.jar)
BUILD_PACKAGE	Compiles and generates an Android application (.apk)
BUILD_PREBUILT	Installs a pre-compiled file or binary

Android.mk: macros

- Several macros are available for use in an Android.mk:
  - all-subdir-c-files: returns all C files recursively from the current directory.

    LOCAL_SRC_FILES := $(call all-subdir-c-files)
  - all-java-files-under: returns all Java files recursively from the directory passed as a parameter.

    LOCAL_SRC_FILES := $(call all-java-files-under, src)
- Most macros are defined in build/core/definitions.mk.

Blueprint files

- Currently, Blueprint files are the default mechanism for building modules in Android.
- To build a module with Blueprint, simply create a file called Android.bp in the root of your source code with the module's build parameters.
- This file is simple and has a declarative syntax similar to JSON.
  - The syntax and semantics of Blueprint are similar to Bazel's BUILD files (which should make it easier to migrate to the future Android build system).
- The Soong build system is responsible for processing Android.bp and generating the build .ninja file.

Example Android.bp

```
cc_binary {
    name: "myapp",
    srcs: ["myapp.c"],
    shared_libs: ["liblog"],
    cflags: [
        "-O0",
        "-Wall",
    ],
    vendor: true,
}
```

Android.bp: main properties

Property	Description
srcs	List of source files that will be compiled
name	Name of the module in the build system
shared_libs	List of native libraries that the module depends on
include_dirs	Additional paths of header files used by the module
libs	List of Java libraries that the module depends on
required	List of modules that a module depends on (runtime dependency)

Android.bp: main properties (cont.)

Property	Description
export_include_dirs	Directories with include files that will be exported to other modules
cflags	Extra flags for the C and C++ compilers
cppflags	Extra flags for the C++ compiler
ldflags	Extra flags for the linker
vendor	Forces the module to be installed on the /vendor partition

Android.bp build types

Build type	Description
cc_binary	Compiles and generates an executable in ELF format
cc_library_shared	Compiles and generates a dynamic library (.so)
cc_library_static	Compiles and generates a static library (.a)
java_binary	Compiles and generates a Java executable
java_library	Compiles and generates a Java library (.jar)
android_app	Compiles and generates an Android application (.apk)
cc_prebuilt_binary	Installs a pre-compiled binary

Tools and documentation

- You can convert an Android.mk to Android.bp using the androidmk tool:

  :$ androidmk Android.mk > Android.bp

- The bpfmt command can be used to format a Blueprint file:

  $ bpfmt -w Android.bp

- See Soong Modules Reference for a complete list of properties and build types that can be used in Blueprint files.
- For more information on the Soong build system: https://android.googlesource.com/platform/build/soong/

Compiling a module

- With the compilation file created (Android.mk or Android.bp), to compile simply run the m command, passing the module name:

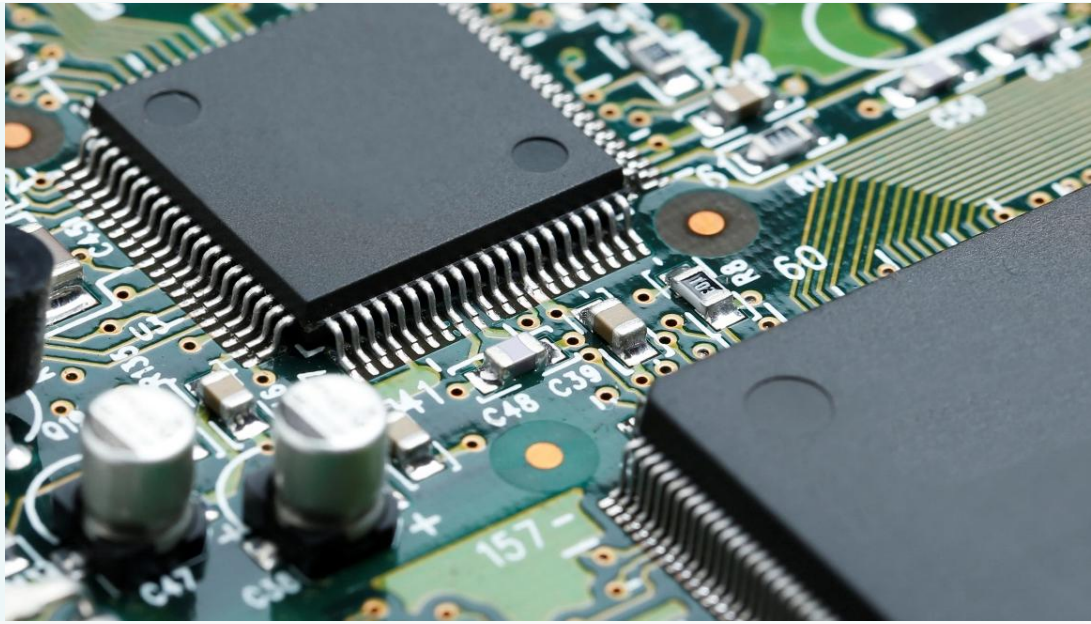
```
$ m myapp
```

- The entire compilation of the module takes place in the out/ directory:

```
$ ls out/soong/.intermediates/device/labworks/superdroid/myapp/*/
myapp  myapp.d  obj  unstripped
$ ls out/target/product/vsoc_x86/obj/EXECUTABLES/myapp_intermediates
myapp
```

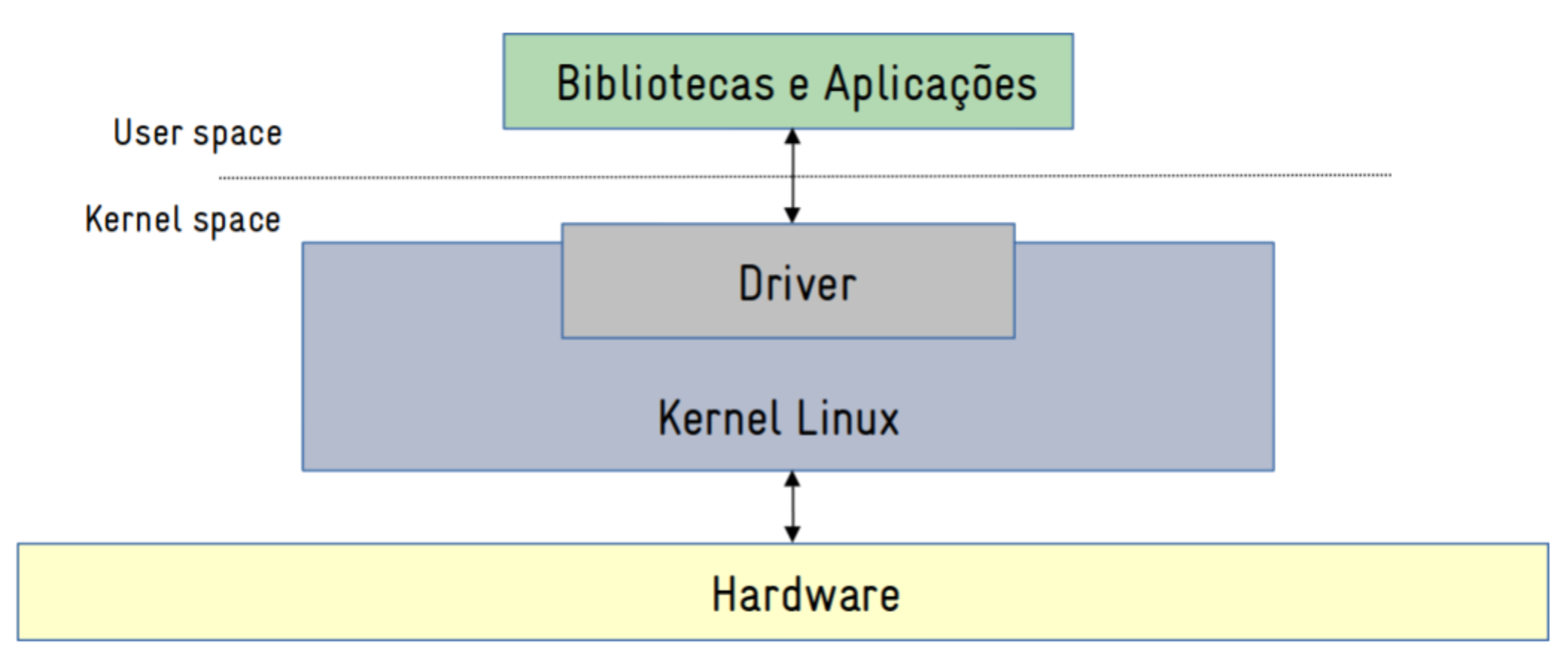
- To clean the compilation, simply remove the intermediate directories or run target clean-<module>:

```
$ m clean-myapp
```

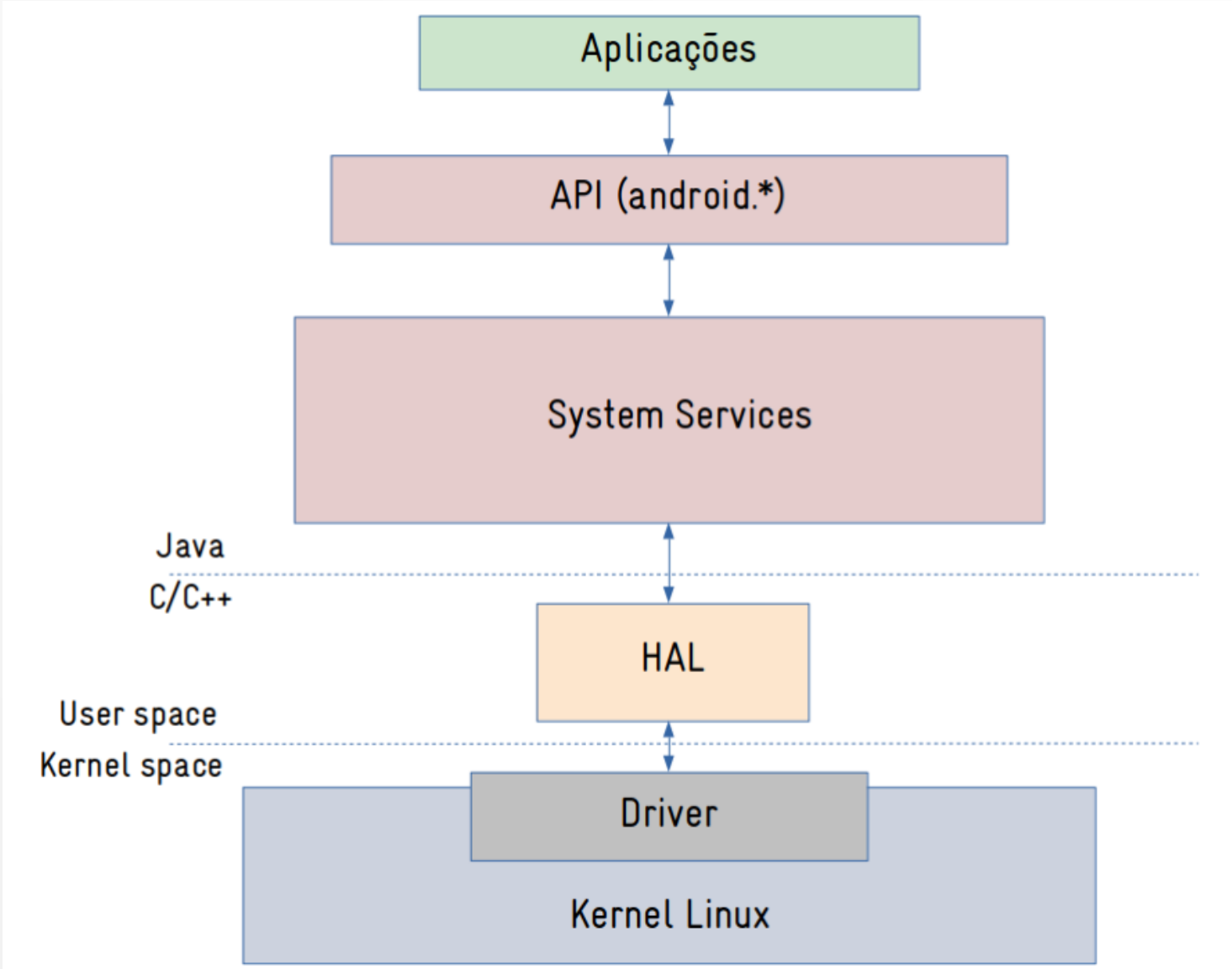


Android HAL

Hardware abstraction layer in Linux



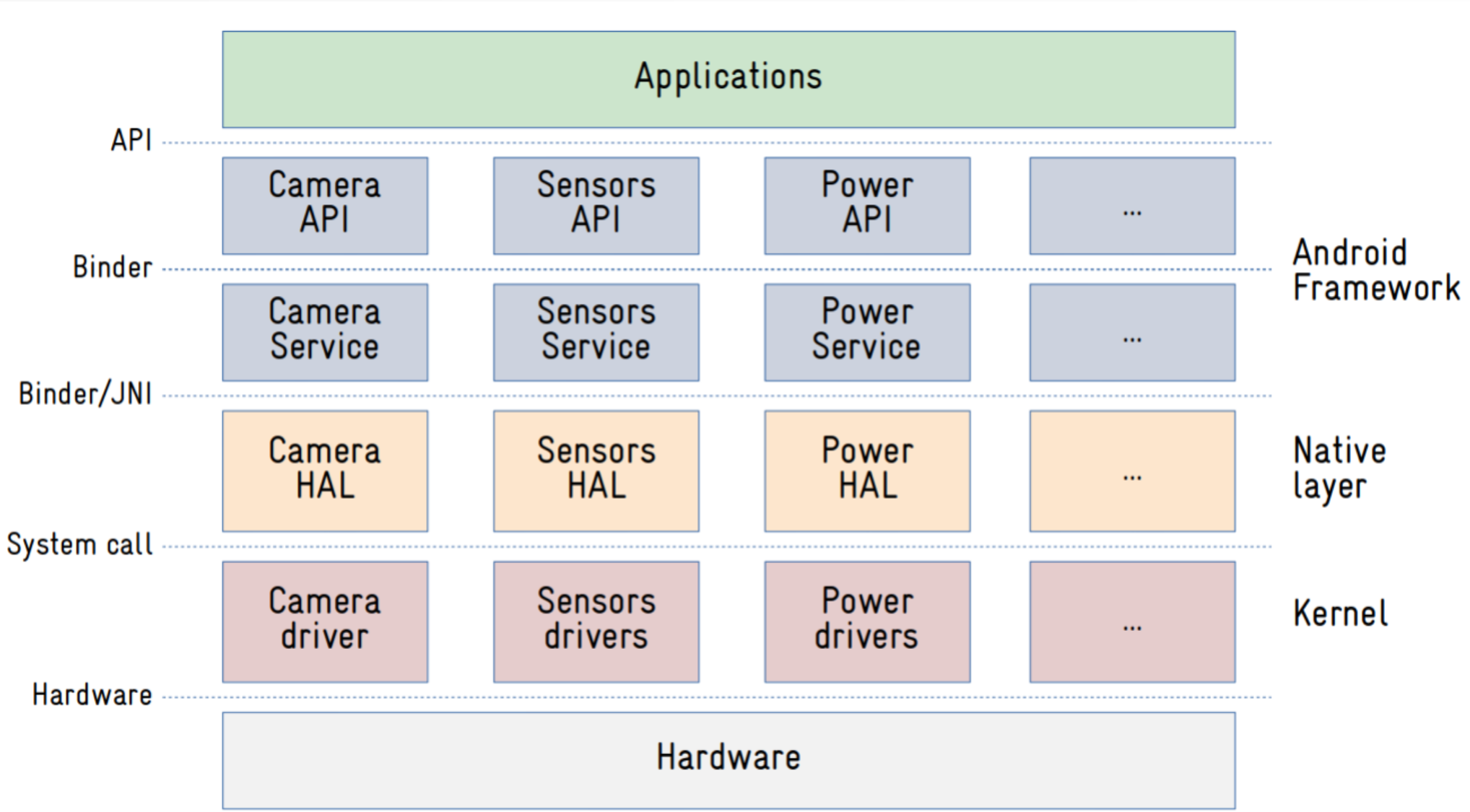
Hardware abstraction layer in Android



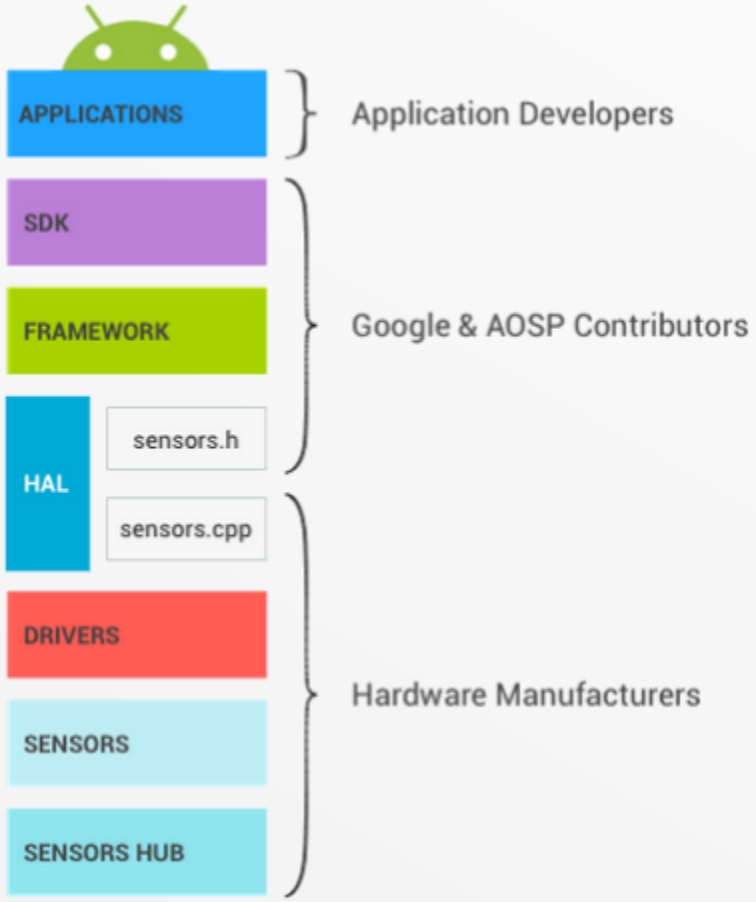
Hardware abstraction layer

- On Linux systems, access to a hardware device is typically exposed to applications through entries in /dev and /sys.
- Android relies on an additional layer called HAL (Hardware Abstraction Layer) to abstract access to hardware.
- The main motivation for the existence of HAL is to decouple the framework from the hardware access code, making it easier to integrate Android generic code (AOSP) with the manufacturer's hardware platform.
- Google does not specify how a HAL should be developed, but only defines a standardized interface that should be implemented by the HAL.

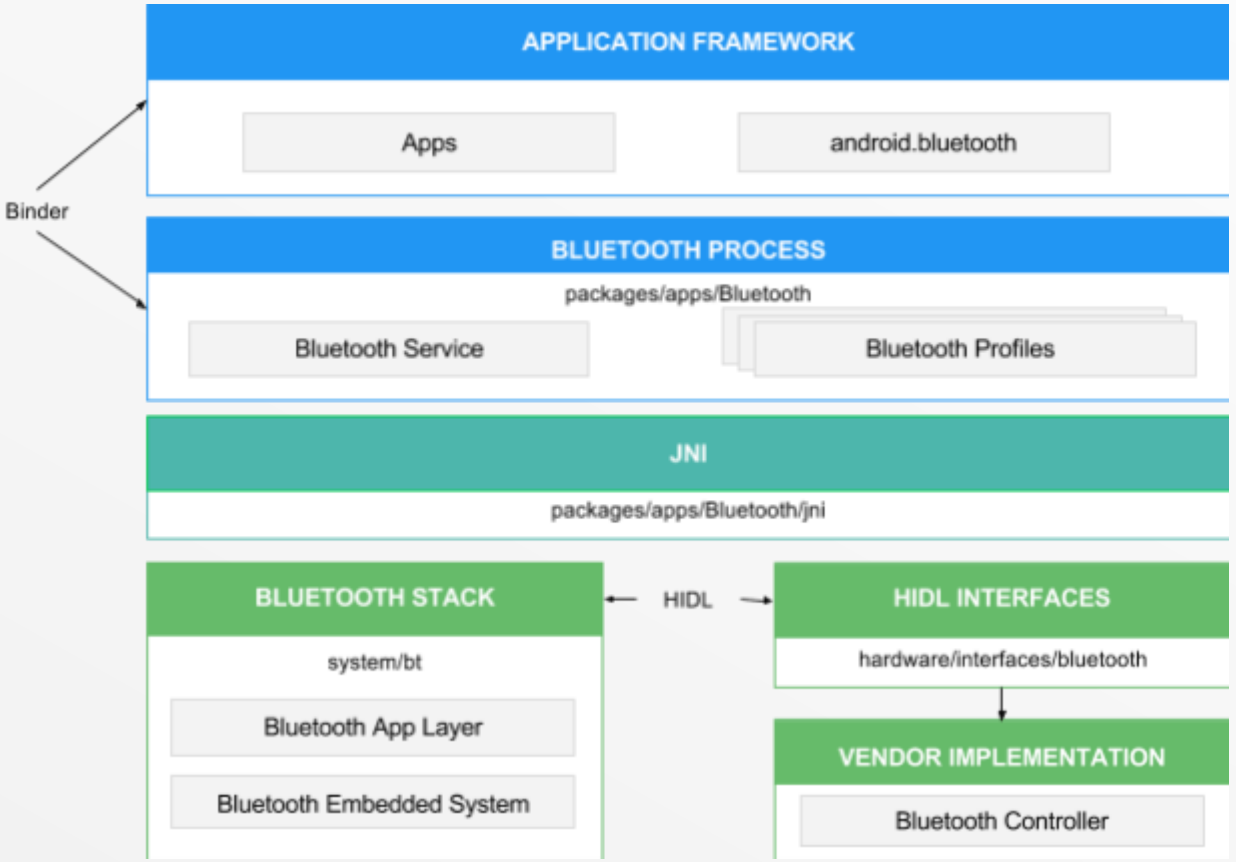
HAL and the Android layers



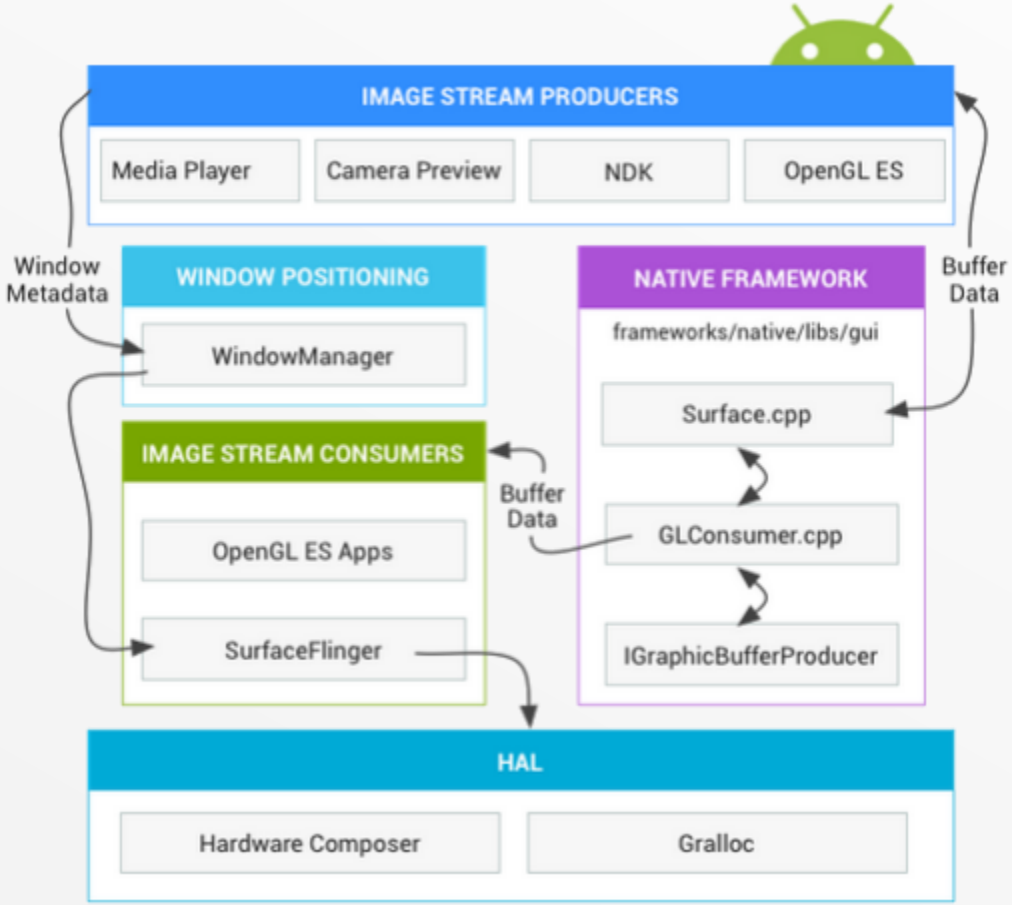
Sensors HAL


APPLICATIONS
Application Developers
SDK
FRAMEWORK
HAL
sensors.h
sensors.cpp
Google & AOSP Contributors
DRIVERS
SENSORS
SENSORS HUB
Hardware Manufacturers

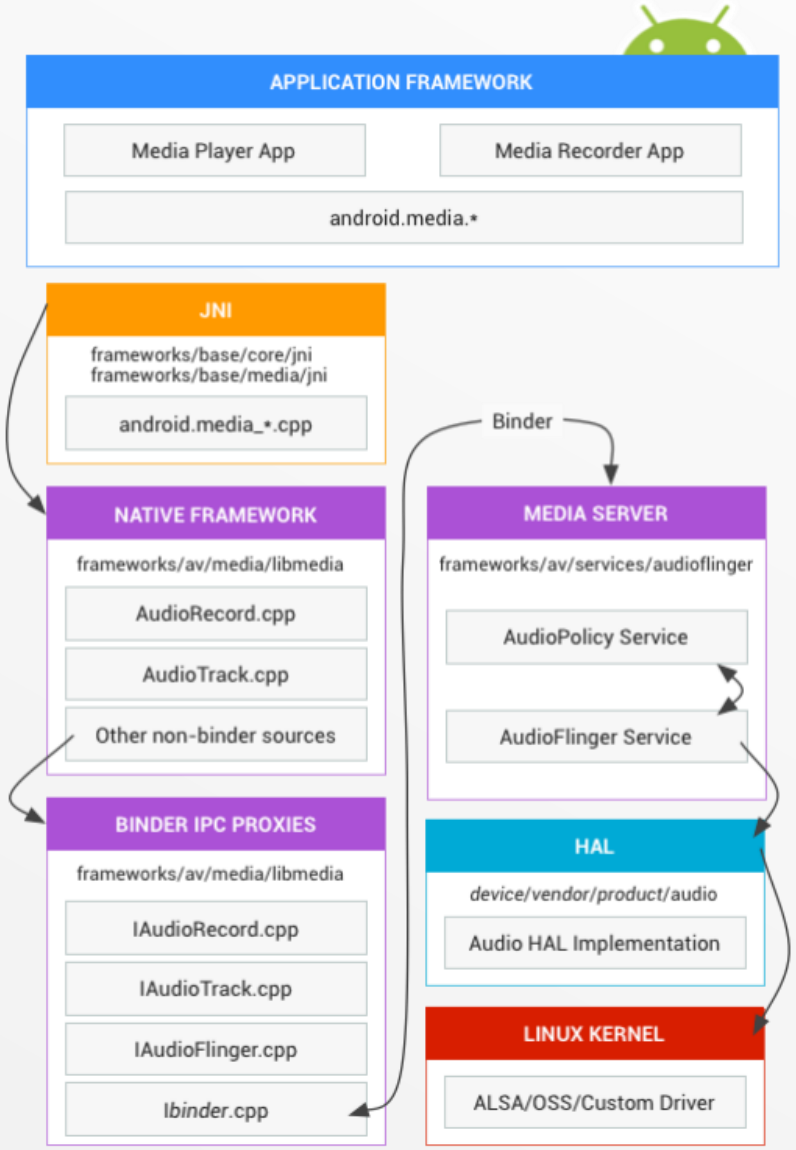
Bluetooth HAL



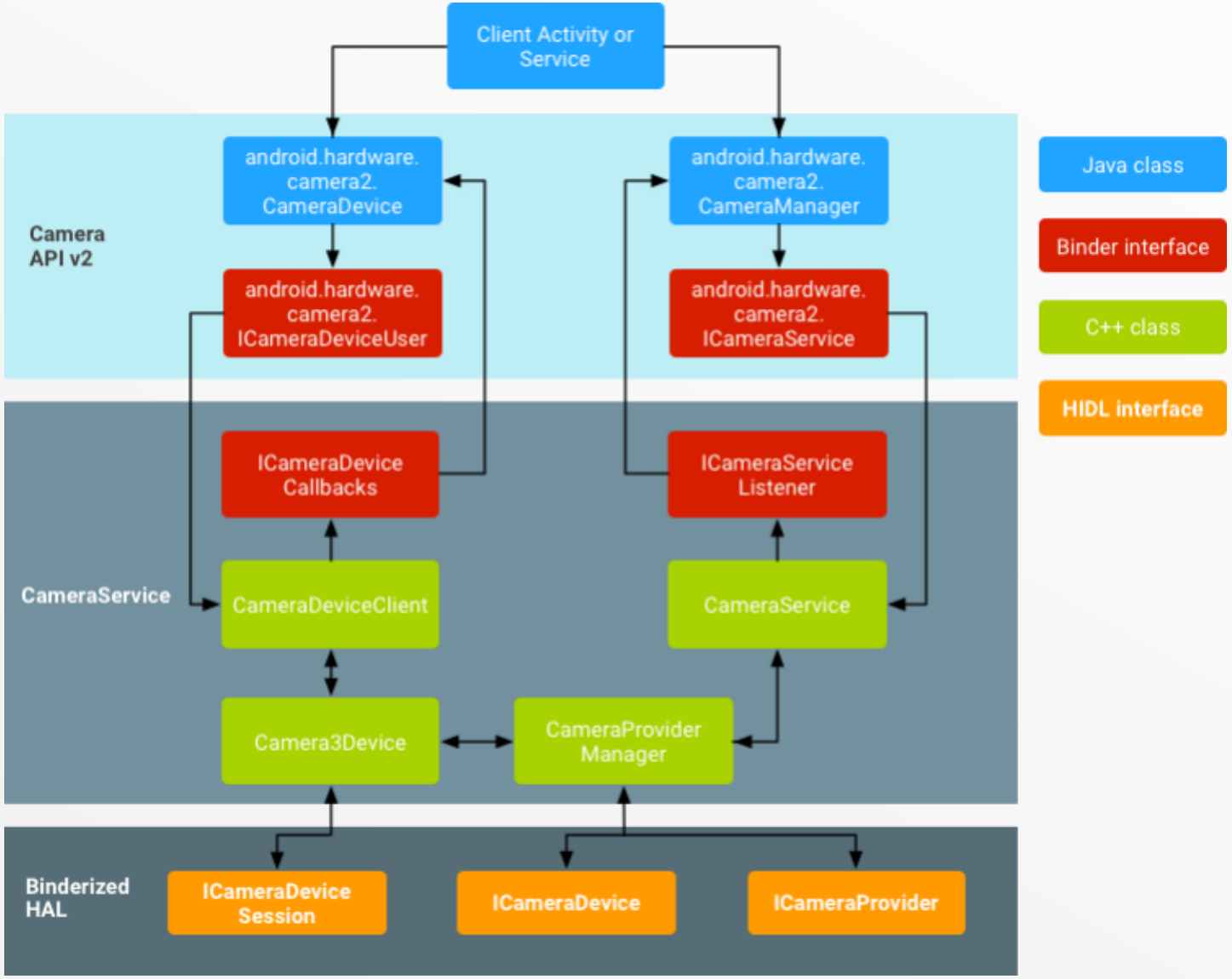
Graphics HAL



Audio HAL


APPLICATION FRAMEWORK
Media Player App
Media Recorder App
android.media.*
JNI
frameworks/base/core/jni
frameworks/base/media/jni
android.media_*.cpp
Binder
NATIVE FRAMEWORK
frameworks/av/media/libmedia
AudioRecord.cpp
AudioTrack.cpp
Other non-binder sources
MEDIA SERVER
frameworks/av/services/audioflinger
AudioPolicy Service
AudioFlinger Service
BINDER IPC PROXIES
frameworks/av/media/libmedia
IAudioRecord.cpp
IAudioTrack.cpp
IAudioFlinger.cpp
Ibinder.cpp
HAL
device/vendor/product/audio
Audio HAL Implementation
LINUX KERNEL
ALSA/OSS/Custom Driver

Camera HAL


Client Activity or Service
Camera API v2
android.hardware.camera2.CameraDevice
android.hardware.camera2.CameraManager
android.hardware.camera2.ICameraDeviceUser
android.hardware.camera2.ICameraService
CameraService
ICameraDevice Callbacks
ICameraService Listener
CameraDeviceClient
CameraService
Camera3Device
CameraProvider Manager
Binderized HAL
ICameraDevice Session
ICameraDevice
ICameraProvider
Java class
Binder interface
C++ class
HIDL interface

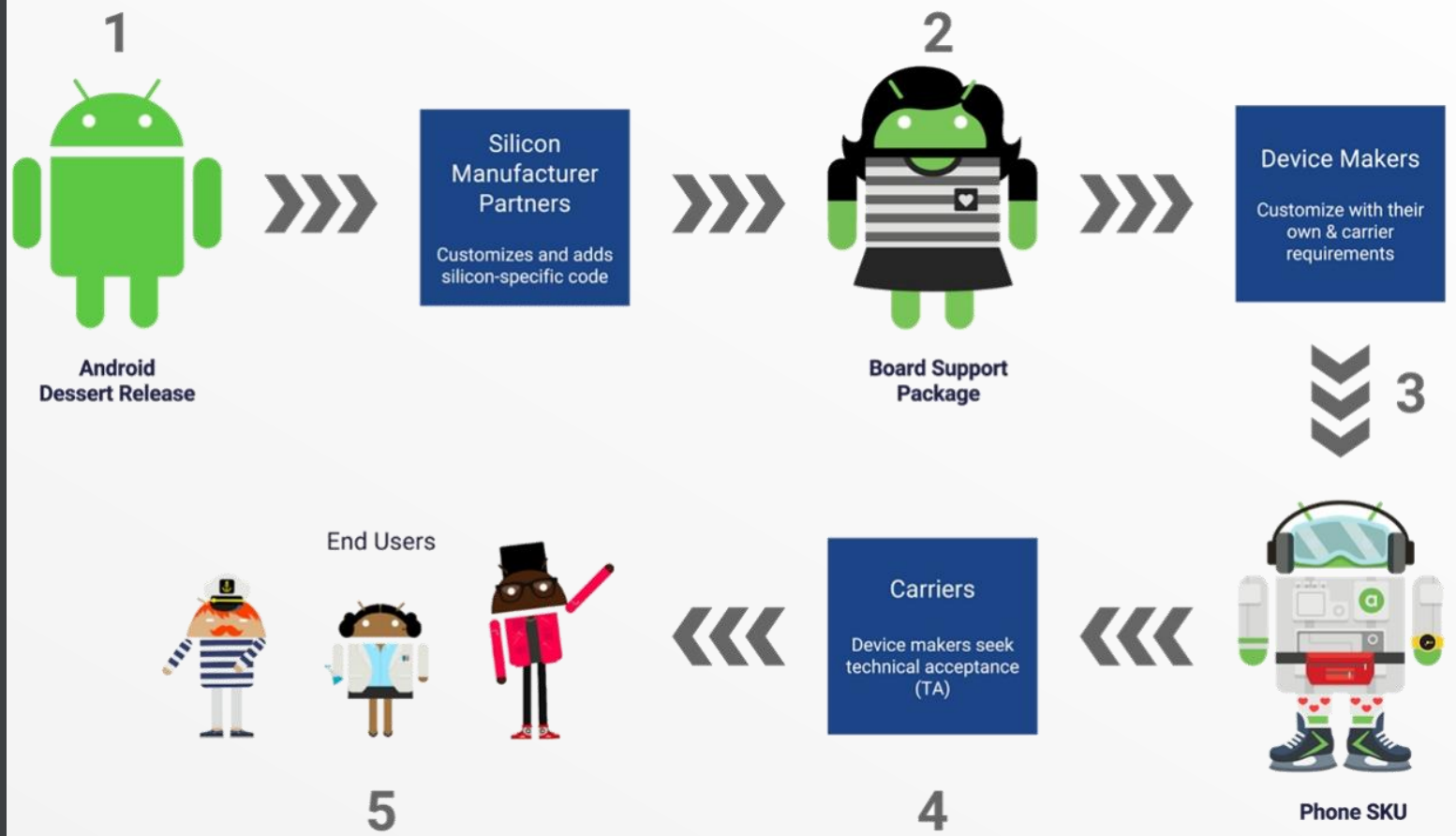
HAL architecture

- Until Android 7, HALs were implemented in shared libraries (.so),consumed by framework services via JNI.
- It is a simple and functional design, but with some shortcomings:
  - The HAL interface is not stable. With each new version of Android, the interface definition can change, requiring the HAL to be reimplemented.
  - There is an API/ABI dependency between the framework services and the HALs.
- For example, if the framework is recompiled with a different toolchain, it is necessary to recompile the HALs as well. These shortcomings make the framework code very dependent on the HAL implementation, and this dependency makes the operating system update process difficult, fragmenting the platform!

Platform fragmentation

ANDROID PLATFORM VERSION		API LEVEL	CUMULATIVE DISTRIBUTION
4.4	KitKat	19	
5.0	Lollipop	21	99,3%
5.1	Lollipop	22	99,0%
6.0	Marshmallow	23	97,2%
7.0	Nougat	24	94,4%
7.1	Nougat	25	92,5%
8.0	Oreo	26	90,7%
8.1	Oreo	27	88,1%
9.0	Pie	28	81,2%
10.	Q	29	68,0%
11.	R	30	48,5%
12.	S	31	24,1%
13.	T	33	5,2%

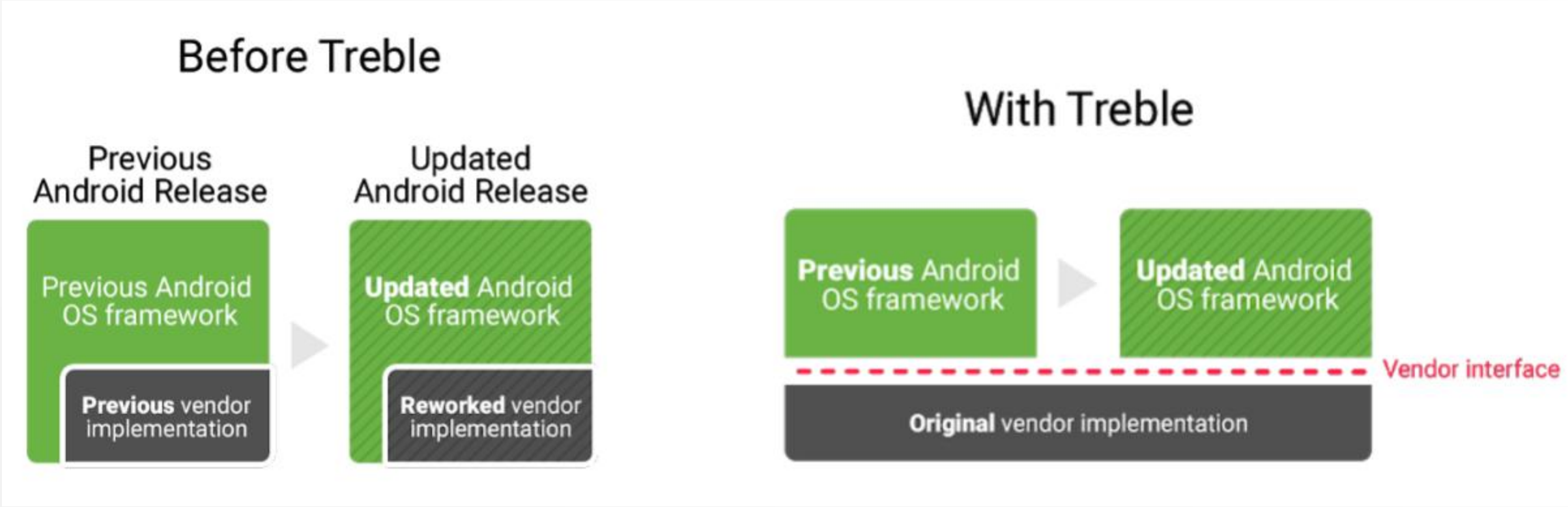
Last updated: January 6th, 2023



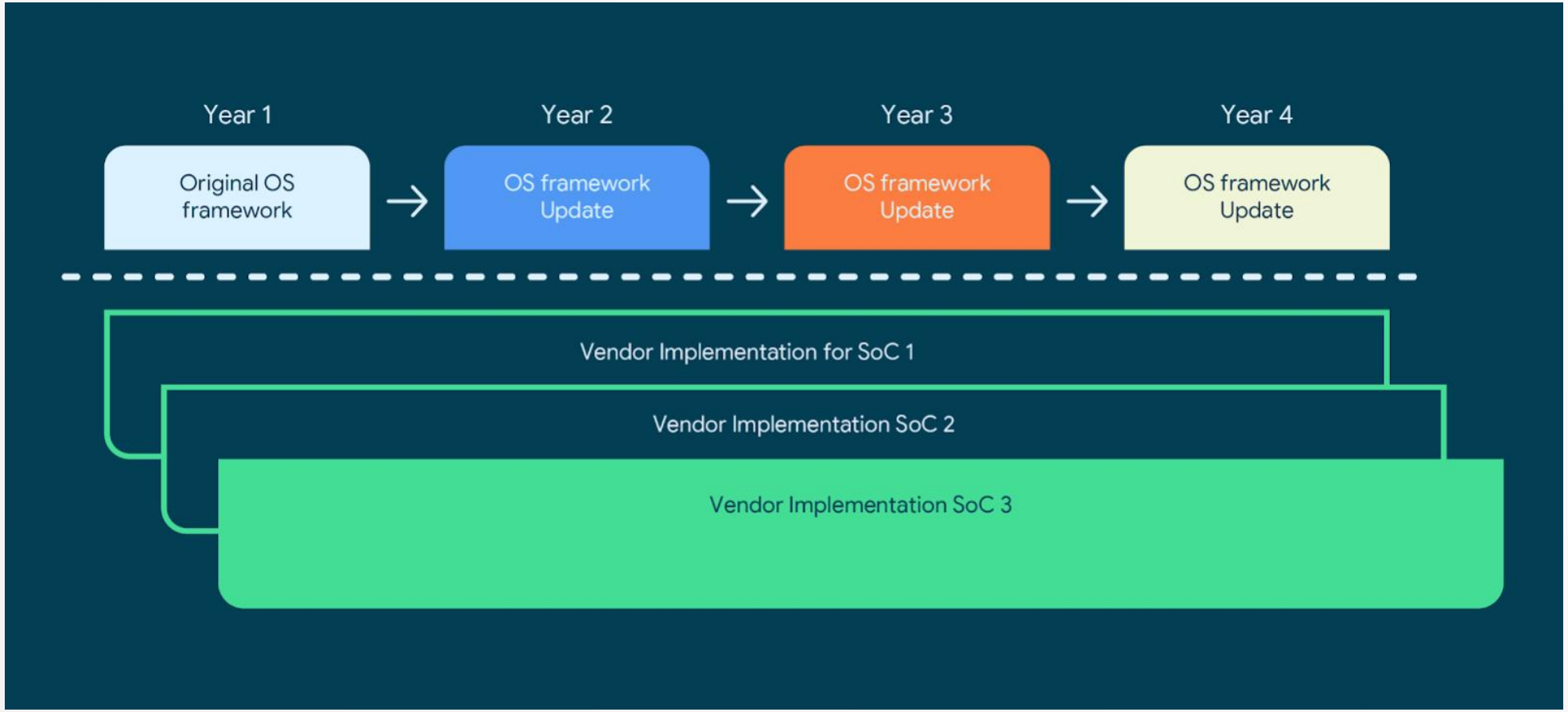
Here comes Treble!

- In 2017, the Treble project was created to minimize/solve this problem. Here comes Treble: A modular base for Android
- Implemented starting with Android 8, the idea of the project is to stabilize and make the interface between the Android framework and the hardware manufacturer's implementation (vendor/oem) independent.
- With Treble, with each Android release, Google ensures that the GSI (Generic System Image) is backwards compatible with the last 3 releases of the operating system.
- Several changes were made to the operating system architecture to make this possible (Binderized HALs, HIDL, VTS, VNDK, VINTF, GSI, etc.).

Project Treble

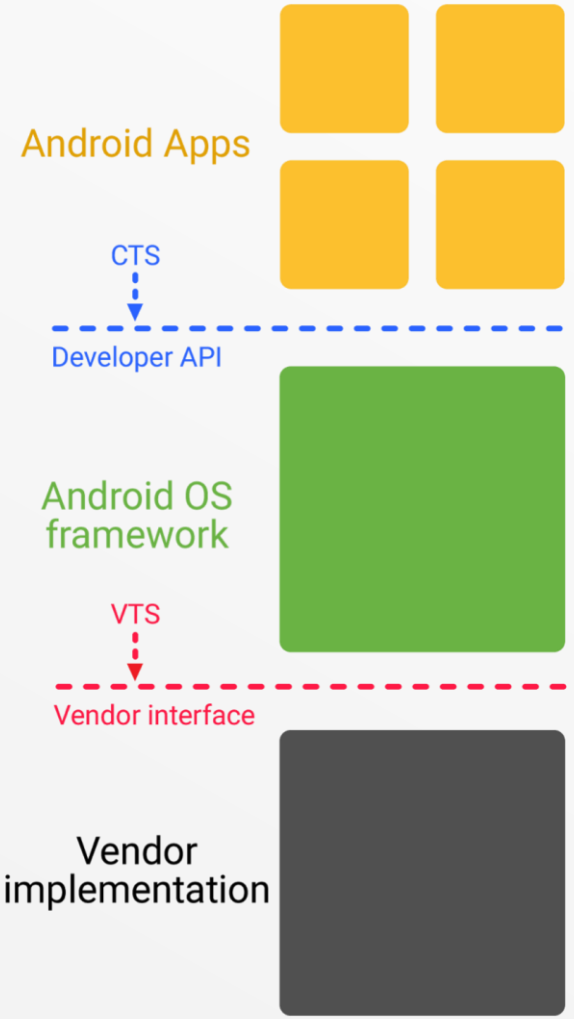


4 years of support

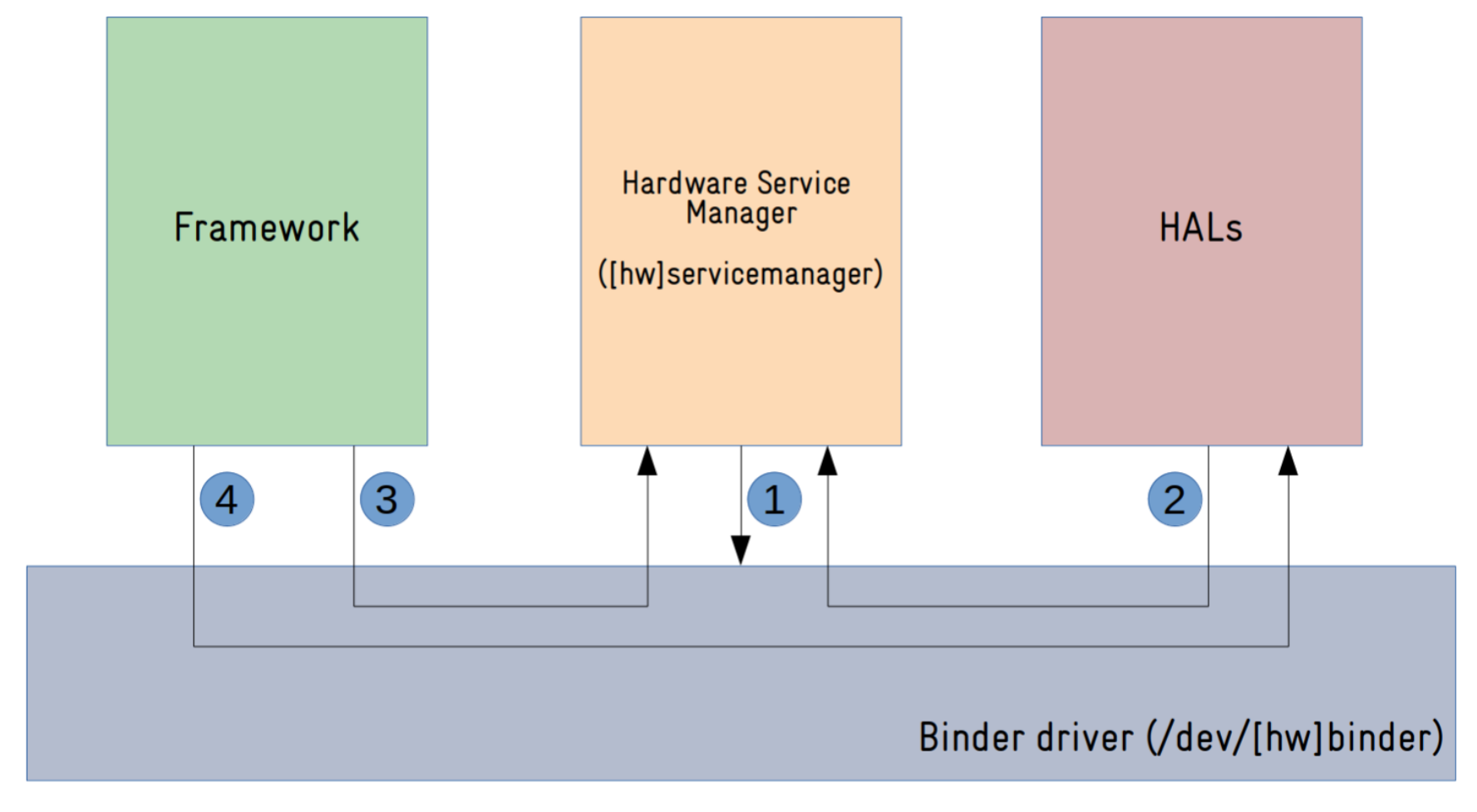


- To stabilize and make the framework code independent from the vendor code, HALs have become services that are consumed via Binder.
- The HAL interface is stable and formally defined through HIDL (HAL Interface Definition Language) or AIDL (Android Interface Definition Language).
  https://source.android.com/devices/architecture/hidl
  https://source.android.com/docs/core/architecture/aidl
- HALs and other customizations are distributed in separate partitions (/vendor, /product, /odm), allowing the /system partition to be updated independently.
- A tool called VTS (Vendor Test Suite) was created to test and validate the implementation of the manufacturer's interface (same concept as the CTS tool for validating the framework).

CTS and VTS


Android Apps
CTS
Developer API
Android OS framework
VTS
Vendor interface
Vendor implementation

Communication with HAL



Other Treble changes

- Standardization of the Linux kernel version used in official Android releases.
- The VNDK (Vendor Native Development Kit) is a set of libraries exclusive to the use of hardware manufacturer code. Executables in the /vendor partition (including HALs) can only link with VNDK libraries.
  https://source.android.com/devices/architecture/vndk
- The VINTF Object (Vendor Interface Object) allows you to specify at compile time (and retrieve at runtime) compatibility information between the Android framework and the vendor implementation.
  https://source.android.com/devices/architecture/vintf

HAL implementation

- Implementing a HAL is laborious and involves different areas of knowledge(hardware, interface provided by the kernel, HAL interface defined by Google, framework operation, etc.).
- For this reason, if possible, choose a hardware device that already has an Android HAL implementation.
- If this is not possible, you will need to study how the HAL works, and depending on the device, read/write a lot of source code!

Documentation and source code

- A general overview of the HAL is documented by Google:

  https://source.android.com/docs/core/architecture

- The HAL interface definition, including some reference implementations, are available in the following AOSP directories:

  hardware/interfaces/
  frameworks/hardware/interfaces/
  system/hardware/interfaces/

- There are also some HAL implementations in the product directories under device/.

Lights HAL (HIDL)

```
$ tree hardware/interfaces/light/2.0/
hardware/interfaces/light/2.0/
├── Android.bp
├── default
│   ├── Android.bp
│   ├── android.hardware.light@2.0-service-lazy.rc
│   ├── android.hardware.light@2.0-service.rc
│   ├── Light.cpp
│   ├── Light.h
│   ├── service.cpp
│   └── serviceLazy.cpp
├── ILight.hal
├── types.hal
└── vts
    └── functional
        ├── Android.bp
        └── VtsHalLightV2_0TargetTest.cpp
```

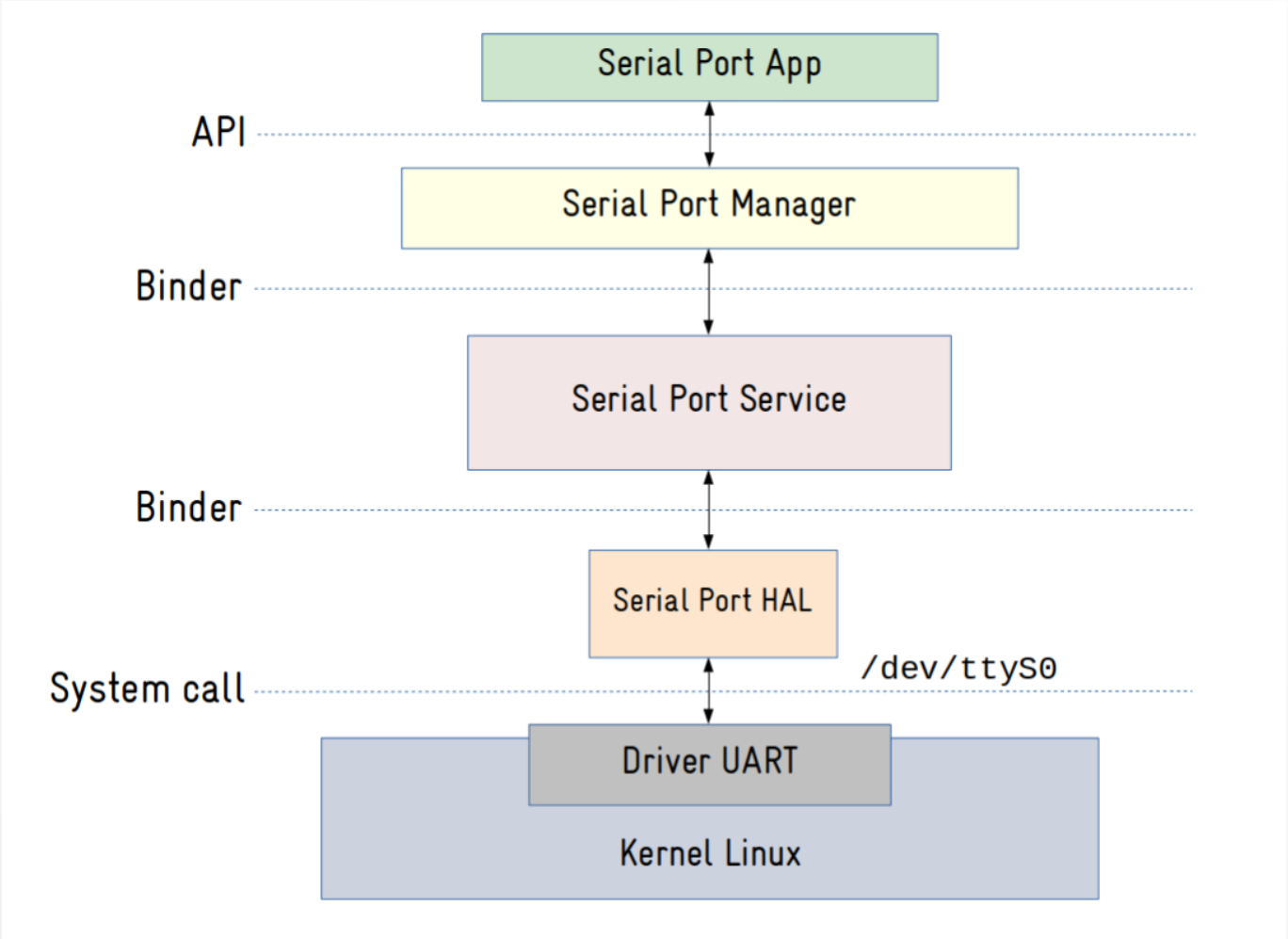
Lights HAL (AIDL)

```
$ tree hardware/interfaces/light/aidl/android/
hardware/interfaces/light/aidl/android/
└── hardware
    └── light
        ├── BrightnessMode.aidl
        ├── FlashMode.aidl
        ├── HwLight.aidl
        ├── HwLightState.aidl
        ├── ILights.aidl
        └── LightType.aidl

$ ls hardware/interfaces/light/aidl/Android.bp
hardware/interfaces/light/aidl/Android.bp

$ tree hardware/interfaces/light/aidl/default/
hardware/interfaces/light/aidl/default/
├── Android.bp
├── Lights.cpp
├── lights-default.rc
├── lights-default.xml
├── Lights.h
└── main.cpp
```

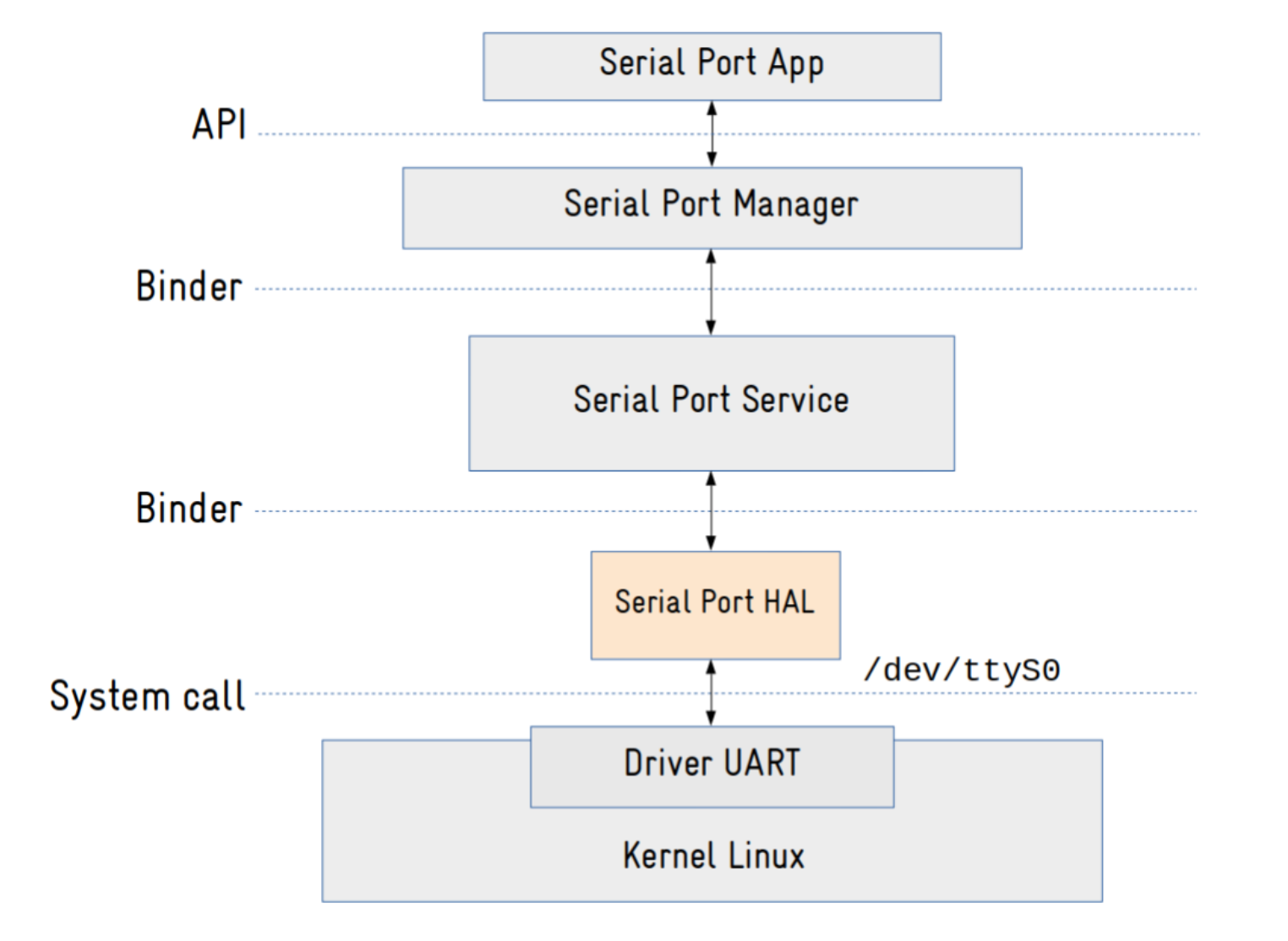
Serial port HAL

- To put all these concepts into practice, we will develop a serial port HAL.
  - The HAL will provide two methods:
    - tx(): data transmission to the serial port.
    - rx(): data reception from the serial port.
- In the next slides, we will see source code excerpts implementing this HAL, whichwill be fully developed in the next training exercise.

Layers to access the serial port



Serial port HAL



Implementing the serial port HAL

- The following steps are necessary to implement the HAL:
  - Step 1: define the HAL interface (HIDL or AIDL).
  - Step 2: implement the HAL (C++, Java or Rust).
  - Step 3: declare the HAL in the device and framework manifest files (VINTF object).
- Although not necessary, we will also implement a HAL test application.

Defining the HAL interface

- It is possible to define the interface of a HAL using the HIDL (Hardware Interface Definition Language) or AIDL (Android Interface Definition Language) languages.
  - HIDL was introduced with Project Treble.
  - AIDL is much older and more mature than HIDL.
- Over time, Google noticed an overlap of functionalities between the languages.
- In Android 10, the main functionalities of HIDL were implemented in AIDL.
- As of Android 11, it became possible to write HALs with AIDL, and the use of HIDL was discontinued. https://source.android.com/devices/architecture/aidl/overview

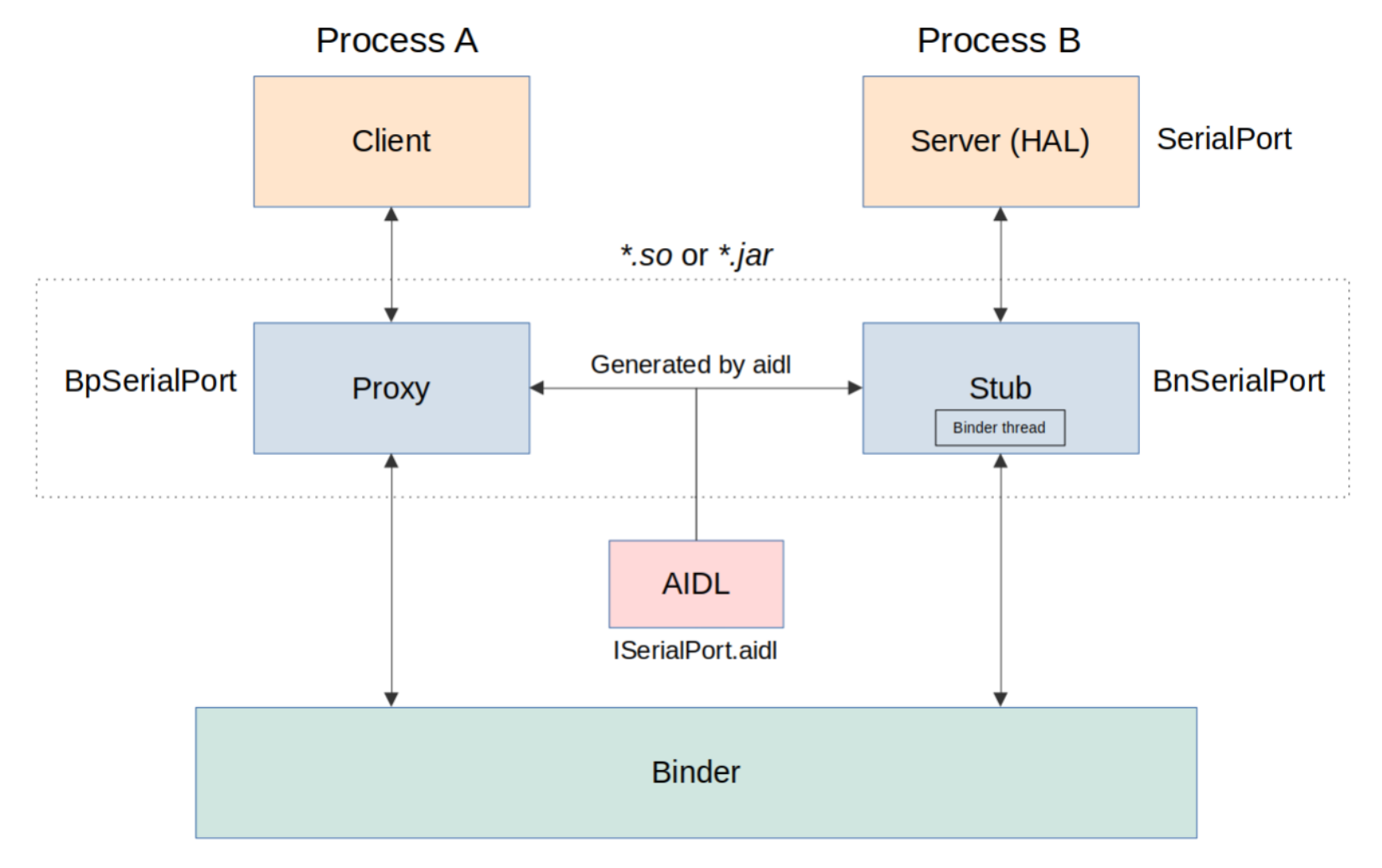
Defining the interface with AIDL

- Defining the serial port HAL interface with the AIDL language:

```
// ISerialPort.aidl
interface ISerialPort {
  void tx(byte data);
  byte rx();
}
```

- The interface definition file (.aidl) will be processed by the aidl tool.
- The aidl tool will generate the HAL communication classes (Proxy and Stub)encapsulated in a shared library (.so or .jar):
  - The Proxy class will be used by the client to send the message.
  - The Stub class will be used by the server (HAL) to receive the message.

Communication architecture with AIDL



Implementing the HAL

- The HAL implementation can be done in C++, Java or Rust.
  https://source.android.com/docs/core/architecture/aidl/aidl-backends
- From the HAL definition, we can implement its interface, extending the Stub class and implementing its methods.
  - The aidl tool is capable of generating a skeleton of this implementation.
- Since the HAL will run inside an Android process, it is necessary to create a program that will instantiate the HAL class and register it in the service manager.
  - A startup file (.rc) is necessary to run the program automatically when Android boots.

Implementing the serialport class

// SerialPort.h

```
#include <aidl/vendor/labworks/serialport/BnSerialPort.h>

class SerialPort : public BnSerialPort {
    ndk::ScopedAStatus tx(int8_t  data) override;
    ndk::ScopedAStatus rx(int8_t* data) override;
};
```

// SerialPort.cpp

```
#include "SerialPort.h"

ndk::ScopedAStatus SerialPort::tx(int8_t data) {
    // TODO: implement method
    return ndk::ScopedAStatus::ok();
}

ndk::ScopedAStatus SerialPort::rx(int8_t* data) {
    // TODO: implement method
    return ndk::ScopedAStatus::ok();
}
```

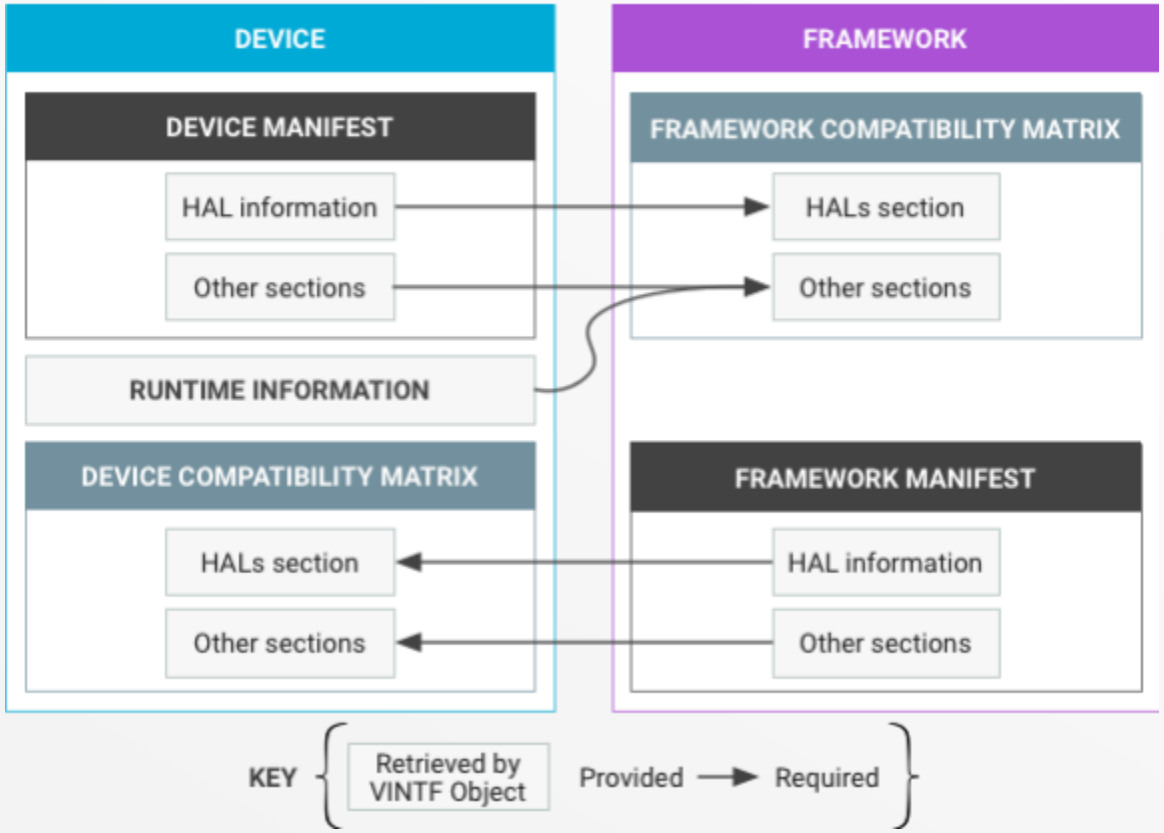
Implementing the HAL program

```
// main.cpp

#include "SerialPort.h"
#include <android/binder_manager.h>
#include <android/binder_process.h>

using ::aidl::vendor::labworks::serialport::SerialPort;

int main() {
    ABinderProcess_setThreadPoolMaxThreadCount(16);

    std::shared_ptr<SerialPort> serialport = ndk::SharedRefBase::make<SerialPort>();

    const std::string instance = std::string() + SerialPort::descriptor + "/default";
    binder_status_t status = AServiceManager_addService(serialport->asBinder().get(),
                                                        instance.c_str());
    CHECK_EQ(status, STATUS_OK);

    ABinderProcess_startThreadPool();
    ABinderProcess_joinThreadPool();

    return EXIT_FAILURE;  // should not reach here
}
```

Manifest files

- To add the HAL to the image, it is recommended to declare it in the product's manifest files through the vendor interface object (VINTF object).
- The VINTF object includes the following files:
  - **Device manifest**: describes the services that the device provides to the framework.
  - **Framework manifest**: describes the services that the framework provides to the device.
  - **Device compatibility matrix**: describes what the device expects from the framework.
  - **Framework compatibility matrix**: describes what the framework expects from the device.
- When implementing a HAL, it may be necessary to populate the device manifest and the framework compatibility matrix.

VINTF OBJECT



Testing the HAL

- To test the HAL, we can create a test client application.
- The application can use the static *fromBinder()* method of the HAL class to obtain a reference to the communication Proxy class.
- With the pointer to the Proxy class, the HAL methods can be invoked directly by the test application.
  - All communication with the HAL via Binder is abstracted by the Proxy class.

HAL test client

```
// client.cpp

int main(int argc, char *argv[])
{
    const std::string instance = std::string() + ISerialPort::descriptor + "/default";
    std::shared_ptr<ISerialPort> serialport;

    serialport = ISerialPort::fromBinder(ndk::SpAIBinder(AServiceManager_checkService(
                                            instance.c_str())));
    if (!serialport) {
        std::cout << "Error requesting serialport service handler!" << std::endl;
        return 1;
    }

    /* TX */
    serialport->tx('0');

    /* RX */
    int8_t data = 0;
    serialport->rx(&data);

    return 0;
}
```

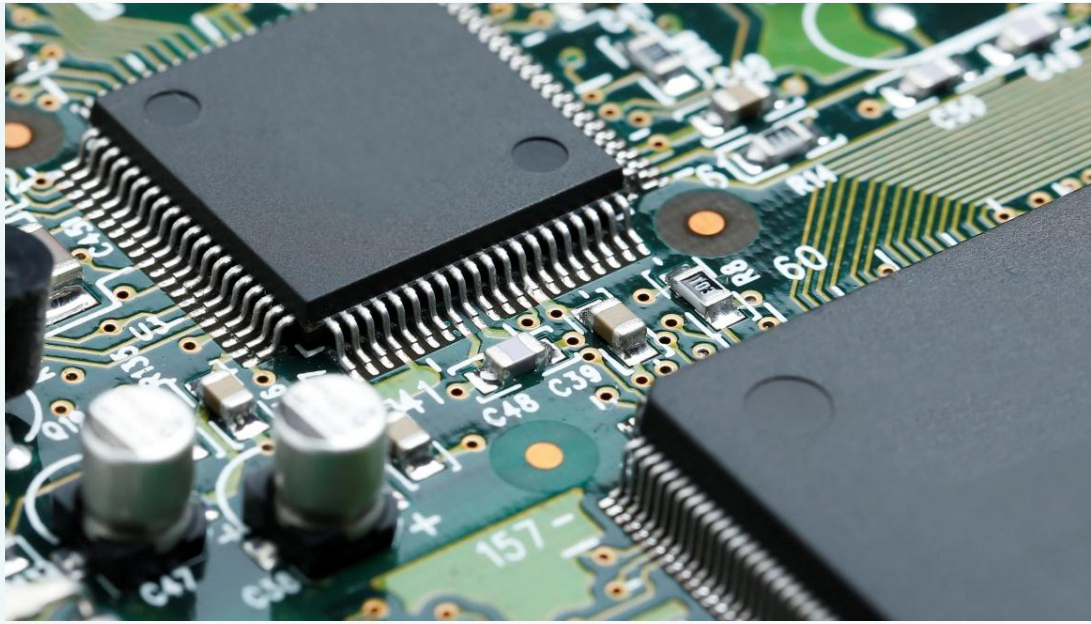
HALS in running

```
# ps -A | grep android.hardware.sensors
system         448     1   33676   6252 binder_wait_for_work 0 S android.hardware.sensors-service

# ps -A | grep android.hardware.bluetooth
bluetooth      387     1   22840   5020 binder_wait_for_work 0 S android.hardware.bluetooth@1.1-s

# lshal | grep android.hardware.bluetooth
DM,FC Y android.hardware.bluetooth@1.0::IBluetoothHci/default                    0/1        38
DM,FC Y android.hardware.bluetooth@1.1::IBluetoothHci/default                    0/1        38

# service list | grep android.hardware.sensors
51      android.hardware.sensors.ISensors/default: [android.hardware.sensors.ISensors]

# logcat | grep android.hardware.bluetooth
06-14 15:00:42.867     0     0 I init    : Parsing file /vendor/etc/init/android.hardware.bluetoo
06-14 15:01:00.839   610   634 I SystemConfig: Reading permissions from /vendor/etc/permissions/a
06-14 15:01:00.860   610   634 I SystemConfig: Reading permissions from /vendor/etc/permissions/a
06-14 15:02:58.981   387   387 I android.hardware.bluetooth@1.1-btlinux: BluetoothHci::initialize
06-14 15:02:58.981   387   387 I android.hardware.bluetooth@1.1-btlinux: openBtHci
06-14 15:02:58.981   387   387 E android.hardware.bluetooth@1.1-btlinux: Unable to open /dev/rfki
06-14 15:02:58.981   387   387 I android.hardware.bluetooth@1.1-btlinux: waitHciDev
06-14 15:02:58.981   387   387 I android.hardware.bluetooth@1.1-btlinux: HCI device ready
```

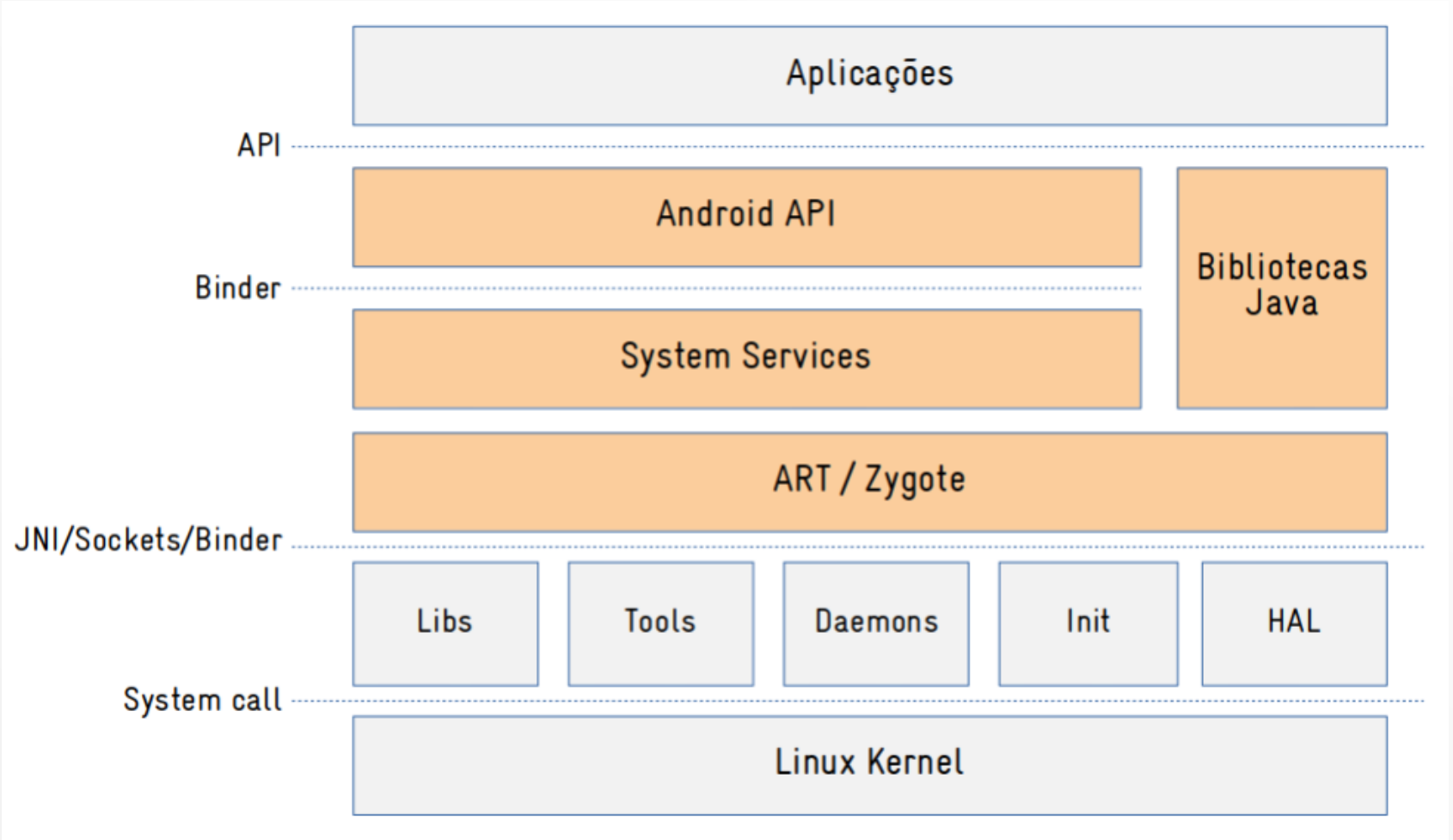
Devices without HAL

- Not every hardware device needs a HAL in Android:
  - Input devices (buttons, mouse, keyboard) are accessed directly via files exported by the kernel in /dev/input/.
  - Network devices are accessed normally via the network interface, via a socket connection or via the netd daemon.
  - Storage devices (SD card) are accessed directly via system calls or via the vold daemon.

Android Framework



Android Framework



Framework

- Framework is the layer responsible for providing services for Android applications, with three main components:
  - Virtual machine runtime (ART/Zygote).
  - System Services (framework services).
  - APIs (provided by the SDK).
- Contains all the "business rules" of the operating system.
- In this section we will focus on framework services (System Services).

System services

- System Services are services responsible for managing access to resources provided by the operating system. Examples:
  - **Power Manager Service**: power management.
  - **Sensor Service**: sensor management.
  - **Location Manager Service**: location management.
  - **Activity Manager Service**: management of application components.
  - **Input Manager Service**: management of input devices.
- Android 13 has over 200 services!

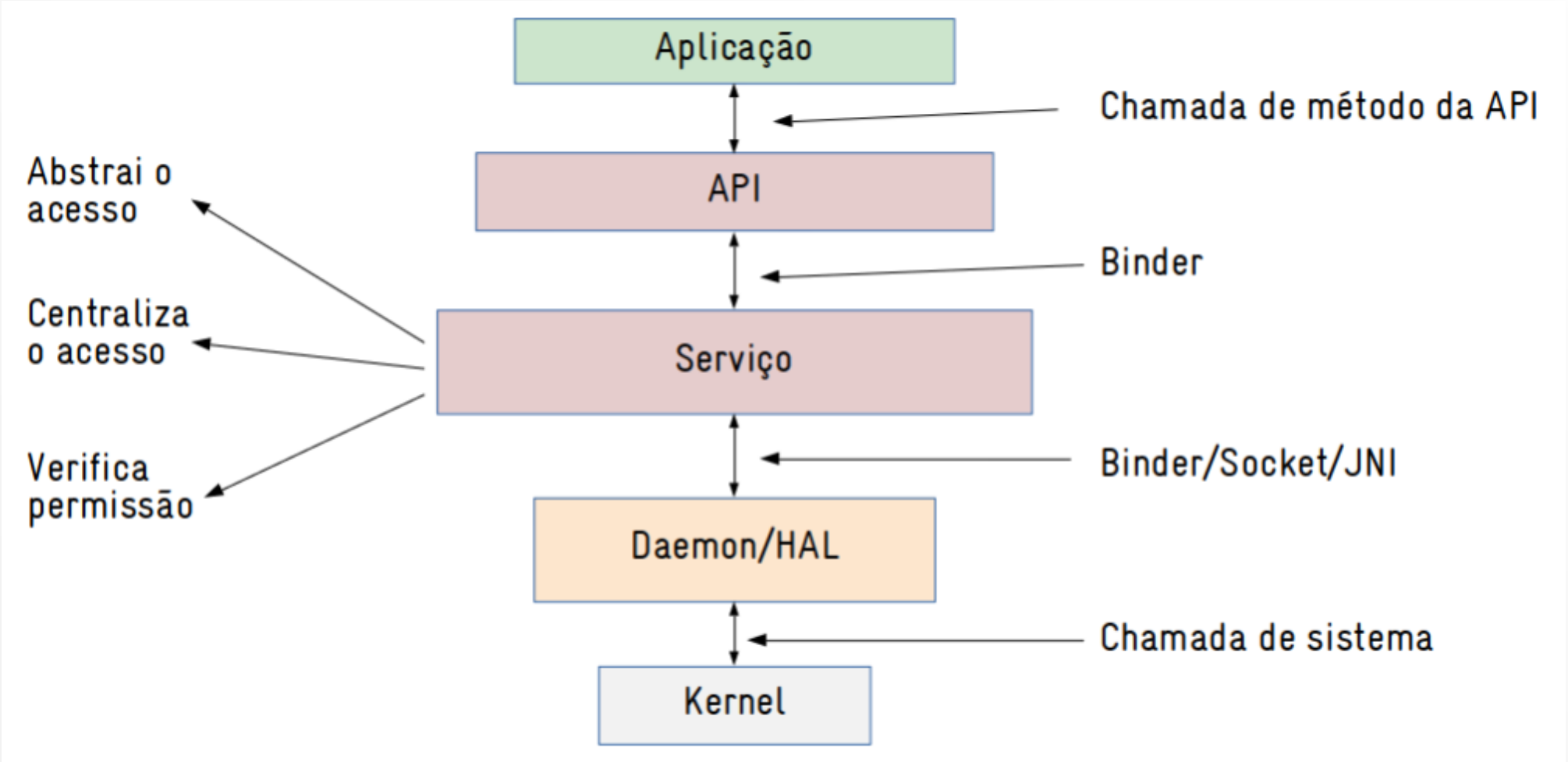
Listing the services

```
# service list
Found 277 services:
0 DockObserver: []
1 SurfaceFlinger: [android.ui.ISurfaceComposer]
2 SurfaceFlingerAIDL: [android.gui.ISurfaceComposer]
3 accessibility: [android.view.accessibility.IAccessibilityManager]
4 account: [android.accounts.IAccountManager]
5 activity: [android.app.IActivityManager]
6 activity_task: [android.app.IActivityTaskManager]
7 adb: [android.debug.IAdbManager]
8 alarm: [android.app.IAlarmManager]
...
267 vold: [android.os.IVold]
268 vpn_management: [android.net.IVpnManager]
269 wallpaper: [android.app.IWallpaperManager]
270 wallpaper_effects_generation: [android.app.wallpapereffectsgeneration.IWallpaperEffectsGenerati
271 webviewupdate: [android.webkit.IWebViewUpdateService]
272 wifi: [android.net.wifi.IWifiManager]
273 wifinl80211: [android.net.wifi.nl80211.IWificond]
274 wifip2p: [android.net.wifi.p2p.IWifiP2pManager]
275 wifiscanner: [android.net.wifi.IWifiScanner]
276 window: [android.view.IWindowManager]
```

Advantages and disadvantages

- Main advantages of the framework's service architecture:
  - Abstracting access to a given system resource (encapsulation).
  - Controlling and centralizing access to the resource (concurrency management).
  - Ensuring security in access to the resource (permission checking).
- Of course, this architecture has its costs:
  - Greater consumption of hardware resources (CPU, memory, etc.).
  - Communication overhead between operating system components.

Service architecture



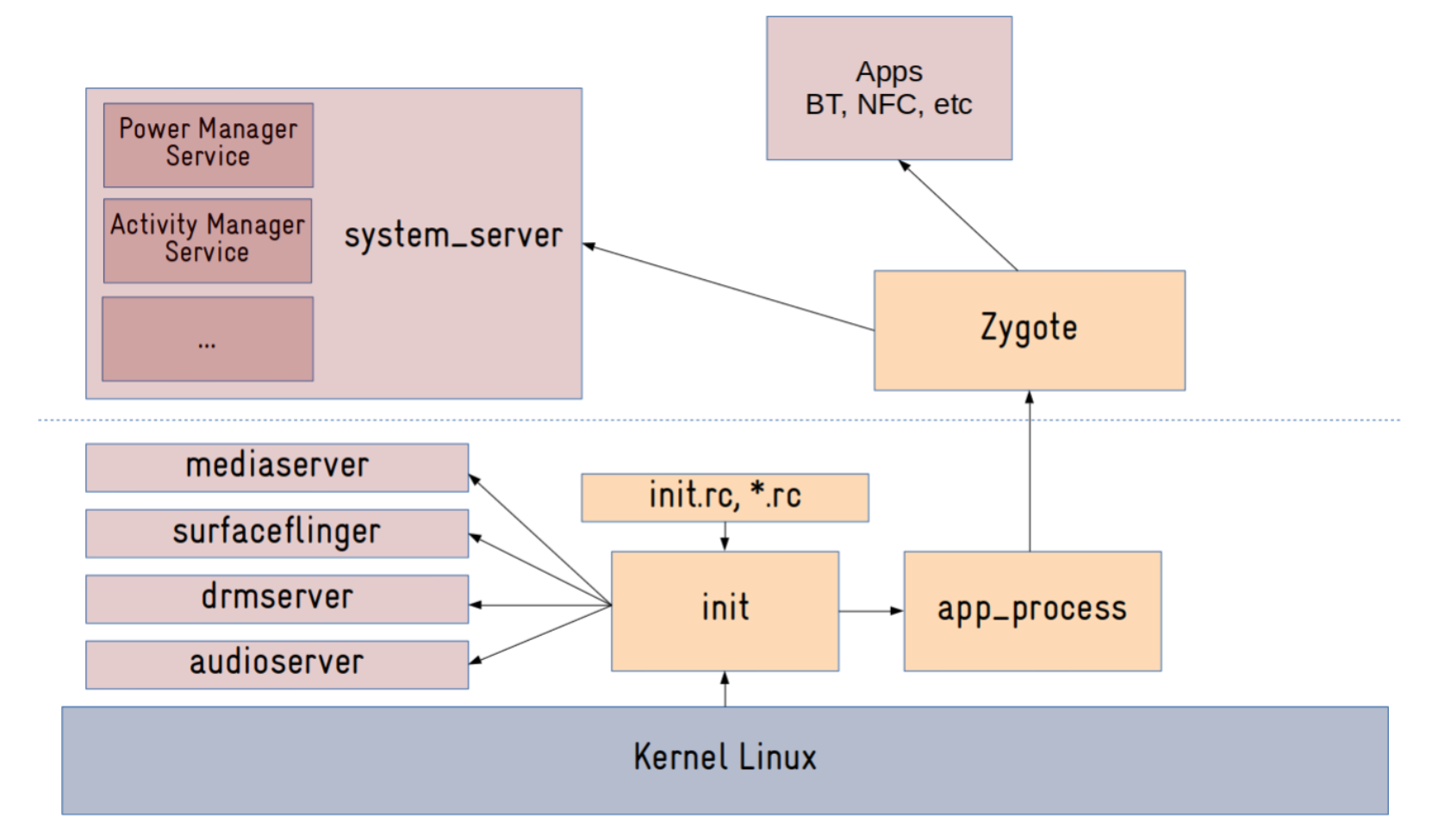
Services and processes

- There is no direct relationship between running processes and framework services.
- Many services are part of the System Server, executed by Zygote when the operating system boots.
- Some services are encapsulated in native processes, executed by init when Android boots (Surface Flinger, DRM Server, Media Server, Audio Server, Camera Server, Gatekeeper, etc.).
- Services can also be encapsulated in Android applications (NFC, Phone, Bluetooth, etc.).

Processes that contain services

- **system_server**: process that contains many framework services (sensor, power,activity, package, location, etc).
- **mediaserver**: Android media player service, responsible for encoding and decoding media.
- **audioserver**: audio service, responsible for interacting with the operating system's sound layer.
- **cameraserver**: camera service.
- **surfaceflinger**: service responsible for composing the graphical interface on the display.
- **drmserver**: service responsible for managing copyrighted content (DigitalRights Management).
- **gatekeeperd**: service responsible for user authentication (pattern, pin, password)in a secure execution environment (TEE - Trusted Execution Environment).
- **com.android.bluetooth**: Bluetooth service, provided by the Bluetooth application.

Service initialization



Services running

```
# ps -A
USER          PID  PPID     VSZ    RSS WCHAN           ADDR S NAME
root            1     0 10982076   9096 do_epoll_+         0 S init
system        240     1 10772980   4368 do_epoll_+         0 S servicemanager
root          405     1 14358672 157028 do_sys_po+        0 S zygote
audioserver   453     1 11045844 19236 binder_wa+         0 S audioserver
system        456     1 10929652 23612 do_epoll_+         0 S surfaceflinger
drm           489     1   34488   6500 binder_wa+         0 S drmserver
cameraserver  492     1 11084840 18608 binder_wa+         0 S cameraserver
media         498     1   96592  22900 binder_wa+         0 S mediaserver
system        523     1 10853128   4812 binder_wa+         0 S gatekeeperd
system        648   405 15350784 313120 do_epoll_+        0 S system_server
u0_a78       1238   405 13983844 218616 do_epoll_+        0 S com.android.systemui
bluetooth    1406   405 13963444 127520 do_epoll_+        0 S com.android.bluetooth
secure_elem+ 1426   405 13786920 92552 do_epoll_+         0 S com.android.se
radio        1439   405 13878448 143984 do_epoll_+        0 S com.android.phone
system       1453   405 13865004 139252 do_freeze+        0 S com.android.settings
u0_a74       1539   405 13946280 207184 do_epoll_+        0 S com.android.launcher3
u0_a84       1720   405 13806908 103640 do_epoll_+        0 S android.ext.services
u0_a63       1745   405 13794448 106320 do_freeze+        0 S com.android.deskclock
...
```

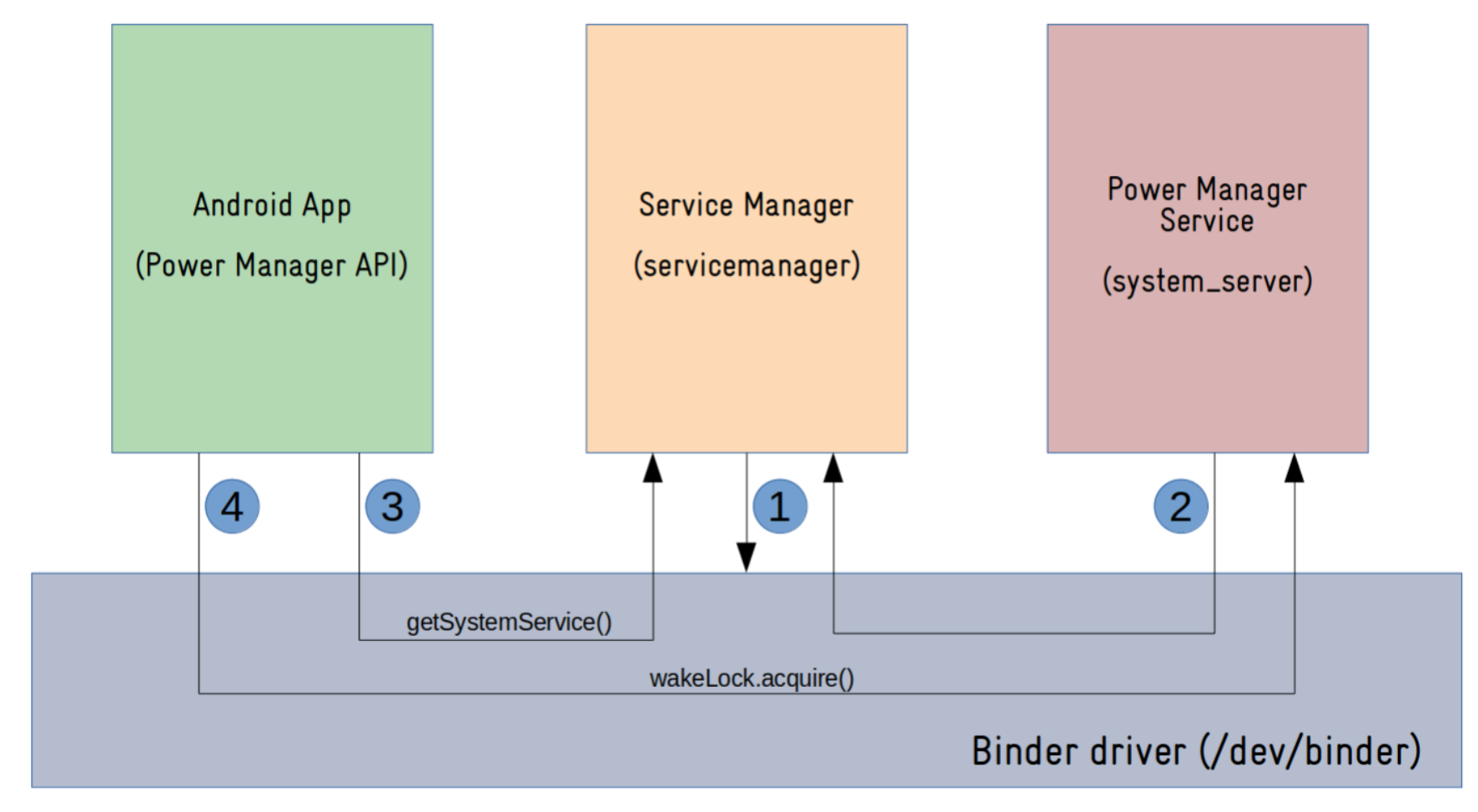
Communication with services

- Communication with framework services is performed through Binder.
- When Android starts up, the Service Manager (servicemanager) registers with the Binder (/dev/binder) as the communication context manager.
  - During the rest of the startup, services register with the Service Manager.
- To communicate with a system service, an Android application requests a reference (handle) to the service from the Service Manager.
  - With the reference, the application can invoke the service methods.
- This communication mechanism is completely transparent to the application developer, who only needs to know the Android APIs.

Example: Power Manager Service

```
// get handle to the Power Manager Service
PowerManager pm = (PowerManager) getSystemService(POWER_SERVICE);

// create a wakelock object
PowerManager.WakeLock wakeLock = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK, "myWakeLock");

// request a wakelock (this will generate a RPC to the service)
wakeLock.acquire();

// do the work here
...
...
...

// release the wakelock (this will generate a RPC to the service)
wakeLock.release();
```

Diagram: Power Manager Service



Services documentation

- There is no formal documentation for the framework services, so to understand how they work you may need to read a lot of source code!
- For example, the System Server entry point is available at:

  frameworks/base/services/java/com/android/server/SystemServer.java

- This could be a good place to start exploring the thousands of lines of code for each of the framework services!

Applications started by services

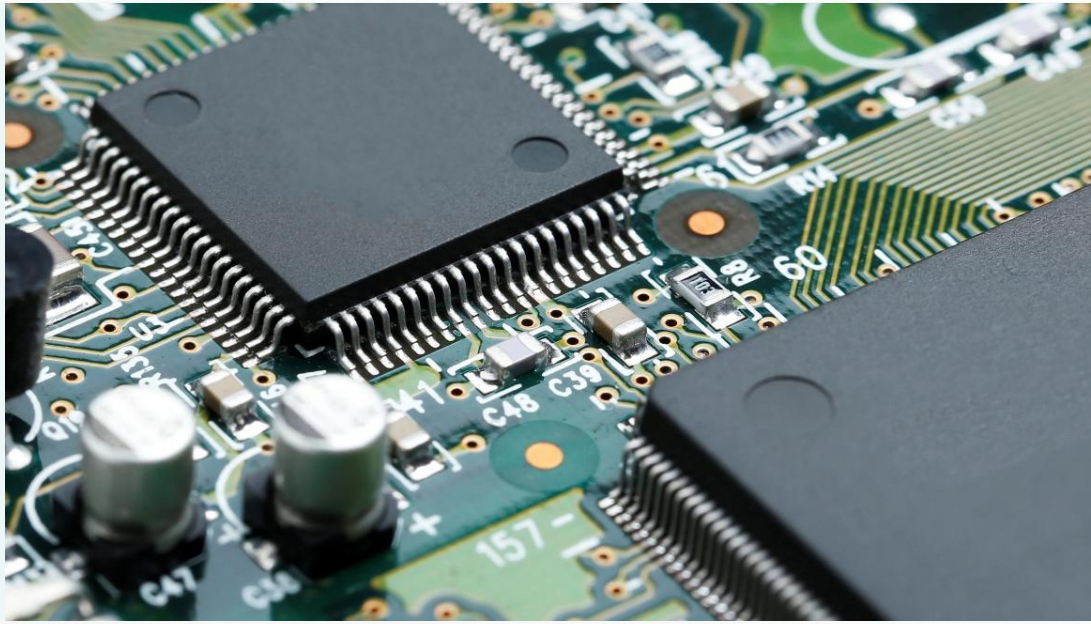
- The framework services are responsible for starting Android applications automatically when the system boots:
  - The Activity Manager Service automatically starts applications that have the *android:persistent="true"* attribute.
  - The Activity Manager Service sends an Intent of the *BOOT_COMPLETED* type, notifying the system about the end of the boot process.
  - The Input Method Manager Service starts all applications that have the *android.view.InputMethod* Intent filter.
  - The Activity Manager Service sends an Intent of the *CATEGORY_HOME* type to run the application registered as a Launcher in the system.

Command line tools

- **am**: allows you to communicate directly with the Activity Manager Service.
- **pm**: allows you to communicate directly with the Package Manager Service.
- **sm**: allows you to communicate directly with the Storage Manager Service.
- **wm**: allows you to interact with the Window Manager Service.
- **ime**: allows you to communicate directly with the Input Method Manager Service.
- **input**: allows you to inject input events into the system (keyboard, touchscreen, etc.).
- **svc**: allows you to communicate with several other services (Power, USB, NFC, etc.).
- **cmd**: tool that interacts with a framework service via its command interface (media_session example implementation)
- **service**: implements a generic interface that allows you to interact with any framework service.
- **dumpsys**: capable of extracting debug information from a framework service, if the service has implemented the dump() method.

Modifying the framework

- The Android framework is quite monolithic, and customizing it (e.g., removing or disabling a service) may require changes to the source code.
- Changing the framework source code can bring some challenges, including:
  - Possibility of breaking the API or causing a regression.
  - Difficulty in certification (may not pass the CTS).
  - Efforts to maintain and port changes to other versions of Android.
- If possible, avoid changing the framework source code.
  - It is possible to extend the behavior of the framework (e.g., add a new service or API) without directly changing the AOSP source code.

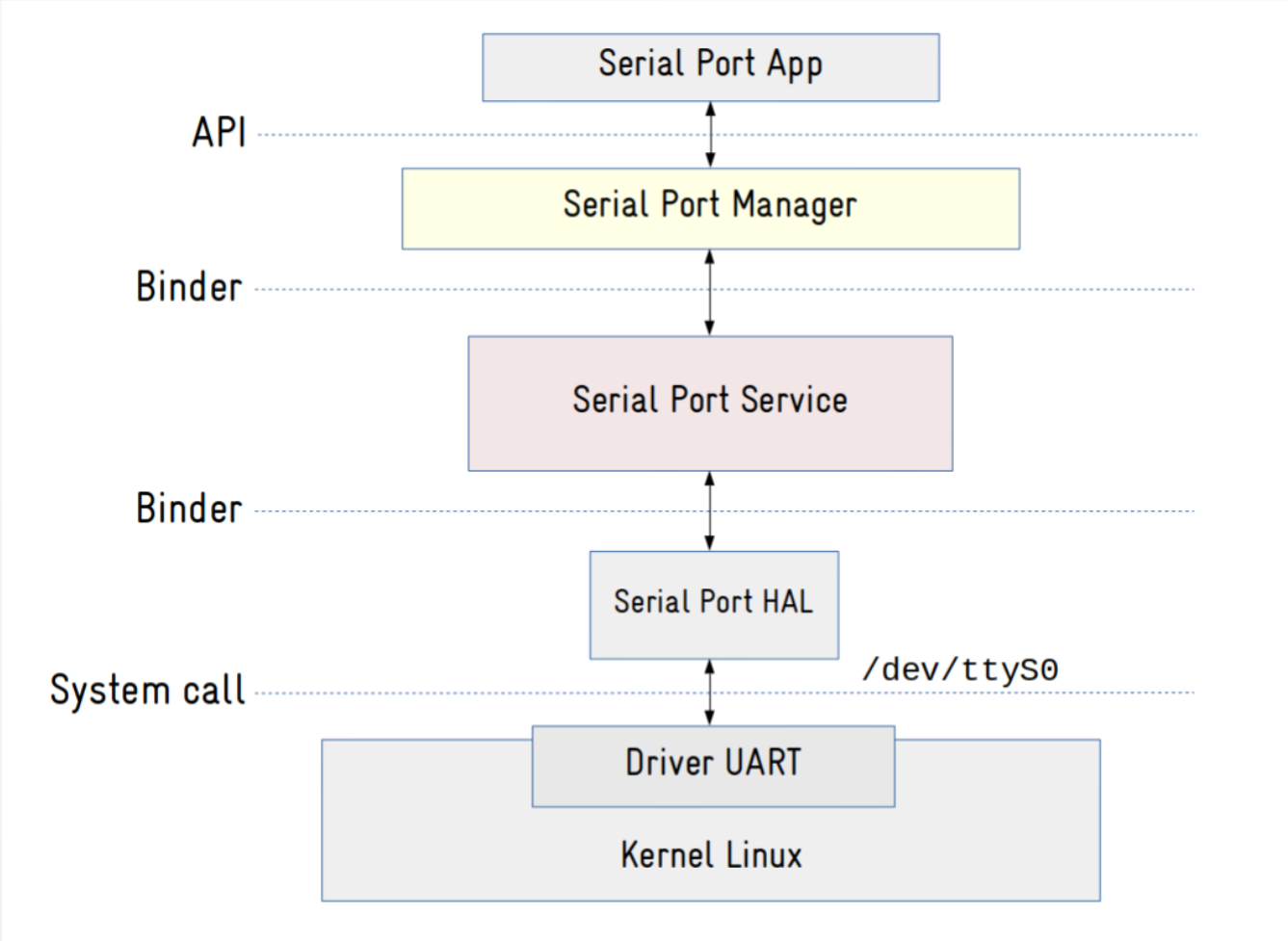


Extending the framework

Extending the framework

- In some situations, for example in embedded systems, it may be necessary to access a certain type of hardware device not supported by the Android API(GPIO, serial port, E2PROM, etc.).
- In the absence of an API, the Android architecture and security model do not allow an application to directly access a hardware device.
- In this case, the best solution is to extend the Android framework to add support for a new hardware device.

Service and API



The application

- The idea is to develop an Android application that can use a simple API to access the serial port.

```
SerialPortManager sp = SerialPortManager.getInstance();
sp.tx(ch);
```

- We also want to control the permission to access the serial port. In order for the application to access the service, it will need to request permission in the manifest file:

```
<uses-permission
        android:name="vendor.labworks.serialportservice.permission.SERIALPORT">
</uses-permission>
```

Implementing the service

- The implementation of a framework service can be divided into four steps:
  - Define the service interface (AIDL).
  - Implement the service (Java).
  - Implement the service's native code to communicate with the HAL (C++).
  - Implement the Manager class (API).

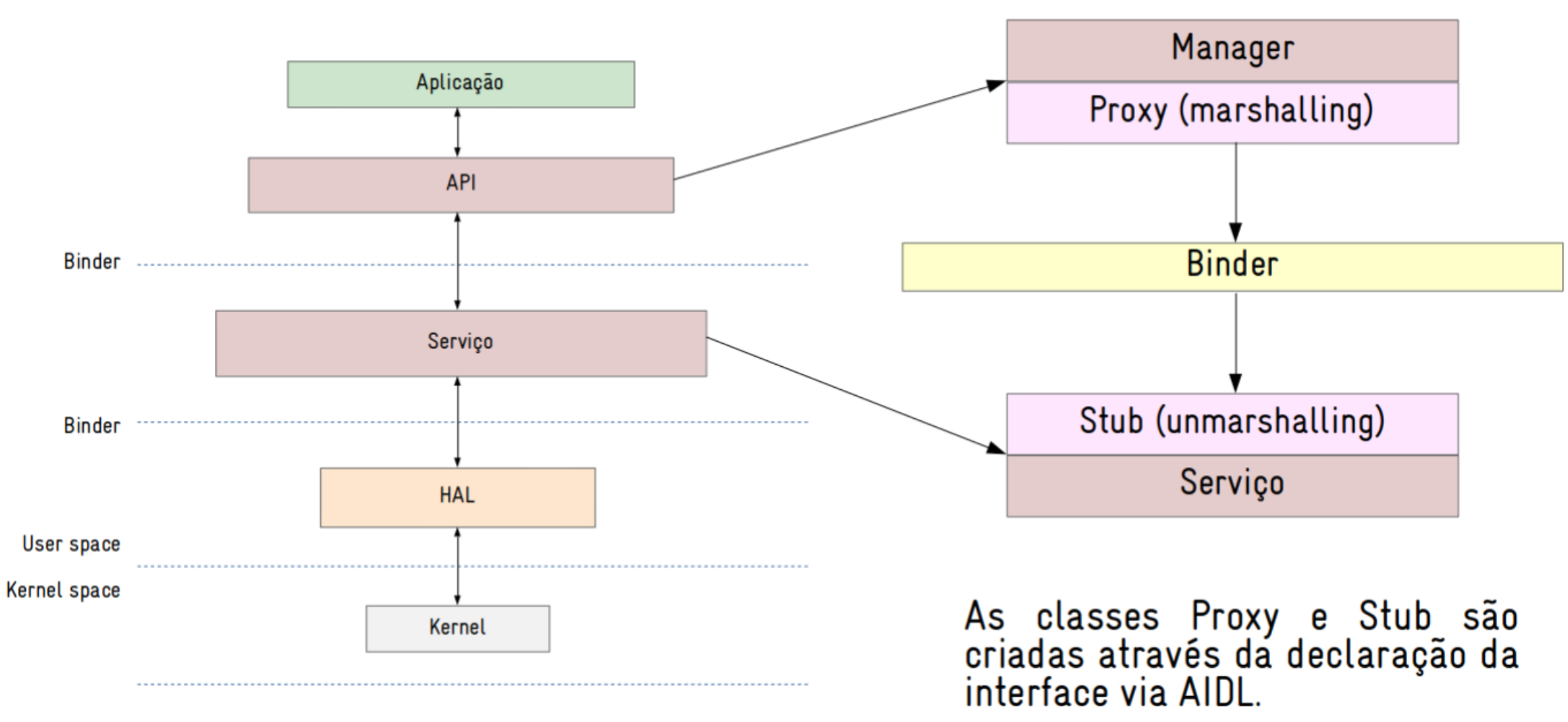
1. Defining the service interface

- The interface of a framework service can be defined through AIDL.

```
interface ISerialPortService {
  void tx(byte data);
  byte rx();
}
```

- A tool called aidl will be responsible for converting the .aidl file into two Java classes:
  - Proxy: used by the client (API) to send requests.
  - Stub: used by the service to receive requests.
- More information about AIDL in the Google documentation:
  https://developer.android.com/guide/components/aidl.html

Proxy and stub classes



2. Implementing the service

- To implement the service, simply extend the Stub class of the interface and implement its methods.
- The service will also be responsible for performing access permission checks.
- To register and run the service, there are a few possibilities:
  - Add the service to the SystemServer source code (not recommended).
  - Create a separate application (approach adopted in the training).

Service source code

```
// SerialPortServiceImpl.java

class SerialPortServiceImpl extends ISerialPortService.Stub {
    public static final String SERIALPORT_PERM =
        "vendor.labworks.serialportservice.permission.SERIALPORT";
    private final Context mContext;

    public SerialPortServiceImpl(Context context) {
        mContext = context;
    }

    public void tx(byte data) {
        mContext.enforceCallingOrSelfPermission(SERIALPORT_PERM, null);
        nativeTx(data);
    }

    public byte rx() {
        mContext.enforceCallingOrSelfPermission(SERIALPORT_PERM, null);
        return nativeRx();
    }

    private static native void nativeTx(byte data);
    private static native byte nativeRx();
}
```

Application source code

```
// SerialPortService.java

package vendor.labworks.serialportservice;

import android.app.Application;
import android.os.ServiceManager;
import android.util.Log;

public class SerialPortService extends Application {
    private static final String SERVICE_NAME = "serialport";
    private SerialPortServiceImpl serviceImpl;

    public void onCreate() {
        super.onCreate();
        Log.i("SerialPortService", "Starting service...");
        serviceImpl = new SerialPortServiceImpl(getApplicationContext());
        ServiceManager.addService(SERVICE_NAME, serviceImpl);
    }

    public void onTerminate() {
        super.onTerminate();
    }
}
```

Implementing native code with JNI

- Services that manage access to a hardware device typically use JNI to communicate with the hardware via HAL.

  $ ls frameworks/base/services/core/jni

- JNI (Java Native Interface) is a development framework that allows Java code running in a virtual machine to access native code running outside the VM (C, C++, etc.).

  https://en.wikipedia.org/wiki/Java_Native_Interface

- Some tips on how to work with JNI on Android:

  https://developer.android.com/training/articles/perf-jni

Native source code of the service

```
static std::shared_ptr<ISerialPort> serialport = nullptr;

JNIEXPORT void JNICALL
  Java_vendor_labworks_serialportservice_SerialPortServiceImpl_nativeTx
  (JNIEnv *, jclass, jbyte data) {
    if (serialport == nullptr) {
        const std::string instance = std::string() + ISerialPort::descriptor + "/default";
        serialport = ISerialPort::fromBinder(ndk::SpAIBinder(AServiceManager_checkService(instanc
    }
    serialport->tx(data);
}

JNIEXPORT jbyte JNICALL
  Java_vendor_labworks_serialportservice_SerialPortServiceImpl_nativeRx
  (JNIEnv *, jclass) {
    if (serialport == nullptr) {
        const std::string instance = std::string() + ISerialPort::descriptor + "/default";
        serialport = ISerialPort::fromBinder(ndk::SpAIBinder(AServiceManager_checkService(instanc
    }
    int8_t data = 0;
    serialport->rx(&data);
    return data;
}
```

Implementing the API(Manager Class)

- The Manager classes are an abstraction for accessing the framework services (Facade Design Pattern).
- These classes implement the API for the application developer to consume the service interface without worrying about the details of communication via Binder (LocationManager, SmsManager, SensorManager, etc.).
- To implement the Manager class, simply invoke the service methods using the Proxy class generated by the AIDL.
- The Manager class can be implemented and distributed in a separate Java library via the Platform Library or Java SDK Library.

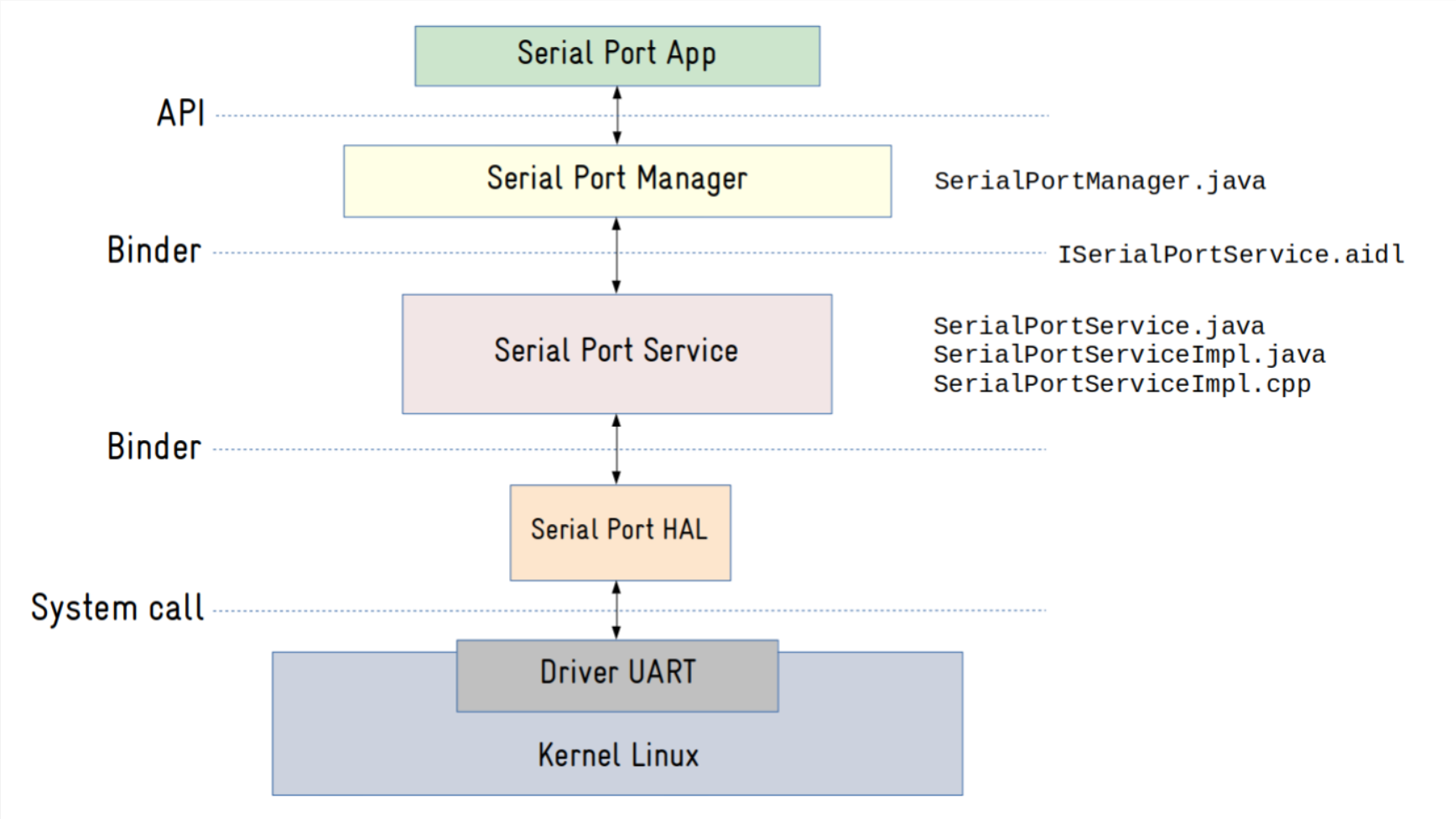
  device/sample/frameworks/PlatformLibrary/README.txt

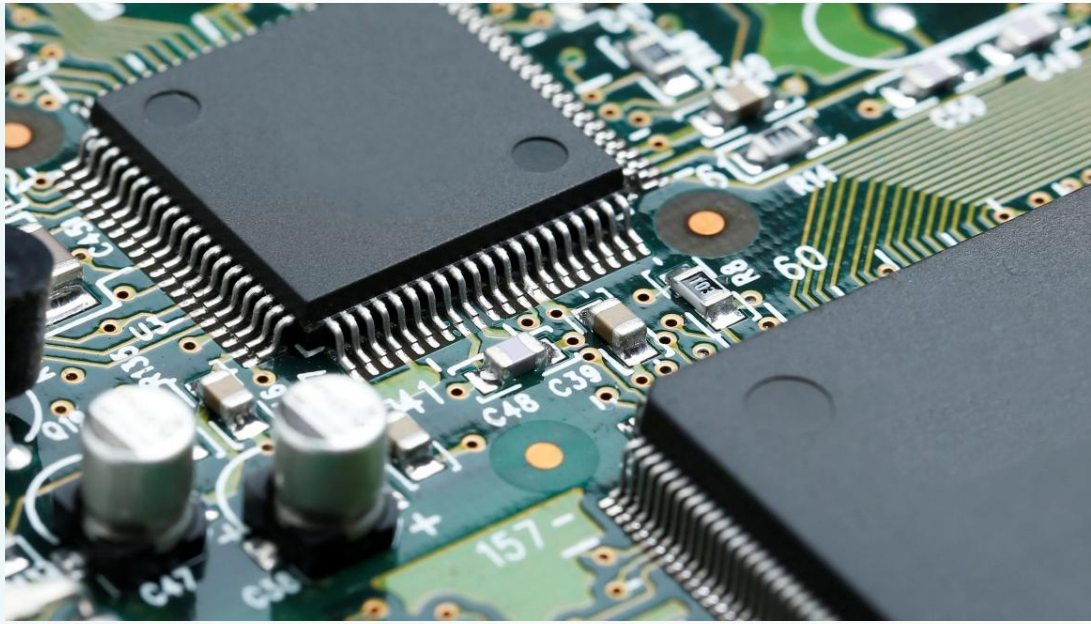
  https://source.android.com/devices/architecture/java-library

Manager Class source code

```
// SerialPortManager.java

public class SerialPortManager {
    private static ISerialPortService service;
    ...
    private SerialPortManager() {
        IBinder binder = ServiceManager.getService(SERVICE_NAME);
        service = ISerialPortService.Stub.asInterface(binder);
        ...
    }
    public static SerialPortManager getInstance() {
        if(instance == null) {
            instance = new SerialPortManager();
        }
        return instance;
    }
    public void tx(byte data) throws RemoteException {
        service.tx(data);
    }
    public byte rx() throws RemoteException {
        return service.rx();
    }
}
```

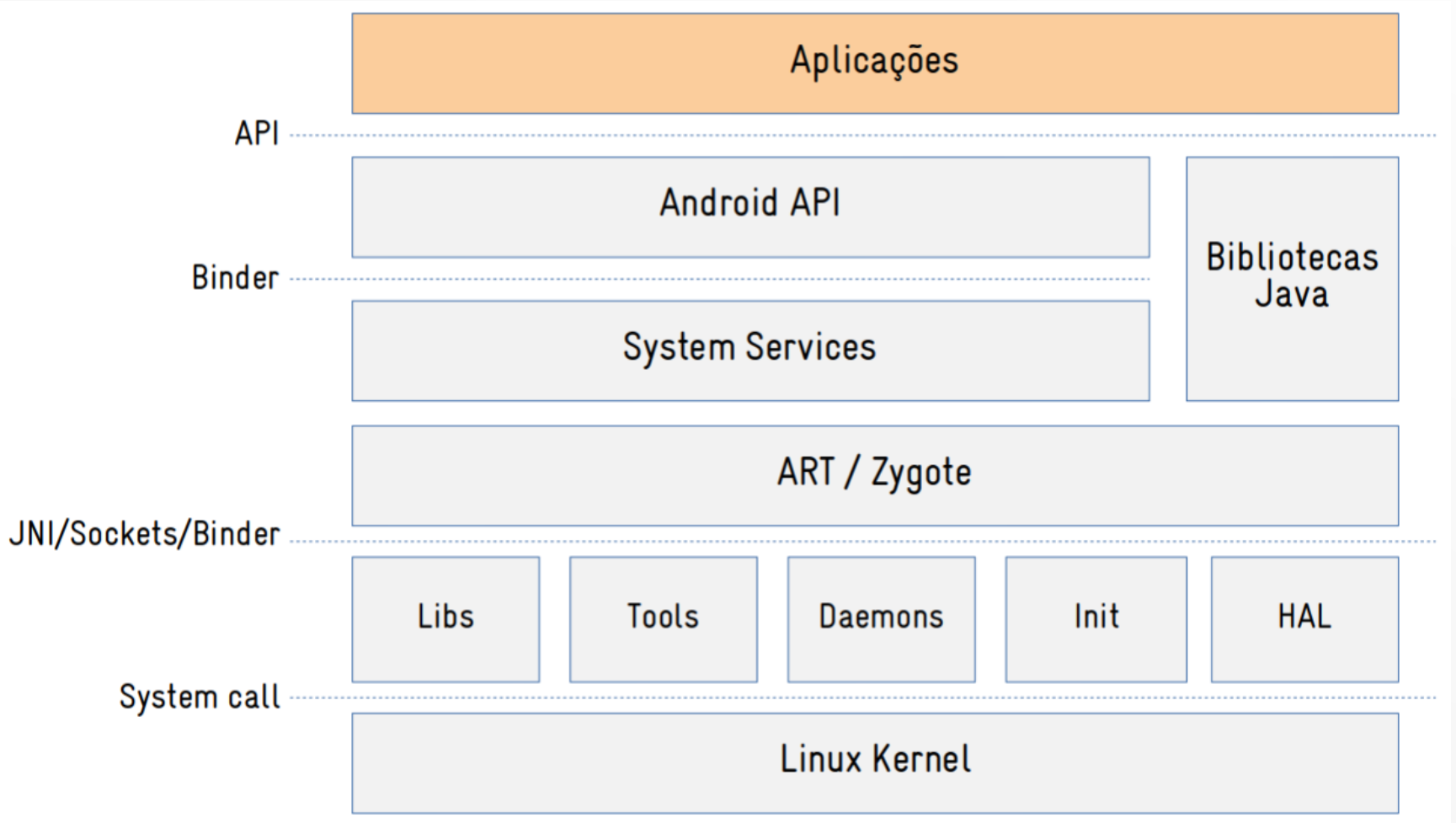
Final implementation





Android Applications

Android Applications



Android Applications

- Everything we have seen so far has the main objective of providing a complete infrastructure for the development of Android applications.
- Android applications are written in Java or Kotlin, compiled in a format called DEX (Dalvik Executable) and executed in the ART virtual machine.
  - They can also be written in native code (C++) through the NDK.
  - They are packaged in files with the extension .apk or .aab (Android App Bundle).
- They can be installed through Google Play or manually (side-loaded) via ADB or any file manager.

Android API

- All Android application development is based on the API provided by Google.
- This API contains two large sets of classes:
  - Standard Java classes (java.*, javax.*, etc.).
  - Android-specific classes (android.*, dalvik.*).
- All Android API documentation is available on the project website.

  https://developer.android.com/reference/packages
- Several services, frameworks, and libraries are available to assist and increase developer productivity (Android Jetpack, Firebase, Flutter, etc.).

Android Application components

- Android applications are composed of a combination of 4 types of components:
  - Activities.
  - Services.
  - Broadcast receivers.
  - Content providers.

Activities

- Activities represent the user interface screens (similar in concept to windows on desktop systems).

  https://developer.android.com/guide/components/activities

- An Android application usually has several Activities, and one of the Activities is defined as the main one, which is presented to the user when the application is executed by the Launcher.
- An application can request the use of an Activity from another application, if it has permission.
  - For example, a web browsing application can request the use of the Activity of an email sending application to share a link via email.

Services

- Services are components that run in the background and have no user interface.

  https://developer.android.com/guide/components/services

- They are used to perform operations in the background or provide services to other applications (e.g. audio playback, server synchronization, etc.).
- Be careful not to confuse them with framework services or init services!

Broadcast receivers

- Broadcast receivers are components that allow you to receive events from the system or an application.

  https://developer.android.com/guide/components/broadcasts

- They have the same concept as a signal or interrupt handling routine(publish-subscribe design pattern).
- Examples of broadcast events:
  - Alarm expired.
  - Screen turned off.
  - Airplane mode was enabled.
  - System boot process completed.

Content providers

- Content providers are components that allow applications to abstract and control access to data.

  https://developer.android.com/guide/topics/providers/content-providers

- An application can use a content provider when it wants to abstract the storage of information and share data with other applications.
  - For example, information about a user's contacts is stored in a content provider, and any application (with access permission) can access this data.
- When creating a content provider, the developer chooses how he wants to store the data, whether locally (SQlite, XML, text, etc.) or on a remote server.

Manifest file

- Every application has a manifest file (AndroidManifest.xml), whose purpose is to declare information about the application, including:
  - Available components (activities, services, broadcast receivers, etc.).
  - Permissions required to run the application.
  - Minimum API level required by the application.
  - Hardware characteristics (display, camera, etc.).
- Full documentation on the project website.

  https://developer.android.com/guide/topics/manifest/manifest-intro

Example of manifest file

```
<manifest package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
                 android:label="@string/app_name" >
        <activity android:name="NotesList"
                  android:label="@string/title_notes_list">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Communication between components

- An application can contain one or more components.
- An application can launch any component, including components from other applications.
- Components communicate with each other through a message-exchange mechanism called **intent**.

Intent

- An intent is an abstraction that serves as a communication mechanism between the components of an Android application.

  https://developer.android.com/reference/android/content/Intent

- An intent basically describes the intention to perform some action (start a service, open an activity, send a broadcast, etc.).
  - For example, an application's activity can send an intent to view a PDF file.
- The intent contains two main pieces of information:
  - **action**: the operation you want to perform (view, edit, etc.).
  - **data**: data associated with the operation in URI (Uniform Resource Identifier) format.

Starting and resolving intents

- Activities, Services, and Broadcast receivers are started through intents.
- Components declare which intents they can handle through the intent-filter attribute in the application manifest file.
- Internally, intents are implemented via Binder and resolved by the ActivityManager Service.
- More information about working with intents is available on the project website.

  https://developer.android.com/guide/components/intents-filters.html

Example of using intent

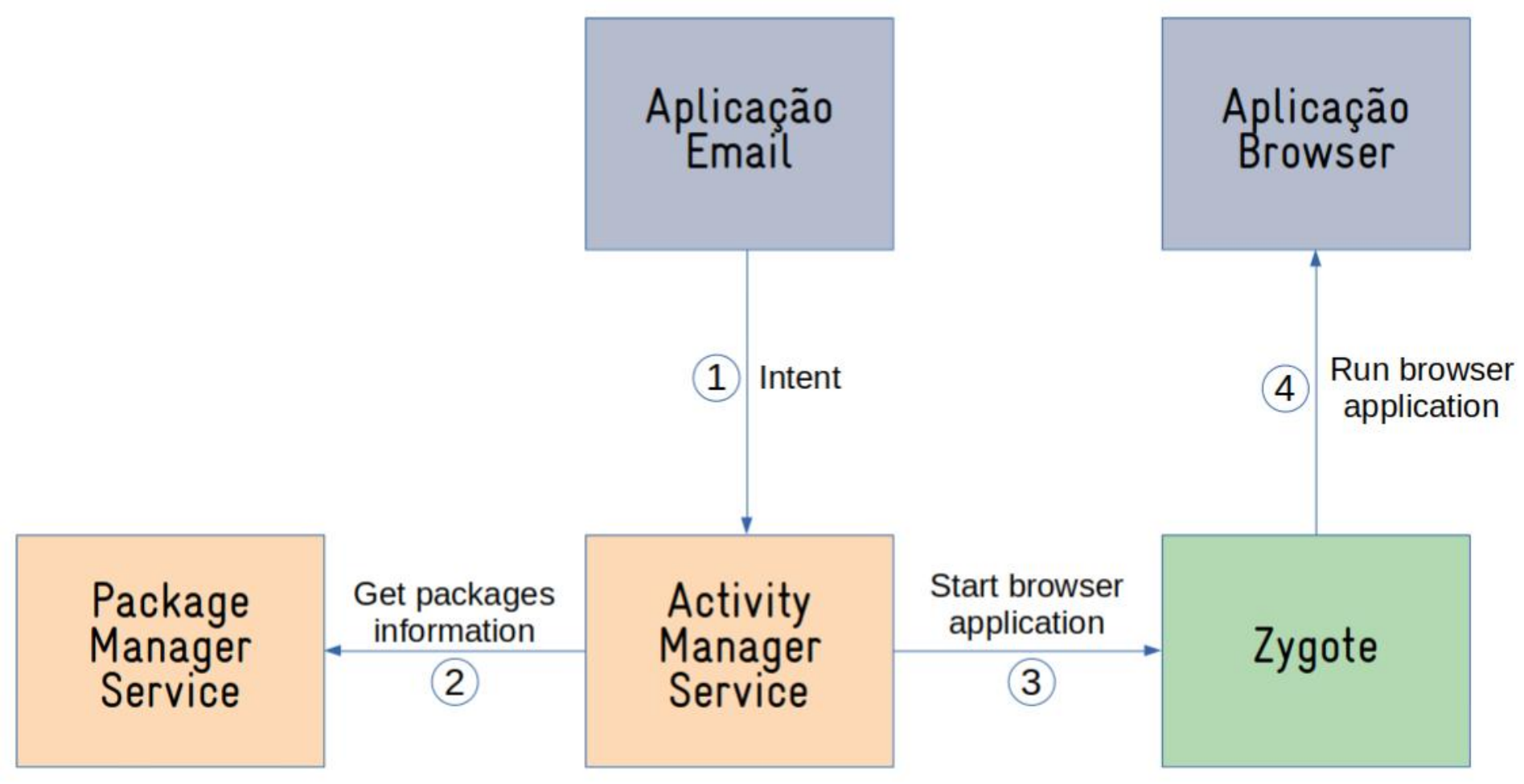
- An application can register an activity as a web browser with the following code in the manifest file:

```
<!-- AndroidManifest.xml -->
<activity android:name=".BrowserActivitiy"
        android:label="@string/app_name">
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <data android:scheme="https"/>
</intent-filter>
</activity>
```

- The following code can be used to start this activity:

```
Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse("https://sergioprado.blog"));
startActivity(i);
```

Intent resolution



Process in Android

- By default, all components of an Android application run in the same process.
- When it is necessary to start a new component:
  - If the process corresponding to the component is not already running, a new process will be created.
  - Otherwise, the component will be started inside the running process.
- It is possible to change this behavior and start a component in a new process by declaring the *android:process* attribute in the application manifest file.

https://developer.android.com/guide/topics/manifest/activity-element#proc

Execution thread

- An Android application has (by default) only one execution thread, so all components and interactions with the user are performed sequentially.
- For this reason, an I/O operation or longer processing in the main thread can crash the application!
- If the application interface blocks for a while (5 seconds by default) the system will display the message "Application Not Responding" (ANR).
- For this reason, it is advisable to execute longer operations in a separate thread.

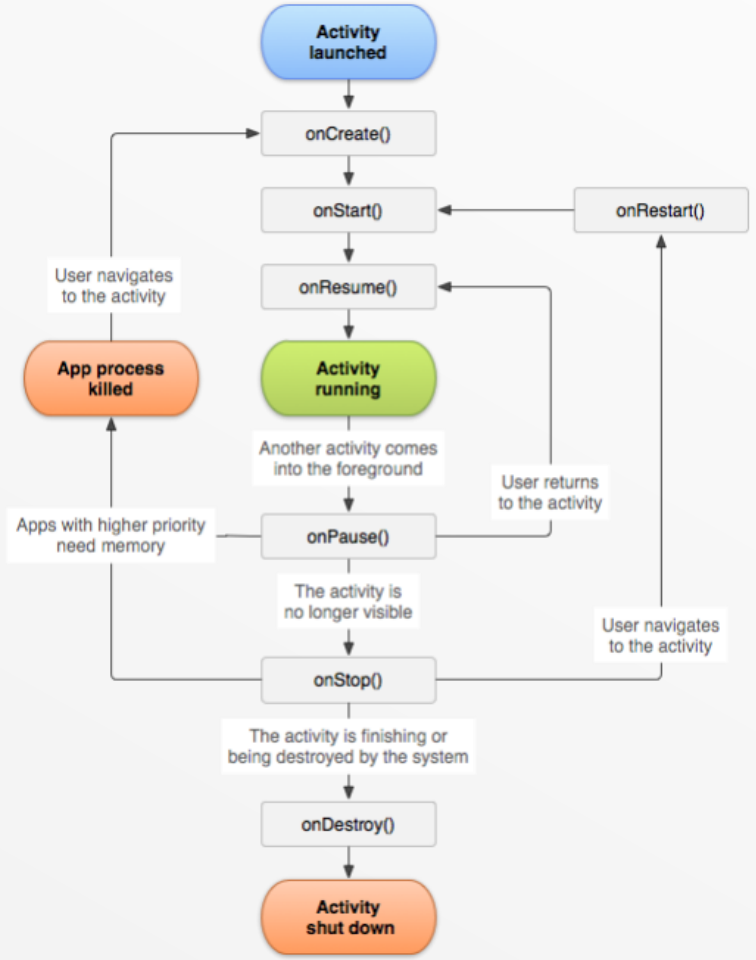
Threads on Android

- In older versions of the Android API, AsyncTask was the recommended class for background operations, especially those that required interactions with the graphical interface.
  - This API has been deprecated in more recent versions of Android.
- There are currently a few options for background processing, including the class WorkManager, the standard Java class java.util.concurrent, and coroutines (for Kotlin code).
- The project website has very complete documentation on background task processing in Android.
  https://developer.android.com/guide/background

Managing running processes

- A central point of Android is that the user does not need to worry about managing the applications and processes that are running.
- The user can start as many applications as they want and switch between them as needed.
  - It is the operating system that manages the life cycle of the applications!
- But after a certain time, several processes may be running, increasing the consumption of resources (CPU, memory, etc.) of the system.

Life cycle of an activity



Memory management

- When memory usage is high, the system may decide to kill a process to free up memory (Low Memory Killer).
- This decision is made based on several factors, including the importance(priority) of the process.
- When a process is terminated, all of its components are also terminated.
- When terminating a process, the framework sets the priority of each process according to its status, from processes with no components or with inactive components to the process running in the foreground (e.g., activity in execution).

Application security

- Android applications run in a sandbox, isolated from other applications and following the principle of least privilege:
  - Each application runs in a separate process.
  - Each application runs in an instance of the virtual machine.
  - The system assigns a user (UID) and group (GID) to each application (these credentials ensure that applications have access only to a limited set of operating system resources).
- Android applications are signed to ensure authenticity.

Permissions in applications

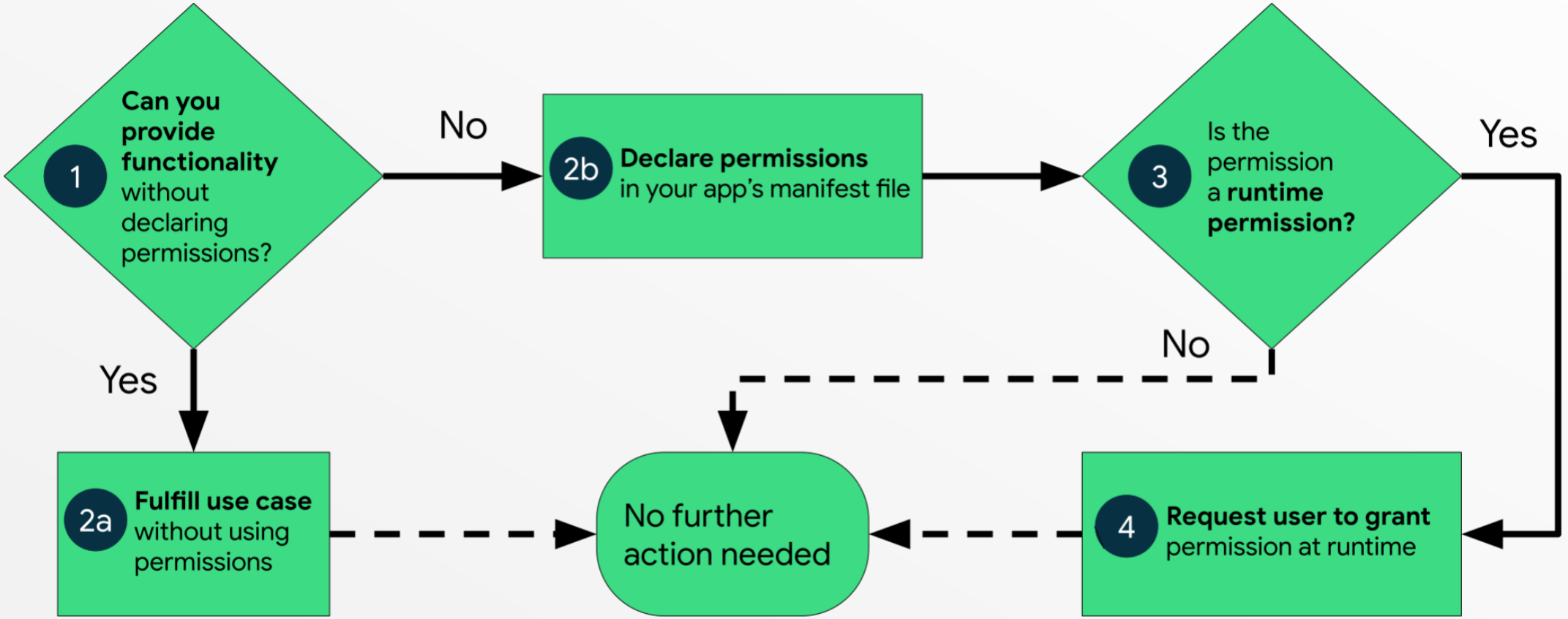
- In order for an application to access a resource that it does not have permission for, the developer must declare the necessary permissions in the application's manifest file.

```
<uses-permission android:name="android.permission.WRITE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

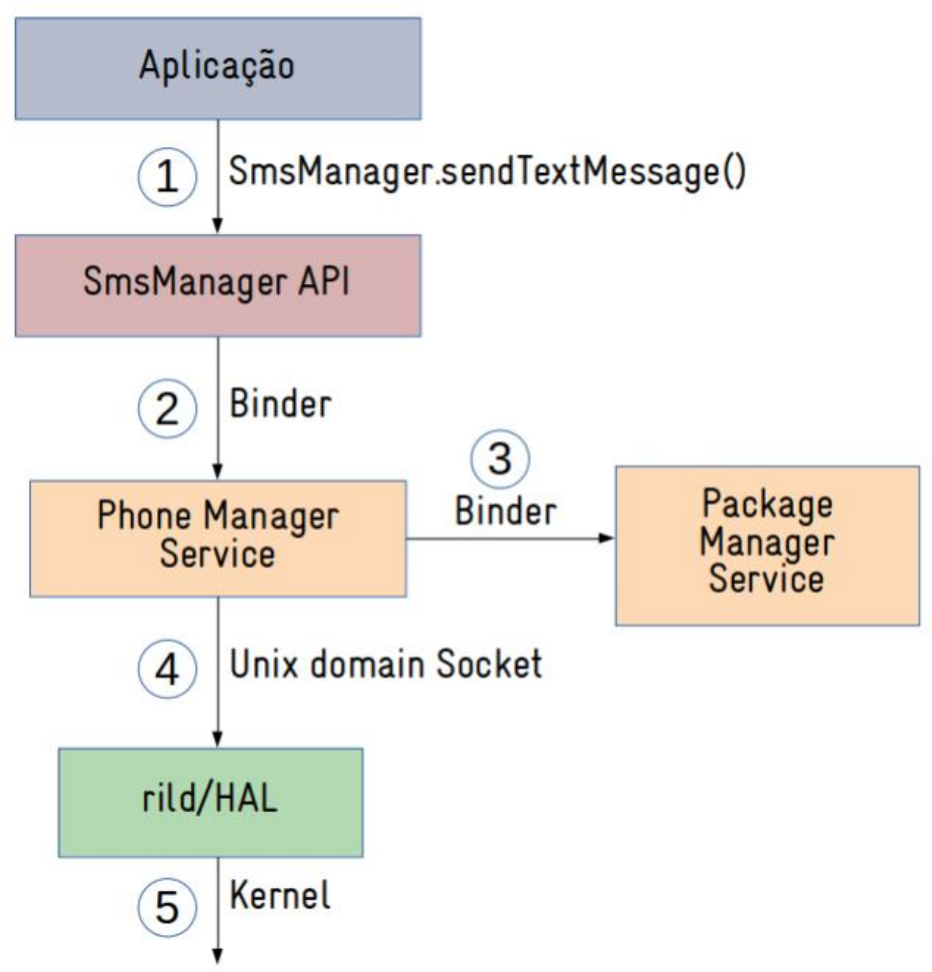
- During installation or execution of the application (depending on the type of permission), the user will need to approve access to the resource.
- A complete list of existing permissions is available on the project website.

  https://developer.android.com/reference/android/Manifest.permission

Workflow for working with permissions



Checking permissions



NDK

- The NDK (Native Development Kit) is a set of tools for developing Android applications in C/C++.
- Its use is common in some situations, including:
  - Performance (e.g.: mathematical calculations, games, etc.).
  - Code reuse (e.g.: Firefox, snes9x, etc.).
  - Use of specific kernel system calls (ioctl(), etc.).
- Allows direct access to the GPU via OpenGL/Vulkan!
- It can be combined with the SDK, allowing you to have an application that is partly in Java and partly in C/C++.

NDK (cont.)

- Applications developed with NDK are also packaged in an .apk.
- Although the code is not executed by the virtual machine, all security mechanisms are maintained.
- Some shortcomings:
  - Losses portability.
  - Access only to a part of the Android API.
  - Does not have the functionality of resources.

Distributing applications

- Android applications can be installed manually, but are usually distributed to users through an application repository.
- The default Android repository is Google Play.
- There are other repositories available, such as the Amazon AppStore.
- The F-Droid project makes it possible to create a custom application repository:

  https://www.f-droid.org/
  https://gitlab.com/fdroid

Dedicated applications on Android

- Developing a product with embedded Android means using the Android operating system to develop a device with a specific objective or purpose.
- In this case, we want to benefit from all the features of the operating system, but hide or disable all the elements of its graphical interface (status bars, navigation bars, notifications, navigation buttons, other applications, etc.).
- In Android, there are some features that we can use to run an application in a dedicated way (kiosk mode):

https://developer.android.com/work/dpc/dedicated-devices

Developing dedicated applications

- Step 1: develop a Launcher-type application.
- Step 2: run the application in a dedicated mode (immersive mode), with the graphical interface elements hidden.
- Step 3: Lock the application so that it is the only one running with the lock task mode feature.

Step 1: Launcher

- The first step to developing a dedicated application on Android is to transform it into a Launcher.
- The Launcher application is the application that runs automatically when Android boots.
- Turning the application into a Launcher is simple, just declare the following intents in the application's main Activity:

```
<category android:name="android.intent.category.HOME" />
<category android:name="android.intent.category.DEFAULT" />
```

Step 2: Immersive Mode

- The next step to run the application in a dedicated mode is to run it in immersive mode, hiding the Android graphical interface elements.
- To do this, we can use some Android APIs, including:
  - Disabling the title bar with the activity's *getSupportActionBar()* method.
  - Disabling graphical interface elements with the *WindowInsetsController* class.
  - Manipulating additional UI flags with the activity's *getWindow()* method.

Step 2: Immersive Mode (cont.)

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    new SerialPortRx().execute();

    // hide title bar
    getSupportActionBar().hide();

    // immersive mode
    getWindow().setDecorFitsSystemWindows(false);
    WindowInsetsController controller = getWindow().getInsetsController();
    if (controller != null) {
        controller.hide(WindowInsets.Type.statusBars() | WindowInsets.Type.navigationBars() | Win
        controller.setSystemBarsBehavior(WindowInsetsController.BEHAVIOR_SHOW_TRANSIENT_BARS_BY_S
    }

    // display always ON
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
}
```

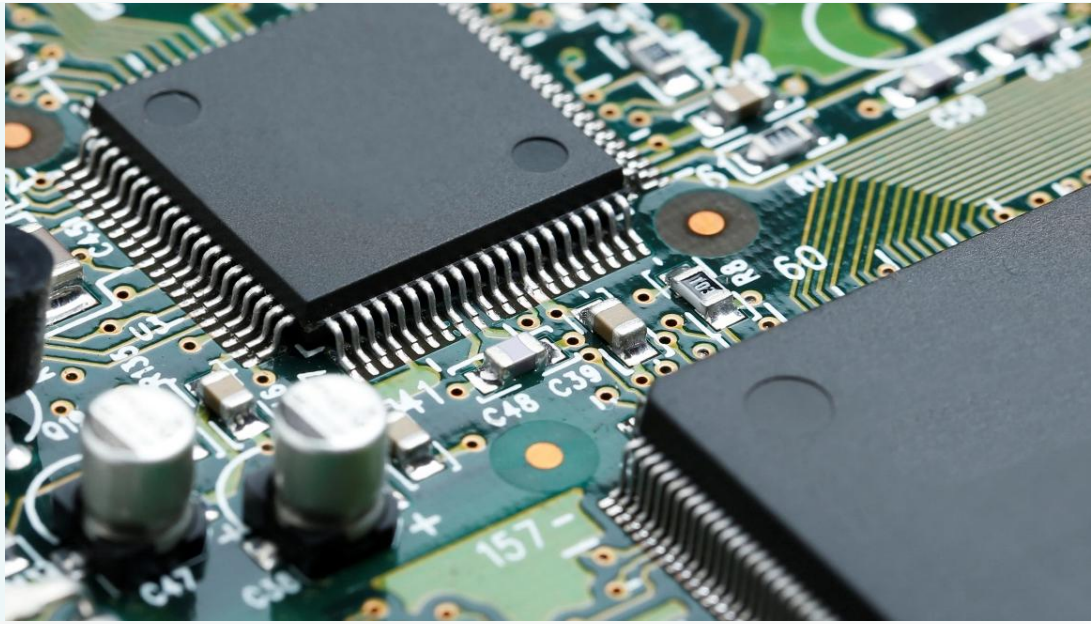
Step 3: Lock task mode

- The lock task mode feature allows you to lock Android to a specific application.

  https://developer.android.com/work/dpc/dedicated-devices/lock-task-mode

- In this mode, you can run the application in a fully dedicated and immersive way:
  - Only the selected application will be run.
  - Notifications will be disabled.
  - Buttons (volume, navigation, etc.) can be disabled.
- To use this mode, you need to make some changes to the application code, in addition to registering it as a device administrator using the dmp (Device Policy Control) command-line tool.

Additional resources

- http://www.elinux.org/Android_Portal
- https://emteria.com/blog
- Embedded-Android – Karim Yaghmour
- Android Internals (Volume 1 and Volume 2) - Jonathan Levin
- Android Security Internals - Nikolay Elenkov
- Learning Embedded Android N Programming - G. Blake Meike, Larry Schiefer, Lawrence Schiefer
- Androids: The Team That Built Android Operating System - Chet Haase

Q&A