Marakana Android Internals

Contents

Chapter 1

Android Stack

- Exploring the anatomy of Android. A walk through the Android stack, starting from the bottom and moving up.
  - Linux Kernel Layer
  - Native Layer
  - Application Framework Layer
  - Applications Layer

1.1 Android Linux Kernel Layer

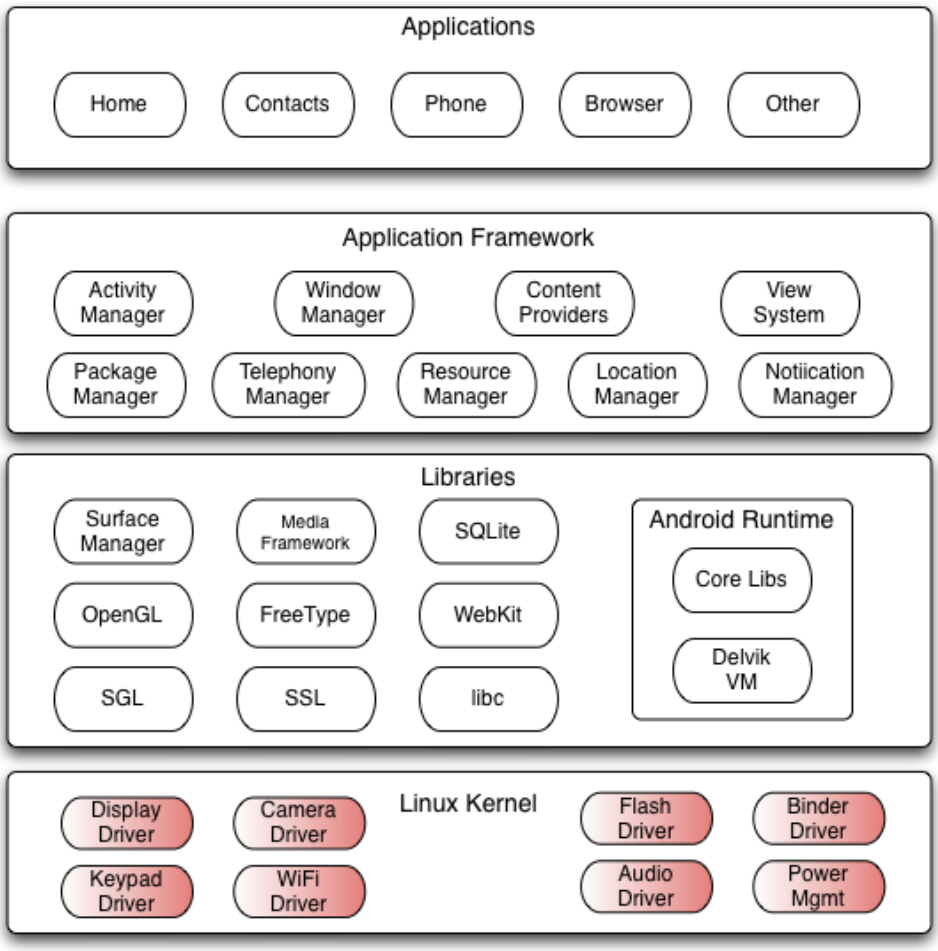


Figure 1.1: Android Kernel Layer

1.1.1 Overview

- Android runs on a modified Linux kernel

Table 1.1: Android Version - Linux Kernel Version

Android Version	**Kernel Version**
1.0	2.6.25
1.5	2.6.27
1.6	2.6.29
2.2	2.6.32
2.3	2.6.35
3.0	2.6.36
common (as of March 2011)	2.6.38

- Android is **not** "Linux"
  - no glibc
  - no X11
  - many standard configuration files are missing: no `/etc/passwd`, no `/etc/fstab`, etc.
  - many standard Linux utilities are missing: no `/bin/cp`, no `/bin/su`, etc.
- In the *Android Stack* Linux kernel provides
  - Hardware abstraction layer (low level)
    - Well-understood driver model
    - Many drivers for common devices
    - "Free" drivers for future devices
  - Process and memory management
  - Simple, but secure, per-process sandboxing (permissions-based security model)
  - Support for shared libraries
  - Network stack
- Application developers and users never "see" the Linux kernel
- The `adb shell` command opens Linux shell (remember, limited standard utils)
- Maintains a separate "forked" git tree (http://android.git.kernel.org/)

Table 1.2: Android Linux Kernel GIT Repositories

Android Version	**Kernel Version**
kernel/common.git	The "official" Android Kernel branch (used by the emulator)
kernel/experimental.git	The experimental extensions
kernel/linux-2.6.git	The mirror of Linus' kernel tree
kernel/lk.git	(L)ittle (K)ernel bootloader
kernel/msm.git	Kernel tree for MSM7XXX family and Android on MSM7XXX (Qualcomm)
kernel/omap.git	Kernel tree for TI's OMAP family SOCs on Android
kernel/samsung.git	Kernel tree for Samsung SOCs systems on Android
kernel/tegra.git	Kernel tree for NVIDIA Tegra family SOCs on Android

- Android introduces **many extensions**
  - Required configuration options at `http://goo.gl/HwEHC`

The following are some of the changes/additions Android makes to the Linux kernel.

1.1.2 Binder IPC

- OpenBinder-based IPC driver enables "object-oriented operating system environment"
  - Exposed via `/dev/binder`
  - Runtime info at `/proc/misc/binder`
  - Like CORBA, but much simpler
  - Initially developed for BeOS later used by Palm (which acquired BeOS)
- By default, apps and services (including system services) run in separate processes, but often need to share data
- Traditional IPC leads to security challenges and adds overhead, which is amplified on a mobile device
- Most of Android infrastructure (services) is supported by Binder
- Binder is lightweight and high-performance
  - Bound services are automatically reference-counted and "garbage collected" when no longer in use
  - Provides automatic per-process thread pooling for services (with remote clients)
  - Remote (service) method calls are synchronous (feels like just a function call, even though it's IPC)
    - Supports `oneway` (asynchronous) execution model as well
- Services defined/exposed via AIDL
- Implementation is at `drivers/misc/binder.c` with include at `include/linux/binder.h`

1.1.3 Ashmem (Anonymous SHared MEMory)

- A reference-counted, virtually mapped, named memory block that is shared between processes that the kernel is allowed to free
- Similar to POSIX SHM but with different behavior and a simpler file-based API (POSIX SHM does not allow the kernel to free shared memory)
- Programs open `/dev/ashmem`, use `mmap()` on it, and then share file handles via binder

```
int size = 4096;
int fd = ashmem_create_region("MySharedRegionName", size);
if (fd == 0) {
  data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if(data != MAP_FAILED) {
        /* for security reasons, no other process can ashmem_create_region() with the ↩
           same name */
        /* instead, to share this memory, we send fd via Binder IPC to another  ↩
           process */
        /* that process then mmap()'s it the same way in order to access the shared  ↩
           memory */
  }
}
```

  - When all processes `close(fd)` to the shared memory region, the kernel automatically reclaims that memory (because it is reference-counted)

- Supports a number of `ioctl()`-s: `ASHMEM_SET_NAME`, `ASHMEM_GET_NAME`, `ASHMEM_SET_SIZE`, `ASHMEM_GET_SIZE`, `ASHMEM_SET_PROT_MASK`, `ASHMEM_GET_PROT_MASK`, `ASHMEM_PIN`, `ASHMEM_UNPIN`, `ASHMEM_GET_PIN_STATUS`, `ASHMEM_PURGE_ALL_CACHES`
- Android uses ashmem to share resources to minimize redundancy across processes
- Kernel can discard unused shared blocks of memory when under pressure
- Implementation at `mm/ashmem.c` with include at `include/linux/ashmem.h`

1.1.4 Pmem

- Process memory allocator
- Used to manage large (1-16+ MB) physically contiguous regions of memory shared between userspace and kernel drivers (DSP, GPU, etc.)
  - Unlike ashmem, pmem is not reference-counted: the process that allocates a pmem heap is required to hold the file descriptor until all the other references are closed, so that it can free it explicitly
- Originally developed for MSM7201A
- Implementation at `drivers/misc/pmem.c` with include at `include/linux/android_pmem.h`
- Supports a number of `ioctl()`-s: `PMEM_GET_PHYS`, `PMEM_MAP`, `PMEM_GET_SIZE`, `PMEM_UNMAP`, `PMEM_ALLOCATE`, `PMEM_CONNECT`, `PMEM_GET_TOTAL_SIZE`, `PMEM_CACHE_FLUSH`

1.1.5 Wakelocks

- Extended power management
- More aggressive power-manager policy than standard Linux PM
  - "CPU shouldn't consume power if no applications or services require power" - i.e. the CPU shuts down eagerly
  - Applications and services that wish to continue running (after a short timeout following a user activity) are required to request "wake locks" via the app framework or native libs
- Wake locks are used by applications and services to request CPU resources
  - `WAKE_LOCK_SUSPEND`: prevents a full system suspend
  - `WAKE_LOCK_IDLE`: prevents entering low-power states from idle to avoid large interrupt latencies (disabled interrupts)
- Exposed via write calls to `/sys/power/wake_lock` and `/sys/power/wake_unlock`, which take `lockname` and an optional `timeout` (in nanoseconds)
- Support for different types of wake locks: `ACQUIRE_CAUSES_WAKEUP`, `FULL_WAKE_LOCK`, `ON_AFTER_RELEASE`, `PARTIAL_WAKE_LOCK`, `SCREEN_BRIGHT_WAKE_LOCK`, `SCREEN_DIM_WAKE_LOCK`
- Alls wake-lock management (above the kernel) should go through Java-based `PowerManager` service (even from user-space native libraries)
- Allows drivers to be notified of early suspend when the user-space write to `/sys/power/request_state`: `EARLY_SUSPEND_LEVE`
  `EARLY_SUSPEND_LEVEL_STOP_DRAWING`, `EARLY_SUSPEND_LEVEL_DISABLE_FB`, `EARLY_SUSPEND_LEVEL_STOP_INP`

- See `http://source.android.com/porting/power_management.html` for a detailed diagram and more info
- Baseband processor normally does not shut down, so network traffic still raises interrupts allowing CPU to wake up
- Stats exposed via `/proc/wakelocks`
- Implementation at `drivers/android/power.c` with include at `include/linux/wakelock.h`

1.1.6 Alarm

- Kernel support for Android's `AlarmManager`
- User-space tells kernel when it would like to wake up
- Kernel schedules a time-based call-back (while holding a WakeLock) regardless of the sleep-state of the CPU
  - Apps can then run (need to hold their own WakeLocks)
- Implementation at `drivers/rtc/alarm.c` with include at `include/linux/android_alarm.h`
- Exposed via `/dev/alarm`
- Supports a number of `ioctl()`-s: `ANDROID_ALARM_CLEAR`, `ANDROID_ALARM_WAIT`, `ANDROID_ALARM_SET`, `ANDROID_ALARM_SET_AND_WAIT`, `ANDROID_ALARM_GET_TIME`, `ANDROID_ALARM_GET_TIME`

1.1.7 Low Memory Killer

- Automatically kills eligible least-recently-used processes when running low on memory
- More aggressive than standard OOM handling
- Implementation at `drivers/misc/lowmemorykiller.c` and `security/lowmem.c`
- System sets up 6 priority slots via `init.rc` (writes to `/sys/module/lowmemorykiller/parameters/`) and user-space `ActivityManager` then sets each app's `/proc/<pid>/oom_adj` (from -16 to 15)

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel.  These are used in ActivityManagerService.
    setprop ro.FOREGROUND_APP_ADJ 0
    setprop ro.VISIBLE_APP_ADJ 1
    setprop ro.PERCEPTIBLE_APP_ADJ 2
    setprop ro.HEAVY_WEIGHT_APP_ADJ 3
    setprop ro.SECONDARY_SERVER_ADJ 4
    setprop ro.BACKUP_APP_ADJ 5
    setprop ro.HOME_APP_ADJ 6
    setprop ro.HIDDEN_APP_MIN_ADJ 7
    setprop ro.EMPTY_APP_ADJ 15

# Define the memory thresholds at which the above process classes will
# be killed.  These numbers are in pages (4k).
    setprop ro.FOREGROUND_APP_MEM 2048
    setprop ro.VISIBLE_APP_MEM 3072
    setprop ro.PERCEPTIBLE_APP_MEM 4096
    setprop ro.HEAVY_WEIGHT_APP_MEM 4096
    setprop ro.SECONDARY_SERVER_MEM 6144
```

```
    setprop ro.BACKUP_APP_MEM 6144
    setprop ro.HOME_APP_MEM 6144
    setprop ro.HIDDEN_APP_MEM 7168
    setprop ro.EMPTY_APP_MEM 8192

# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have combined some of
# the classes into the same memory level; the associated processes of higher
# classes will still be killed first.
    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,4,7,15

    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    write /sys/module/lowmemorykiller/parameters/minfree  ↩
        2048,3072,4096,6144,7168,8192

    # Set init its forked children's oom_adj.
    write /proc/1/oom_adj -16
```

- To application developers this means that low memory killer stacks processes based on the following order:
  1. Foreground processes - with an `Activity` that just ran `onResume()`, or a `Service` bound to it or started as foreground, or executing its callback methods, or a `BroadcastReceiver` executing `onReceive()`
  2. Visible processes - with an `Activity` that just ran `onPause()` but is still visible or a `Service` bound to a component from a visible process
  3. Service processes - with a `Service` that has been started with `Context.startService()`
  4. Background processes - with an `Activity` that just ran `onStop()`
  5. Empty processes - with no components (kept around just for caching purposes)
- See http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html
- Everything directly started from `init.rc` (including the `system_server`) has its `oom_adj` set to `-16`
  - If we get to killing those, the system is toast anyway
- Applications can request that they be kept persistent (in memory) by setting `<application android:persistent="true" ...>` in their `AndroidManifest.xml` file
  - `ActivityManager` then starts persistent apps and initializes their `oom_adj` to `-12`

1.1.8 Logger

- System-wide logging facility (from Kernel all the way to apps)
- This is what `logcat` command reads from
- Supports four auto-rotating log buffers managed by the kernel
  - `/dev/log/main` (64KB)
    - Destination for Android apps
    - Most logging happens via `android.util.Log`
  - `/dev/log/system` (64KB)

    - Destination for Android framework's system services and libraries
    - Most logging happens via hidden `android.util.Slog` or directly via `liblog` library
  - `/dev/log/events` (256KB)
    - Destination for Android system diagnostic events - e.g. garbage collections, activity manager state, system watchdogs, and other low level activity
    - Logging via `android.util.EventLog` or directly via `liblog` library
    - Binary-encoded - can be decoded via `/system/etc/event-log-tags`
  - `/dev/log/radio` (64KB)
    - Destination for radio and phone-related information
- Log reading and writing is done via normal Linux file I/O
  - Calls to `open()`, `write()`, and `close()` are extremely low-overhead on these devices
  - Each `read()` returns exactly one log entry (up to 4KB) and can be both blocking and non-blocking
- Implementation at `drivers/misc/logger.c` (with a `logger.h` in the same directory)
- Supports a number of `ioctl()`-s: `LOGGER_GET_LOG_BUF_SIZE`, `LOGGER_GET_LOG_LEN`, `LOGGER_GET_NEXT_ENTRY_LEN`, `LOGGER_FLUSH_LOG`

1.1.9 Paranoid Network Security

- Restricts access to some networking features depending on the group of the calling process
- Enabled via `ANDROID_PARANOID_NETWORK` kernel build option, which defines Kernel group IDs that have special network access

/include/linux/android_aids.h (Kernel source-tree)

```
/* AIDs that the kernel treats differently */
...
#define AID_NET_BT_ADMIN 3001
#define AID_NET_BT       3002
#define AID_INET         3003
...
```

- These are re-defined for Android user-space and assigned logical group-names

/system/core/include/private/android_filesystem_config.h (Android source-tree)

```
...
#define AID_NET_BT_ADMIN  3001  /* bluetooth: create any socket */
#define AID_NET_BT        3002  /* bluetooth: create sco, rfcomm or l2cap sockets */
#define AID_INET          3003  /* can create AF_INET and AF_INET6 sockets */
...
static const struct android_id_info android_ids[] = {
    ...
    { "net_bt_admin",     AID_NET_BT_ADMIN, },
    { "net_bt",           AID_NET_BT, },
    ...
    { "inet",             AID_INET, },
    ...
};
...
```

- Android app permissions are then mapped to group names

  /system/etc/permissions/platform.xml (system image)

```
<permissions>
  ...
  <permission name="android.permission.BLUETOOTH_ADMIN" >
    <group gid="net_bt_admin" />
  </permission>

  <permission name="android.permission.BLUETOOTH" >
    <group gid="net_bt" />
  </permission>

  <permission name="android.permission.INTERNET" >
    <group gid="inet" />
  </permission>
  ...
</permissions>
```

1.1.10 Other Kernel Changes

- Timed output / Timed GPIO
  - Generic GPIO allows user space to access and manipulate GPIO registers
  - Timed GPIO allows changing a GPIO pin and having it restored automatically after a specified timeout
  - Implementation at `drivers/android/timed_output.c` and `drivers/android/timed_gpio.c`
  - Used by the vibrator by default
- Linux Scheduler
  - Not a custom scheduler, just Android-specific configuration in `init.rc`

```
write /proc/sys/kernel/panic_on_oops 1
write /proc/sys/kernel/hung_task_timeout_secs 0
write /proc/cpu/alignment 4
write /proc/sys/kernel/sched_latency_ns 10000000
write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
write /proc/sys/kernel/sched_compat_yield 1
write /proc/sys/kernel/sched_child_runs_first 0
```

- Switch events - userspace support for monitoring GPIO used by `vold` to detect USB
- USB gadget driver for ADB (`drivers/usb/gadget/android.c`)
- yaffs2 flash filesystem, though this is switching to ext4
- RAM console
  - Kernel's `printk` goes to a RAM buffer
  - A kernel panic can be viewed in the next kernel invocation via `/proc/last_kmsg`
- Support in FAT filesystem for `FVAT_IOCTL_GET_VOLUME_ID`

1.2 Android User-Space Native Layer

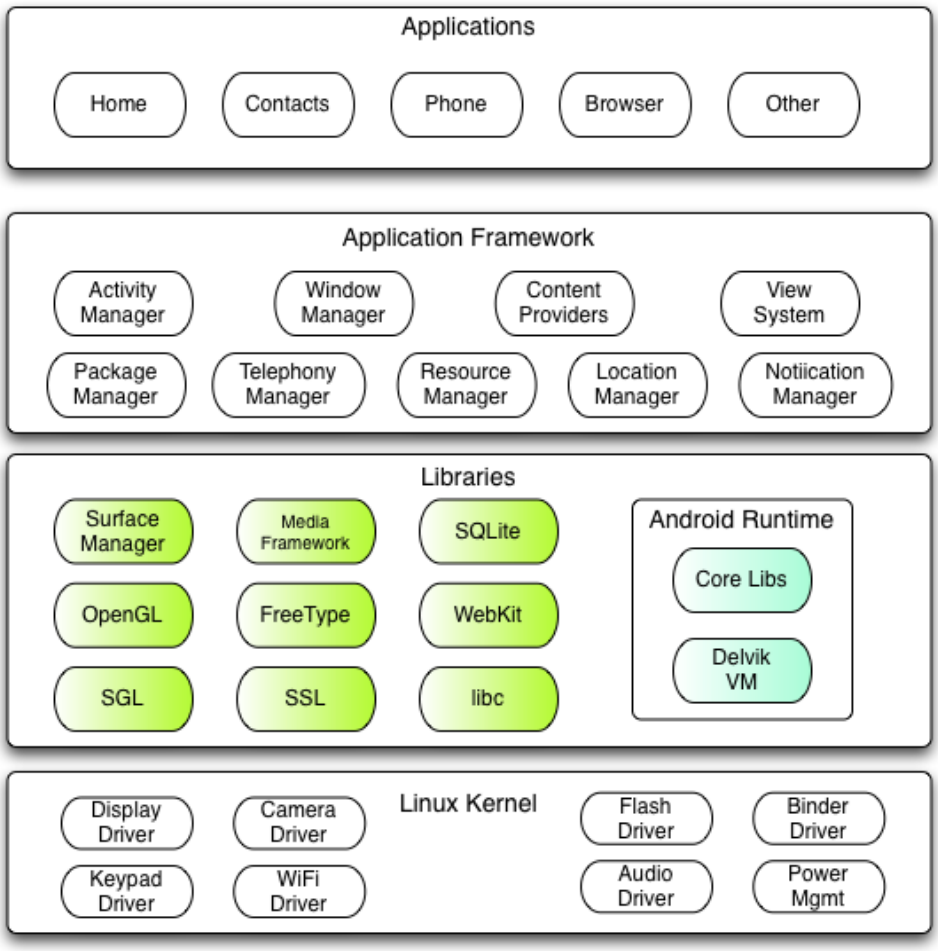


Figure 1.2: Android User-Space Native Layer

- Android user-space native layer is divided into multiple "logical" categories

1.2.1 Bionic Library

- Custom standard C library (libc) derived from BSD optimized for Android
- Why Bionic?
  - BSD licensed (business-friendly - i.e. keeps GPL out of user-space)
    - Proprietary code linked to bionic can remain proprietary
  - Lean (~200KB, or about galf the size of glibc)
    - It is loaded into every process, so it needs to be small
  - Fast (custom pthread impl) - perfect for embedded use

1.2.2 Changes From BSD libc

- Support for arbitrary Android system-properties via `<sys/system_properties.h>`
- Support for Android Kernel Logger Driver (via `liblog`)
- Support for Android-specific user/group management
  - Enabled via `getpwnam()`, `getgrouplist()`, and, `getgrgid()`, which are aware of generated UIDs of the applications (>10000) and their corresponding synthetic user/group-names (e.g. app_123)
  - Basic UID/GIDs defined in `/system/core/include/private/android_filesystem_config.h`
- Support for Android-specific `getservent()`, `getservbyport()`, and `getservbyname()` in place of `/etc/services`
- No support for `/etc/protocol`
- "Clean" kernel headers that allow user-space to use kernel-specific declarations (e.g. ioctl's, structure declarations, constants, etc.)
- Custom pthread implementation based on Linux futexes
  - Bundled-in (i.e. `-lpthread` not required)
  - Optimized for embedded use - strives to provide **very** short code paths for common operations
    - Normal, recursive and error-check mutexes are supported
    - No support for process-shared mutexes and condition variables (use Android's own Binder-IPC instead) as well as read/write locks, priority-ceiling in mutexes, `pthread_cancel()`, and other more advanced features (not a priority for embedded use)
    - Provides only 64 as opposed to 128 thread-specific storage slots required by POSIX
    - No support for read/write memory barriers (restricts SMP/multi-core on certain architectures)
  - Does not support all the relocations generated by other GCC ARM toolchains
- No support for System V IPCs - to avoid denial-of-service
  - SysV has no way to automatically release a semaphore allocated in the kernel when
    - a buggy or malicious process exits
    - a non-buggy and non-malicious process crashes or is explicitly killed (e.g. via low-memory-killer)
  - Again, we use Android's own Binder-IPC instead
- No support for locales (I18N done at the application/Dalvik layer via well-defined resource mechanism)
  - No support for wide chars (i.e. multi-byte characters)
- `time_t` is 32-bit on 32-bit hardware
- Timezones are defined via `TZ` env-vars or via `persist.sys.timezone` system property
- NetBSD-derived DNS resolver library
  - Reads from `/system/etc/resolv.conf`
  - Uses name servers defined in `net.dns1`, `net.dns2` system properties
    - Can be process specific: `net.dns1.<pid>`
- Built-in linker

  - Support pre-linked mapping files
- Support for x86, ARM and ARM thumb CPU instruction sets and kernel interfaces
- **Not binary compatible** with any other known Linux C library (glibc, ucLibc, etc.)
  - Not event fully POSIX-compliant
  - No support for C++ exceptions
  - **Requires recompile** of existing legacy code against bionic
- See `ndk/docs/system/libc/OVERVIEW.html` for more info (in the Android source tree)

1.2.3 User-space Hardware Abstraction Layer (HAL)

- User-space C/C++ hardware abstraction layer - as shared libraries
- Communicate with Linux drivers via `/dev/`, `/sys/`, or `/proc/`
- Why not just use Linux drivers directly?
  - Separates Android platform logic from specific hardware interfaces
  - Linux does not have common definitions of hardware that upper layers depend on
  - User-space HAL offer *standard* "driver" definitions for graphics, audio, camera, bluetooth, GPS, radio (RIL), WiFi, etc.
  - Makes porting easier
- OEMs implement "drivers" for specific hardware as shared libraries
  - `libhardware`
  - `libhardware_legacy`
  - The code can remain proprietary since the user-space drivers link against bionic, not the kernel (i.e. no GPL)
- Platform loads these HAL libs at runtime via pre-determined naming strategies
  - `libhardware` is a simple shared library that can load device-specific shared libraries via `hw_get_module(const char *id, const struct hw_module_t **module)`
    * First checks under `/vendor/lib/hw/` and then `/system/lib/hw/` as follows:
      1. `<*_HARDWARE_MODULE_ID>.<ro.product.board>.so`
      2. `<*_HARDWARE_MODULE_ID>.<ro.board.platform>.so`
      3. `<*_HARDWARE_MODULE_ID>.<ro.arch>.so`
      4. `<*_HARDWARE_MODULE_ID>.default.so`
    * For example, on a Nexus S (where `TARGET_BOARD_PLATFORM=s5pc110`, `board=herring`, and `/vendor/` → `/system/vendor`)
      · The GPS "driver" is loaded from `/system/vendor/lib/hw/gps.s5pc110.so` (where `GPS_HARDWARE_MODULE_ID="`
      · The Sensors "driver" is loaded from `/system/lib/hw/sensors.herring.so` (where `SENSORS_HARDWARE_MODULE_I`
  - `libhardware_legacy` is a shared library for vibrator, wifi-module-loader, power, uevent, audio, camera, etc.
    * Some board-independent hardware (like vibrator, power, wifi, etc.) is directly supported by `/system/lib/libhardware_lega` via well-defined paths on the file system (mostly via `/proc` or `/sys`) or well-defined system properties
    * Board-specific devices (like audio, camera, etc.) are supported by separate shared libraries loaded by well-defined names (e.g. `/system/lib/libaudio.so`, `/system/lib/libcamera.so`, etc.)
  - Radio is a bit special, as its `rild` (Radio Interface Link Daemon) loads "libril" as defined by `rild.libpath` system property
    * On Nexus S, this is `/vendor/lib/libsec-ril.so`

1.2.4 Native Daemons

- `/system/bin/servicemanager`
  - *The naming service* for all other `systemserver`'s (i.e. framework) services
  - Registers as `BINDER_SET_CONTEXT_MGR` on `/dev/binder` and starts reading from it (in a loop)
- `/system/bin/vold`
  - Volume Daemon used for mounting/unmounting removable media (like `/mnt/sdcard`) on demand
  - Configured via `/system/etc/vold.fstab`
  - For example, Nexus One's SD Card is automatically re/mounted:

```
dev_mount sdcard /mnt/sdcard auto /devices/platform/goldfish_mmc.0 /devices/ ↩
    platform/msm_sdcc.2/mmc_host/mmc1
```

  - While Nexus S, has a "virtual" but fixed SDCard:

```
dev_mount sdcard /mnt/sdcard 3 /devices/platform/s3c-sdhci.0/mmc_host/mmc0/mmc0 ↩
    :0001/block/mmcblk0
```

- `/system/bin/rild`
  - Acts as a bridge between platform framework services (`TelephonyManager`) and `libril`, which is OEM-specific interface to the baseband modem
  - Stateful - helps handle incoming calls/messages (unsolicited requests)
  - Interfaces with upper layers via a Unix socket connection
- `/system/bin/netd`
  - Manages network connections, routing, PPP, etc.
  - Enables tethering, connections over USB/Bluetooth, etc.
- `/system/bin/mediaserver`
  - Home of `AudioFlinger`, `MediaPlayerService`, `CameraService`, `AudioPolicyService`
- `/system/bin/installd`
  - Listens on a unix socket and performs installation/uninstallation of packages (i.e. apps)
  - Used by the `PackageManager`
- `/system/bin/keystore`
  - Listens on a unix socket and provides secured storage for key-value pairs
  - Keys are encoded in file names, and values are encrypted with checksums
  - The encryption key is protected by a user-defined password
- `/system/bin/debuggerd`
  - Catches crashes of native processes and dumps their stack trace to `/data/tombtones/`
  - When native processes initialize, they implicitly connect (through a unix socket) to `debuggerd` through a separate thread spawned by bionic
- `/system/bin/wpa_supplicant`

  - Handles WPA authentication for WiFi networks
- `/system/bin/dhcpd`
  - Requests (leases) IPs from DHCP servers
  - Handles network changes (e.g. 3G to Wifi)
- BlueZ (Bluetooth support daemons)
  - `/system/bin/dbus-daemon`
    * A simple IPC (bus) framework
    * Provides `systemserver` with a way to access `hcid` (Bluetooth Host Controller Interface Daemon)
  - `/system/bin/bluetoothd`
    * Manages device pairings and the rest of the stack
  - `/system/bin/sdptool`
    * Used to manage individual Bluetooth profile as services
      · `hfag` - Hands-Free Profile
      · `hsag` - Headset Profile
      · `opush` - Object Push Profile
      · `pbap` - Phonebook-Access Profile
      · etc.
- `/system/bin/racoon`
  - Assists with ipsec key negotiations (IKE)
  - Used for VPN connections
- `/system/ueventd`
  - Handles uevents from the Kernel and sets up correct ownership/permissions on the device file descriptors
  - Applies configuration from `/ueventd.rc`

```
...
/dev/urandom              0666   root       root
/dev/ashmem               0666   root       root
/dev/binder               0666   root       root
...
/dev/log/*                0662   root       log
...
/dev/ttyMSM0              0600   bluetooth  bluetooth
...
/dev/alarm                0664   system     radio
...
/dev/cam                  0660   root       camera
...
/dev/akm8976_pffd         0640   compass    system
/dev/lightsensor          0640   system     system
...
/dev/bus/usb/*            0660   root       usb
/sys/devices/virtual/input/input*   enable      0660  root   input
...
```

- `/sbin/adbd`

  - End-point of Android Debug Bridge
  - Accepts ADB connections over USB
  - Possible to accept network connections as well
- Vendor-specific daemons which fasciliate interaction with the hardware
  - For example `/system/vendor/bin/gpsd` on Nexus S
- Other daemons, like `zygote` and `systemserver` will be discussed separately

1.2.5 Flingers

1.2.5.1 Surface Flinger

- `SurfaceFlinger` is Android's system-wide screen composer that draws into standard Linux frame-buffer (`/dev/fb0`)
- Apps draw (in 2D or 3D) into "windows", which are implemented as double-buffered `Surface` objects backed by the surface flinger
  - Front-buffer used for composition, back-buffer for drawing
  - Buffers are flipped after drawing
    * Minimal buffer copying
    * Avoids flickers and artifacts as the front-buffer is always available for composition
- Surface flinger expects the video driver to offer:
  - A linear address space of mappable memory
    * Video memory is `mmap()`'ed to process address-space for direct writing
    * Enough video memory for twice the physical screen area
      · Otherwise, regular system memory has to be used for buffering, and is copied on flips (slow!)
  - Support for RGB 565 pixel format

1.2.5.2 Audio Flinger

- `AudioFlinger` is Android's system-wide audio stream routing engine/mixer and audio input capture facility
  - Sits on top of device-specific `libaudio.so` implementation, which usually simply bridges to ALSA
- To play audio, apps send uncompressed mono/stereo PCM streams to audio flinger (usually via `MediaPlayer`)
  - Streams include ringtones, notifications, voice calls, touch tones, key tones, music
  - Audio flinger routes these streams to various outputs (earpiece, speakers, Bluetooth)
- To capture audio, apps request access to uncompressed input path managed by the audio flinger (usually via `MediaRecorder`)

1.2.6 Function Libraries

- Provide computation-intensive services to the rest of the platform
  - This is in addition to bionic
- Many pieces borrowed from other open source projects
  - LibWebCore/WebKit, V8, SQLite, OpenSSL, FreeType, etc.
  - Usually abstracted by Java counterparts
- Media Framework Libraries
  - Originally based on PacketVideo's OpenCORE platform
  - Switched to Stagefright with Gingerbread
  - Support for playback and recording of many popular audio and video formats, as well as static image files
    * MPEG4, H.264, VP8/WebM, MP3, AAC, AMR, JPG, and PNG
  - Pluggable via Khronos' OpenMAX IL (supports for codecs in both software and hardware)
- 3D libraries
  - Support for OpenGL ES 1.0 and 2.0 APIs
  - Comes with highly optimized 3D software rasterizer - when hardware does not offer native OpenGL support
- 2D libraries
  - SGL (Skia) - the underlying 2D graphics engine

1.2.7 Dalvik

- Dalvik is a custom clean-room implementation of a virtual machine, semantically similar to a JVM but **not** a JVM

  - Licensed under Apache 2.0 open-source license
- Provides Android app portability and consistency across various hardware (like a JVM)
- Runs Dalvik byte-code, stored in `.dex` files (**not** Java byte code)
  - Developers program in the Java language (i.e. `.java` files), which get compiled into Java byte-code (i.e. `.class` files)
  - Build-tools compile Java's `.class` files into a `.dex` file before packaging (into `.apk` files)
    * 3rd party libraries are also re-compiled into dex code
  - Dalvik never sees any Java byte-code
- Why not Java SE?
  - Java SE is too bloated for mobile environment
  - Would require too much redundancy (at the library level)
  - Not well-optimized for mobile (at the bytecode and interpreter level)
- Why not Java ME?
  - Costs $$$ - hinders adoption
  - Designed by a committee - hard to imagine iOS-like developer appeal
  - Apps share a single VM - not great for security sandboxing
  - Apps are second-rate citizens - don't get access to all the hardware
- Dalvik is optimized for embedded environment:
  - Minimal-memory footprint while providing a secure sandboxing model
    * Uncompressed `.dex` files are smaller than compressed `.jar` files due to more efficient bytecode
      · On average Dalvik byte code is 30% smaller than JVM byte code
      · Multiple classes in one `.dex` file
      · Shared constant pool (assumes 32-bit indexes)
      · Simpler class-loading
      · Because it is uncompressed, dex code can be memory-mapped and shared (i.e. `mmap()`-ed)
    * Each app runs in a separate instance of Dalvik
      · At startup, system launches `zygote`, a half-baked Dalvik process, which is forked any time a new VM is needed
      · Due to copy-on-write support, large sections of the heap are shared (including 1800+ preloaded classes)
      · Since each VM runs in a separate process, we get great security isolation
  - Register-based fixed-width CPU-optimized byte-code interpreter
    * Standard JVM bytecode executes 8-bit stack instructions - local variables must be copied to or from the operand stack by separate instructions
      · Memory speed to CPU speed is amplified on mobile CPUs - we want to minimize access to the main memory
    * Dalvik uses 16-bit instruction set that works directly on local variables (managed via a 4-bit virtual register field)
  - With JIT support, as of Froyo (2-5x performance improvement in CPU-bound code)
    * Trace-level granularity (more optimal than whole method-level compilations)
    * Fast context-switching (interpreted mode to native mode and back)
    * Well-balanced from performance vs. memory overhead perspective (~ 100-200KB overhead per app)

    * ~ 1:8 ratio of Dalvik to native code (mostly due to optimizations, like inlining)

- Includes support for instrumentation to allow tracing and profiling of running code
- Core libraries based on Java SE 5 (mostly from Apache Harmony), with many differences
  - No support for `java.applet`, `java.awt`, `java.lang.management` and `javax.management` (JMX), `java.rmi` and `javax.rmi`, `javax.accessibiliy`, `javax.activity`, `javax.imageio`, `javax.naming` (JNDI), `javax.print`, `javax.security.auth.kerberos`, `javax.security.auth.spi`, `javax.security.spi`, `javax.security.sasl`, `javax.sound`, `javax.swing`, `javax.transaction`, `javax.xml` (except for javax.xml.parsers), `org.ietf`, `org.omg`, `org.w3c.dom.*` (subpackages)
  - But support for Android APIs (including wrappers for OpenGL, SQLite, etc.), Apache HTTP Client (`org.apache.http`), JSON parser (`org.json`), XML SAX parser (`org.xml.sax`), XML Pull Parser (`org.xmlpull`), etc.

1.3 Android Application Framework Layer

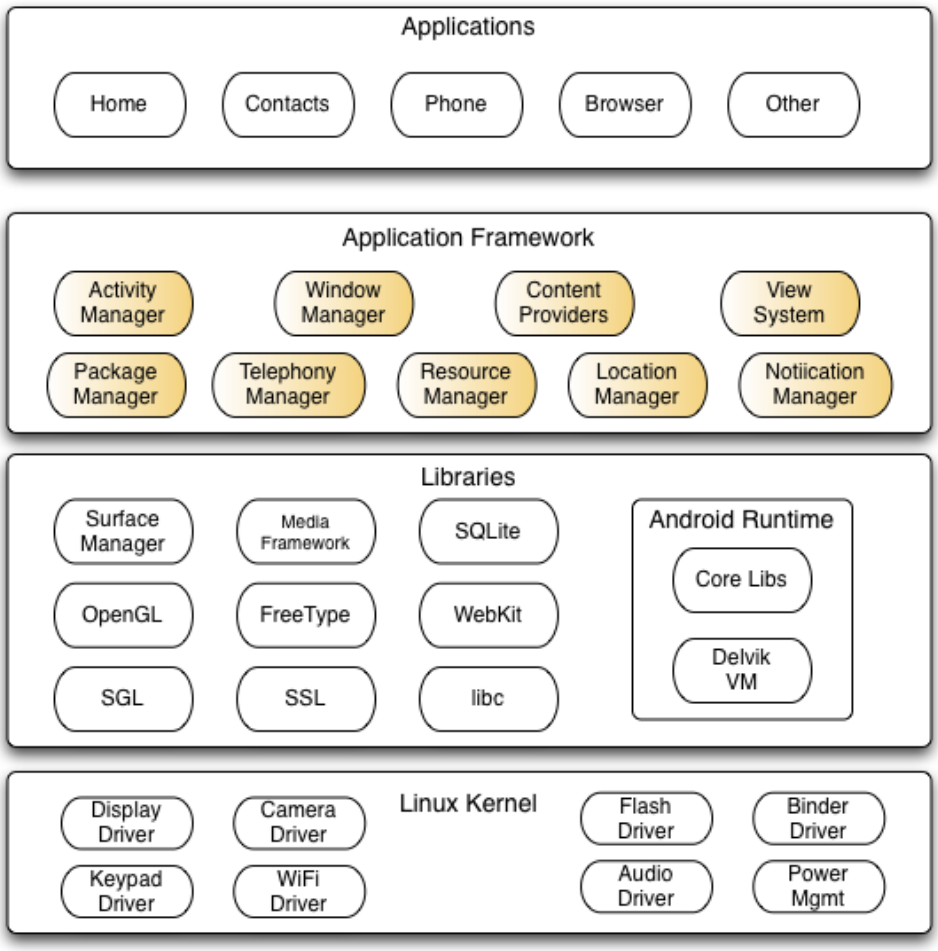


Figure 1.3: Android Application Framework Layer

1.3.1 Overview

- The rich set of system services wrapped in intuitive Java APIs
  - Most managed by the `systemserver` process and accessible via Binder/AIDL
- Abstraction of hardware services
  - Location, telephony, WiFi, Bluetooth, sensors, camera, etc.
- Java-language bindings for the native libraries (e.g. OpenGL, SQLite)
- Core platform services (like life-cycle management)
  - Essential to the apps, even if most are not used directly

1.3.2 Activity Manager Service

- Manages lifecycle of applications and their components
  - Sets up `oom_ajd` setting read by Low Memory Killer (Section 1.1.7) Android extension to the Linux kernel
- Handles application requests to `startActivity()`, `sendBroadcast()`, `startService()`, `bindService()`, etc.
  - Enforces security permissions on those requests
- Maintains user task state - i.e. the back-stack

1.3.3 Package Manager Service

- Along with `installd` responsible for installation of `.apk`-s on the Android system
- Maintains internal data structures representing installed packages as well as their individual components
  - Used by Activity Manager when handling intents (i.e. intent resolution is handled here)
  - Provides this info on demand to other services and apps
- Very central to the platform's security

1.3.4 Power Manager Service

- Controls power management
- Provides access to wake locks

1.3.5 Alarm Manager Service

- Manages wake-up alarms for applications
- Supports inexact wakeup frequencies - helps consolidate wake-ups into fewer slots
- Uses power manager for wake locks

1.3.6 Notification Manager Service

- Used by apps and other services to notify the user of events that may be of interest
  - This is how background events "bubble up" as notifications
- Supports persistent notification, as well as notifications that use LEDs, screen backlight, sound, and/or vibration to notify the user

1.3.7 Keyguard Manager Service

- Manages locking/unlocking of the keyguard

1.3.8 Location Manager Service

- Handles geographic location updates (e.g. GPS) and distributes them to the listening applications
- Support proximity alerts (via Intents)
- Supports providers of different granularity (GPS, Network, WiFi)

1.3.9 Sensor Manager Service

- Provides a uniform access to the device's sensors
- Apps request sensor notifications via this manager
  - Sensor manager delivers sensor updates via a generic (timestamped) array of values (which are sensor-dependent)
- Supported sensor types: accelerometer, linear acceleration, gravity, gyroscope, light, magnetic field, orientation, pressure, proximity, rotation vector, temperature
  - Actual sensor support is (obviously) hardware-dependent

1.3.10 Search Manager Service

- Provides a framework for device-wide (global) or app-specific search

1.3.11 Vibrator Manager Service

- Provides simplistic access to the vibrator hardware
- Can be used for simple haptic feedback (using patterns)

1.3.12 Connectivity Manager Service

- Monitors network connections (Wi-Fi, GPRS, UMTS, etc.) and
  - Send broadcast intents when network connectivity changes
  - Attempts to "fail over" to another network when connectivity to a network is lost
- Provides an API that allows applications to query the coarse-grained or fine-grained state of the available networks

1.3.13 Wifi Manager Service

- Unlike the connectivity manager, the Wifi Manager supports Wifi-specific operations
- Provides a list and allows management of configured networks
- Provides access to and management of the state of the currently active Wi-Fi network connection, if any
- Enables access point scans
- Broadcasts Intents on Wifi-connectivity state change events

1.3.14 Telephony Manager Service

- Provides access to information about the telephony services on the device
- Apps query Telephony Manager to determine telephony services and states, as well as to access some types of subscriber information (e.g. device id)
- Apps can also register a listener to receive notification of telephony state changes
- Handles tethering requests

1.3.15 Input Method Manager Service

- Central system to the overall input method framework (IMF) architecture
  - Arbitrates interaction between applications (each has a separate client) and the current input method
- Responsible for creating and running an input method (IME) to capture the actual input and translate it into text
  - Allows multiple apps to requests input focus and control over the state of IME

1.3.16 UI Mode Manager Service

- Provides access to the system UI mode
  - Enable/disable car-mode
  - Enable/disable night-mode
- System uses it to implement automatic UI mode changes
- Apps use it to manually control UI modes of the device

1.3.17 Download Manager Service

- Handles long-running HTTP downloads
- Clients request that a URI be downloaded to a particular destination file
- The download manager handles the download in the background, taking care of HTTP interactions and retrying downloads after failures or across connectivity changes and system reboots
- New as of Gingerbread (2.3, API 9)

1.3.18 Storage Manager Service

- Handles storage-related items such as Opaque Binary Blobs (OBBs)
- "OBBs contain a filesystem that maybe be encrypted on disk and mounted on-demand from an application. OBBs are a good way of providing large amounts of binary assets without packaging them into APKs as they may be multiple gigabytes in size. However, due to their size, they're most likely stored in a shared storage pool accessible from all programs. The system does not guarantee the security of the OBB file itself..."
  - E.g. great for GPS/Mapping applications that need support for off-line maps
- New as of Gingerbread (2.3, API 9)

1.3.19 Audio Manager Service

- Provides access to volume and ringer mode control
- Allows management/querying for the state of the audio system
  - Set/Get the current mode: normal, ringtone, in-call, in-communications
  - Set/Get the current ringer mode: normal, silent, vibrate
  - Set/Get the state of audio channels: speaker, bluetooth headset, wired headset, speakerphone
  - Set/Get the volume
  - Get audio focus requests
- Allows playing of sound effects: clicks, key-presses, navigation, etc.

1.3.20 Window Manager Service

- Manages windows (z-order) composed in a surface
- Allows us to place custom views (windows)

1.3.21 Layout Inflater Manager Service

- Used to instantiate layout XML files into its corresponding `View` object trees

1.3.22 Resource Manager Service

- Provides access to non-code resources such as localized strings, graphics, and layout files

1.3.23 Additional Manager Services

- Lights Service: handles status lights on the device
- Throttle Service: answers queries about data transfer amounts and throttling
- Mount Service: mount/unmount removable storage
- Battery Service: reports on battery health/status
- Wallpaper Service: manages changes to the wallpaper
- Backup Agent: provides remote backup/restore capabilities
- Bluetooth Service: manages bluetooth pairings
- Headset, Dock, USB Observers: observe specific device connections

1.4 Android Applications Layer

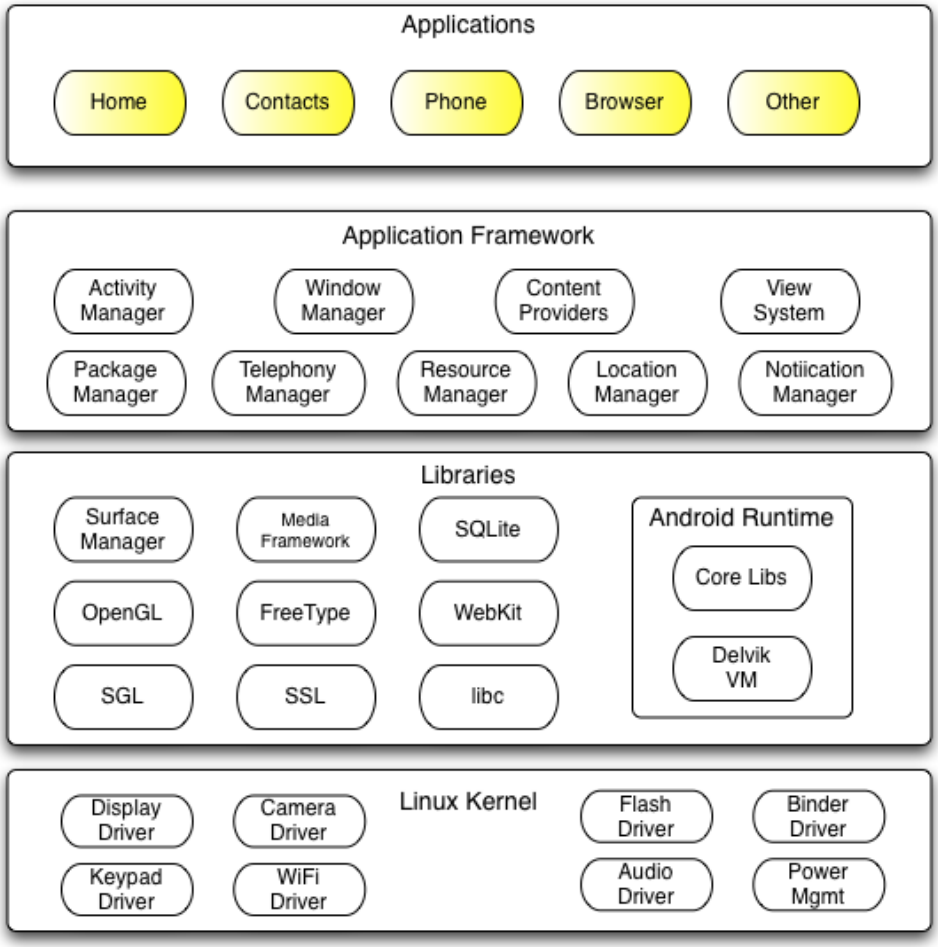


Figure 1.4: Android Application Layer

- Android ships with a number of built-in "applications"
  - These are stored on the read-only `/system/` partition under `/system/app`
  - Cannot be uninstalled without re-flashing the ROM
- An Android application is a loose set of components, which may be used as
  - A single independent cohesive unit (a "traditional" application)
  - A set to re-usable modules used by other applications via binder and/or Intents (in form of an API)
  - A combination of the two
- Applications are distributed as APKs (ZIP-compressed `.apk` files) consisting of
  - `AndroidManifest.xml` configuration file

- Dalvik byte-code (`classes.dex`)
- Optional native code as shared libraries (typically compiled for ARM)
- Optional resources including layout/menu/preference definitions, drawables, text, audio, styles, etc.
- Optional assets
- Signature/public key (used for signing)

1.4.1 Android Built-in Applications

- AccountsAndSyncSettings
- Bluetooth
- Browser
- Calculator
- Calendar
- Camera
- CertInstaller
- Contacts
- DeskClock
- Email
- Gallery
- Gallery3D
- HTMLViewer
- Launcher2
- Mms
- Music
- Nfc
- PackageInstaller
- Phone
- Protips
- Provision
- QuickSearchBox
- Settings
- SoundRecorder
- SpeechRecorder
- Stk
- Tag
- VoiceDialer
- *OEM-specific application packages*

1.4.2 Android Built-in Content Providers

- ApplicationsProvider
- CalendarProvider
- ContactsProvider
- DownloadProvider
- DrmProvider
- MediaProvider
- TelephonyProvider
- UserDictionaryProvider
- *OEM-specific content providers*

1.4.3 Android Built-in Input Methods

- LatinIME
- OpenWnn
- PinyinIME
- *OEM-specific input methods (like Swype)*

1.4.4 Android Built-in Wallpapers

- Basic
- LivePicker
- MagicSmoke
- MusicVisualization
- *OEM-specific wallpapers*

Chapter 2

Android Native Development Kit (NDK)

2.1 What is in NDK?

- Android's Dalvik VM allows our applications written in Java to call methods implemented in native code through the Java Native Interface (JNI)
  - For example:

```
package com.marakana.android.fetchurl;
public class FetchUrl {
    public static native byte[] fetch(String url);
    static {
        System.loadLibrary("fetchurl");
    }
}
```

- NDK is a tool-chain to build and cross-compile our native code for the device-specific architecture
  - At the moment, NDK supports ARMv5TE, ARMv7-A, and as of NDK r6 (July 2011) x86 ABIs
  - For example, we would implement native `fetch` method in C and compile it into a library `libfetchurl.so`
- NDK offers a way to package our native code library (as `lib<something>.so`) into the APK file so we can distribute it with our application easily
  - For example, our library would be packaged as `libs/armeabi/libfetchurl.so` in the APK
- NDK provides a set of native system headers that will be supported for the future releases of Android platform (`libc`, `libm`, `libz`, `liblog`, `libjnigrahics`, OpenGL/OpenSL ES, JNI headers, minimal C++ support headers, and Android native app APIs)
- Finally, NDK comes with extensive documentation, sample code and examples

2.2 Why NDK?

- NDK allows us to develop parts of our Android application in C/C++

- Generally, we do not develop native-only apps via NDK
  - Android's Gingerbread (2.3) release supports `NativeActivity`, which allows handling lifecycle callbacks in native code, so we can now develop native-only apps,
    - No support for native services, broadcast receivers, content providers, etc.
- NDK code still subject to security sandboxing - we don't get extra permissions for running natively
- Main motivation for native code is performance (CPU-intensive, self-contained, low-memory footprint code) and the re-use of legacy code
  - For system integrators, NDK offers access to low-level libraries (e.g. access to user-space HAL)
  - **Using NDK always increases complexity** of applications, so it should only be used when it's essential to the application
  - Programming in Java offers richer APIs, memory protection and management, higher-level language constructs (OOP), all of which generally results in higher productivity

2.3 Java Native Interface (JNI)

2.3.1 JNI Overview

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
  - Code reusability
    - Reuse existing/legacy code with Java (mostly C/C++)
  - Performance
    - Native code used to be up to 20 times faster than Java, when running in interpreted mode
    - Modern JIT compilers (HotSpot) make this a moot point
  - Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

Note
JNI can also be used to invoke Java code from within natively-written applications - such as those written in C/C++. In fact, the java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

2.3.2 JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - `javah` automatically includes this file in the application header files

2.3.3 JNI Development (Java)

- Create a Java class with native method(s): `public native void sayHi(String who, int times);`
- Load the library which implements the method: `System.loadLibrary("HelloImpl");`
- Invoke the native method from Java

For example, our Java code could look like this:

```
package com.marakana.jniexamples;

public class Hello {
  public native void sayHi(String who, int times); //❶

  static { System.loadLibrary("HelloImpl"); } //❷

  public static void main (String[] args) {
    Hello hello = new Hello();
    hello.sayHi(args[0], Integer.parseInt(args[1])); //❸
  }
}
```

❶, ❸ The method `sayHi` will be implemented in C/C++ in separate file(s), which will be compiled into a library.

❷ The library filename will be called `libHelloImpl.so` (on Unix), `HelloImpl.dll` (on Windows) and `libHelloImpl.jnil` (Mac OSX), but when loaded in Java, the library has to be loaded as `HelloImpl`.

2.3.4 JNI Development (C)

- We use the JDK `javah` utility to generate the header file `package_name_classname.h` with a function prototype for the `sayHi` method:
  `javac -d ./classes/ ./src/com/marakana/jniexamples/Hello.java`
  Then in the classes directory run: `javah -jni com.marakana.jniexamples.Hello` to generate the header file `com_marakana_jniexamples_Hello.h`
- We then create `com_marakana_jniexamples_Hello.c` to implement the `Java_com_marakana_jniexamples_Hello_sa` function

The file `com_marakana_jniexamples_Hello.h` looks like:

```
...
#include <jni.h>
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi
  (JNIEnv *, jobject, jstring, jint);
...
```

The file Hello.c looks like:

```
#include <stdio.h>
#include "com_marakana_jniexamples_Hello.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi(JNIEnv *env, jobject ↩
    obj, jstring who, jint times) {
  jint i;
  jboolean iscopy;
  const char *name;
  name = (*env)->GetStringUTFChars(env, who, &iscopy);
  for (i = 0; i < times; i++) {
    printf("Hello %s\n", name);
  }
}
```

2.3.5 JNI Development (Compile)

- We are now ready to compile our program and run it
  - The compilation is system-dependent
- This will create `libHelloImpl.so`, `HelloImpl.dll`, `libHelloImpl.jnilib` (depending on the O/S)
- Set `LD_LIBRARY_PATH` to point to the directory where the compiled library is stored
- Run your Java application

For example, to compile `com_marakana_jniexamples_Hello.c` in the "classes" directory (if your `.h` file and `.c` file are there) on Linux do:

```
gcc -o libHelloImpl.so -lc -shared \
    -I/usr/local/jdk1.6.0_03/include \
    -I/usr/local/jdk1.6.0_03/include/linux com_marakana_jniexamples_Hello.c
```

On Mac OSX :

```
gcc -o libHelloImpl.jnilib -lc -shared \
    -I/System/Library/Frameworks/JavaVM.framework/Headers ↩
        com_marakana_jniexamples_Hello.c
```

Then set the `LD_LIBRARY_PATH` to the current working directory:

```
export LD_LIBRARY_PATH=.
```

Finally, run your application in the directory where your compiled classes are stored ("classes" for example):

```
java com.marakana.jniexamples.Hello Student 5
Hello Student
Hello Student
Hello Student
Hello Student
Hello Student
```

Note
Common mistakes resulting in `java.lang.UnsatisfiedLinkError` usually come from incorrect naming of the shared library (O/S-dependent), the library not being in the search path, or wrong library being loaded by Java code.

2.3.6 Type Conversion

- In many cases, programmers need to pass arguments to native methods and they do also want to receive results from native method calls
- Two kind of types in Java:
  - Primitive types such as `int`, `float`, `char`, etc
  - Reference types such as classes, instances, arrays and strings (instances of `java.lang.String` class)
- However, primitive and reference types are treated differently in JNI
  - Mapping for primitive types in JNI is simple

Table 2.1: JNI data type mapping in variables:

Java Language Type	Native Type	Description
boolean	jboolean	8 bits, unsigned
byte	jbyte	8 bits, signed
char	jchar	16 bits, unsigned
double	jdouble	64 bits
float	jfloat	32 bits
int	jint	32 bits, signed
long	jlong	64 bits, signed
short	jshort	16 bits, signed
void	void	N/A

  - Mapping for objects is more complex. Here we will focus only on strings and arrays but before we dig into that let us talk about the native methods arguments

---

- JNI passes objects to native methods as opaque references
- Opaque references are C pointer types that refer to internal data structures in the JVM
- Let us consider the following Java class:

```
package com.marakana.jniexamples;

public class HelloName {
  public static native void sayHelloName(String name);

  static { System.loadLibrary("helloname"); }
```

```
  public static void main (String[] args) {
    HelloName hello = new HelloName();
        String name = "John";
    hello.sayHelloName(name);
  }
}
```

- The `.h` file would look like this:

```
...
#include <jni.h>
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName
  (JNIEnv *, jclass, jstring);
...
```

- A `.c` file like this one would not produce the expected result:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv * ↩
    env, jclass class, jstring name){
        printf("Hello %s", name);
}
```

2.3.7 Native Method Arguments

- All native method implementation accepts two standard parameters:
  - `JNIEnv *env`: Is a pointer that points to another pointer pointing to a function table (array of pointer). Each entry in this function table points to a JNI function. These are the functions we are going to use for type conversion
  - The second argument is different depending on whether the native method is a static method or an instance method
    - Instance method: It will be a `jobject` argument which is a reference to the object on which the method is invoked
    - Static method: It will be a `jclass` argument which is a reference to the class in which the method is define

2.3.8 String Conversion

- We just talked about the `JNIEnv *env` that will be the argument to use where we will find the type conversion methods
- There are a lot of methods related to strings:
  - Some are to convert `java.lang.String` to C string: `GetStringChars` (Unicode format), `GetStringUTFChars` (UTF-8 format)
  - Some are to convert `java.lang.String` to C string: `NewString` (Unicode format), `NewStringUTF` (UTF-8 format)
  - Some are to release memory on C string: `ReleaseStringChars`, `ReleaseStringUTFChars`

Note
Details about these methods can be found at
http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html

- If you remember the previous example, we had a native method where we wanted to display "Hello *name*":

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv * ↩
    env, jclass class, jstring name) {
        printf("Hello %s", name); //❶
}
```

❶ This example would not work since the `jstring` type represents strings in the Java virtual machine. This is different from the C string type (`char *`)

- Here is what you would do, using UTF-8 string for instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv * ↩
    env, jclass class, jstring name){
        const jbyte *str;
        str = (*env)->GetStringUTFChars(env, name, NULL); //❶
        printf("Hello %s\n", str);
        (*env)->ReleaseStringUTFChars(env, name, str); //❷
}
```

❶ This returns a pointer to an array of bytes representing the string in UTF-8 encoding (without making a copy)

❷ When we are not making a copy of the string, calling `ReleaseStringUTFChars` prevents the memory area used by the string to stay "pinned". If the data was copied, we need to call `ReleaseStringUTFChars` to free the memory which is not used anymore

- Here is another example where we would construct and return a `java.lang.String` instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_GetName.h"

JNIEXPORT jstring JNICALL Java_com_marakana_jniexamples_ReturnName_GetName(JNIEnv * ↩
    env, jclass class) {
        char buffer[20];
        scanf("%s", buffer);
        return (*env)->NewStringUTF(env, buffer);
}
```

2.3.9 Array Conversion

- Here we are going to focus on primitive arrays only since they are different from objects arrays in JNI
- Arrays are represented in JNI by the `jarray` reference type and its "subtypes" such as `jintArray` ⇒ A `jarray` is not a C array!
- Again we will use the `JNIEnv *env` parameter to access the type conversion methods
  - `Get<Type>ArrayRegion`: Copies the contents of primitive arrays to a preallocated C buffer. Good to use when the size of the array is known
  - `Get<Type>ArrayElements`: Gets a pointer to the content of the primitive array
  - `New<Type>Array`: To create an array specifying a length

- We are going to see an example of how to read a Java primitive array in the native world
- First, this would be your Java program:

```
package com.marakana.jniexamples;

public class ArrayReader {
        private static native int sumArray(int[] arr); //❶
        public static void main(String[] args) {
                //Array declaration
                int arr[] = new int[10];
                //Fill the array
                for (int i = 0; i < 10; i++) {
                        arr[i] = i;
                }
                ArrayReader reader = new ArrayReader();
                //Call native method
                int result = reader.sumArray(arr); //❷
                System.out.println("The sum of every element in the array is " +  ↩
                    Integer.toString(result));
        }
        static {
                System.loadLibrary("arrayreader");
        }
}
```

❶, ❷ This method will return the sum of each element in the array

- After running `javah`, create your `.c` file that would look like this:

```
#include <stdio.h>
#include "com_marakana_jniexamples_ArrayReader.h"

JNIEXPORT jint JNICALL Java_com_marakana_jniexamples_ArrayReader_sumArray(JNIEnv *env ↩
    , jclass class, jintArray array) {
        jint *native_array;
        jint i, result = 0;
        native_array = (*env)->GetIntArrayElements(env, array, NULL); /* ❶ */
        if (native_array == NULL) {
```

```
            return 0;
        }
        for (i=0; i<10; i++) {
            result += native_array[i];
        }
        (*env)->ReleaseIntArrayElements(env, array, native_array, 0);
        return result;
}
```

❶ We could also have used `GetIntArrayRegion` since we exactly know the size of the array

2.3.10 Throwing Exceptions In The Native World

- We are about to see how to throw an exception from the native world
- Throwing an exception from the native world involves the following steps:
  - Find the exception class that you want to throw
  - Throw the exception
  - Delete the local reference to the exception class
- We could imagine a utility function like this one:

```
void ThrowExceptionByClassName(JNIEnv *env, const char *name, const char *message) {
  jclass class = (*env)->FindClass(env, name); //❶
  if (class != NULL) {
      (*env)->ThrowNew(env, class, message); //❷
  }
  (*env)->DeleteLocalRef(env, class); //❸
}
```

❶ Find exception class by its name

❷ Throw the exception using the class reference we got before and the message for the exception

❸ Delete local reference to the exception class

- Here would be how to use this utility method:

```
ThrowExceptionByClassName(env,"java/lang/IllegalArgumentException","This exception is ↩
    thrown from C code");
```

2.3.11 Access Properties And Methods From Native Code

- You might want to modify some properties or call methods of the instance calling the native code
- It always starts with this operation: Getting a reference to the object class by calling the `GetObjectClass` method
- We are then going to get instance field id or an instance method id from the class reference using `GetFieldID` or `GetMethodID` methods
- For the rest, it differs depending on whether we are accessing a field or a method

- From this Java class, we will see how to call its methods or access its properties in the native code:

```
package com.marakana.jniexamples;

public class InstanceAccess {
        public String name; //❶

        public void setName(String name) { //❷
                this.name = name;
        }

        //Native method
        public native void propertyAccess(); //❸
        public native void methodAccess(); //❹

        public static void main(String args[]) {
                InstanceAccess instanceAccessor = new InstanceAccess();
                //Set the initial value of the name property
                instanceAccessor.setName("Jack");
                System.out.println("Java: value of name = \""+ instanceAccessor.name  ↩
                    +"\"");
                //Call the propetyAccess() method
                System.out.println("Java: calling propertyAccess() method...");
                instanceAccessor.propertyAccess(); //❺
                //Value of name after calling the propertyAccess() method
                System.out.println("Java: value of name after calling propertyAccess ↩
                    () = \""+ instanceAccessor.name +"\"");
                //Call the methodAccess() method
                System.out.println("Java: calling methodAccess() method...");
                instanceAccessor.methodAccess(); //❻
                System.out.println("Java: value of name after calling methodAccess()  ↩
                    = \""+ instanceAccessor.name +"\"");
        }

        //Load library
        static {
                System.loadLibrary("instanceaccess");
        }
}
```

❶ Name property that we are going to modify along this code execution

❷ This method will be called by the native code to modify the name property

❸, ❺ This native method modifies the name property by directly accessing the property

❹, ❻ This native method modifies the name property by calling the Java method `setName()`

- This would be our C code for native execution:

```
#include <stdio.h>
#include "com_marakana_jniexamples_InstanceAccess.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess_propertyAccess( ↩
    JNIEnv *env, jobject object){
        jfieldID fieldId;
        jstring jstr;
        const char *cString;
```

```
        /* Getting a reference to object class */
        jclass class = (*env)->GetObjectClass(env, object); /* ❶ */

        /* Getting the field id in the class */
        fieldId = (*env)->GetFieldID(env, class, "name", "Ljava/lang/String;"); /*  ↩
            ❷ */
        if (fieldId == NULL) {
            return; /* Error while getting field id */
        }

        /* Getting a jstring */
        jstr = (*env)->GetObjectField(env, object, fieldId); /* ❸ */

        /* From that jstring we are getting a C string: char* */
        cString = (*env)->GetStringUTFChars(env, jstr, NULL); /* ❹ */
        if (cString == NULL) {
            return; /* Out of memory */
        }
        printf("C: value of name before property modification = \"%s\"\n", cString);
        (*env)->ReleaseStringUTFChars(env, jstr, cString);

        /* Creating a new string containing the new name */
        jstr = (*env)->NewStringUTF(env, "Brian"); /* ❺ */
        if (jstr == NULL) {
            return; /* Out of memory */
        }
        /* Overwrite the value of the name property */
        (*env)->SetObjectField(env, object, fieldId, jstr); /* ❻ */
}


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess_methodAccess( ↩
    JNIEnv *env, jobject object){
        jclass class = (*env)->GetObjectClass(env, object); /* ❼ */
        jmethodID methodId = (*env)->GetMethodID(env, class, "setName", "(Ljava/lang/ ↩
            String;)V"); /* ❽ */
        jstring jstr;
        if (methodId == NULL) {
            return; /* method not found */
        }
        /* Creating a new string containing the new name */
        jstr = (*env)->NewStringUTF(env, "Nick"); /* ❾ */
        (*env)->CallVoidMethod(env, object, methodId, jstr); /* ❿ */
}
```

❶, ❼ This is getting a reference to the object class

❷ Gets a field Id from the object class, specifying the property to get and the internal type. you can find information on the jni type there: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/types.html

❸ This will return the value of the property in the native type: here a jstring

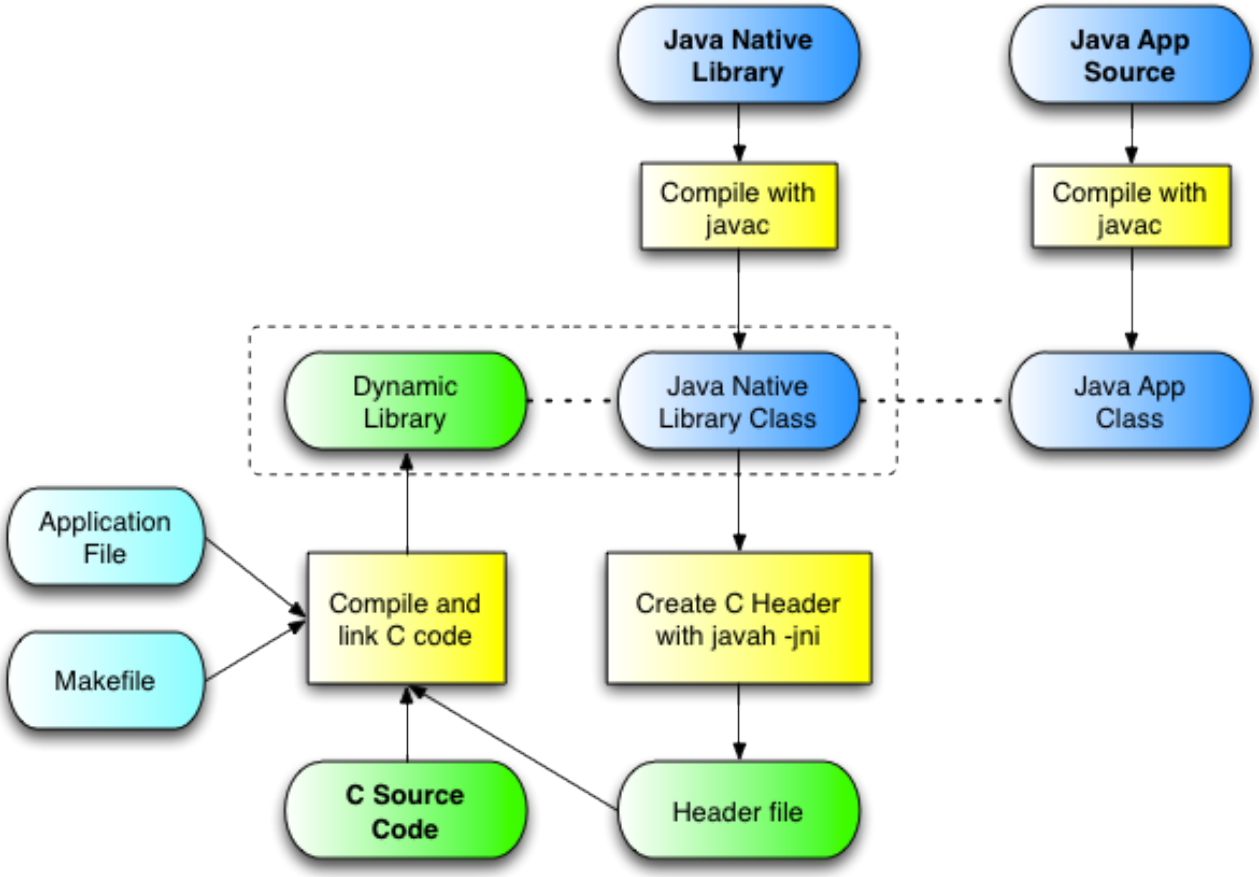
❹ We need to convert the jstring to a C string

❺ This creates a new java.lang.String that is going be use to change the value of the property

❻ This sets the property to its new value

8. Gets a method id from the object class previously obtained, specifying the name of the method along with its signature. There is a very useful java tool that you can use to get the signature of a method: `javap -s -p ClassName` for instance `javap -s -p InstanceAccess`
9. This creates a new java.lang.String that we are going to use as an argument when calling the java method from native code
10. Calling `CallVoidMethod` since the Java method return type is `void` and we are passing the previously created `jstring` as a parameter

2.4 Using NDK



2.5 Fibonacci Example Overview

- Start by creating a new Android Project
  - Project Name: FibonacciNative
  - Build Target: Android 2.2 (API 8)
  - Application Name: Fibonacci Native
  - Package: `com.marakana.android.fibonaccinative`
  - Create Activity: `FibonacciActivity`

2.5.1 Fibonacci - Java Native Function Prototypes

We start off by defining C function prototypes as `native` Java methods (wrapped in some class):

FibonacciNative/src/com/marakana/android/fibonaccinative/FibLib.java +

```
package com.marakana.android.fibonaccinative;

public class FibLib {

    public static long fibJR(long n) { // ❶
        return n <= 0 ? 0 : n == 1 ? 1 : fibJR(n - 1) + fibJR(n - 2);
    }

    public static long fibJI(long n) { // ❷
        long previous = -1;
        long result = 1;
        for (long i = 0; i <= n; i++) {
            long sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }

    public native static long fibNR(long n); // ❸

    public native static long fibNI(long n); // ❹

    static {
        System.loadLibrary("com_marakana_android_fibonaccinative_FibLib"); // ❺
    }
}
```

❶ Recursive Java implementation of the Fibonacci algorithm (included for comparison only)

❷ Iterative Java implementation of the Fibonacci algorithm (included for comparison only)

❸ Function prototype for future native recursive implementation of the Fibonacci algorithm

❹ Function prototype for future iterative recursive implementation of the Fibonacci algorithm

❺ Use `System.loadLibrary()` to load the native module (to be compiled)

2.5.2 Fibonacci - Function Prototypes in a C Header File

We then extract our C header file with our function prototypes:

1. On the command line, change to your project's root directory

   ```
   $> cd /path/to/workspace/FibonacciNative
   ```

2. Create `jni` sub-directory

   ```
   $> mkdir jni
   ```

3. Extract the C header file from `com.marakana.android.fibonaccinative.FibLib` class:

```
$> javah -jni -classpath bin -d jni com.marakana.android.fibonaccinative.FibLib
```

This produces:

FibonacciNative/jni/com_marakana_android_fibonaccinative_FibLib.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_marakana_android_fibonaccinative_FibLib */

#ifndef _Included_com_marakana_android_fibonaccinative_FibLib
#define _Included_com_marakana_android_fibonaccinative_FibLib
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_marakana_android_fibonaccinative_FibLib
 * Method:    fibNR
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNR
  (JNIEnv *, jclass, jlong);

/*
 * Class:     com_marakana_android_fibonaccinative_FibLib
 * Method:    fibNI
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNI
  (JNIEnv *, jclass, jlong);

#ifdef __cplusplus
}
#endif
#endif
```

Note
The function prototype names are name-spaced to the classname they are found in.

2.5.3 Fibonacci - Provide C Implementation

We provide the C implementation of `com_marakana_android_fibonacci_FibLib.h` header file:

FibonacciNative/jni/com_marakana_android_fibonaccinative_FibLib.c +

```
#include "com_marakana_android_fibonaccinative_FibLib.h" /* ❶ */

jlong fib(jlong n) { /* ❷ */
    return n <= 0 ? 0 : n == 1 ? 1 : fib(n - 1) + fib(n - 2);
}

JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNR
  (JNIEnv *env, jclass clazz, jlong n) { /* ❸ */
```

```
    return fib(n);
}

JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNI
  (JNIEnv *env, jclass clazz, jlong n) { /* ❹ */
    jlong previous = -1;
    jlong result = 1;
    jlong i;
    for (i = 0; i <= n; i++) {
        jlong sum = result + previous;
        previous = result;
        result = sum;
    }
    return result;
}
```

❶ Include the header file that was created via `javah -jni` command.

❷ Recursive implementation of the fibonacci algorithm (in a helper function)

❸ Actual implementation of JNI-defined `fibNR` (recursive) function

❹ Actual implementation of JNI-defined `fibNI` (iterative) function

2.5.4 Fibonacci - Makefile

We need a `Android.mk` makefile, which will be used by NDK to compile our JNI code into a shared library:

FibonacciNative/jni/Android.mk

```
# ❶
LOCAL_PATH := $(call my-dir)

# ❷
include $(CLEAR_VARS)

# ❸
LOCAL_SRC_FILES := com_marakana_android_fibonaccinative_FibLib.c

# ❹
LOCAL_MODULE    := com_marakana_android_fibonaccinative_FibLib

# ❺
include $(BUILD_SHARED_LIBRARY)
```

❶ `LOCAL_PATH` is used to locate source files in the development tree. An `Android.mk` file must begin with the definition of this variable. The macro function `my-dir`, provided by the build system, specifies the path of the current directory (i.e. the directory containing the `Android.mk` file itself).

❷ `include $(CLEAR_VARS)` clears many `LOCAL_XXX` variables with the exception of `LOCAL_PATH` (this is needed because all variables are global)

❸ `LOCAL_SRC_FILES` lists all of our C files to be compiled (header file dependencies are automatically computed)

4. `LOCAL_MODULE` defines the name of our shared module (this name will be prepended by `lib` and postfixed by `.so`)
5. `include $(BUILD_SHARED_LIBRARY)` collects all `LOCAL_XXX` variables since `include $(CLEAR_VARS)` and determines what to build (in this case a shared library)

Note
It's easiest to copy the `Android.mk` file from another (sample) project and adjust `LOCAL_SRC_FILES` and `LOCAL_MODULE` as necessary

Note
See `/path/to/ndk-installation-dir/docs/ANDROID-MK.html` for the complete reference of Android make files (build system)

2.5.5 Fibonacci - Compile Our Shared Module

Finally, from the root of our project (i.e. `FibonacciNative/`), we run `ndk-build` to build our code into a shared library (`FibonacciNative/libs/armeabi/libcom_marakana_android_fibonacci_FibLib.so`):

```
$> ndk-build
Compile thumb  : com_marakana_android_fibonaccinative_FibLib <=  ↩
    com_marakana_android_fibonaccinative_FibLib.c
SharedLibrary  : libcom_marakana_android_fibonaccinative_FibLib.so
Install        : libcom_marakana_android_fibonaccinative_FibLib.so => libs/armeabi/ ↩
    libcom_marakana_android_fibonaccinative_FibLib.so
```

Running `ndk-build clean` will clean all generated binaries.

Note
The command `ndk-build` comes from the NDK's installation directory (e.g. `/path/to/android-ndk-r5b`), so it's easiest if we add this directory to our `PATH`.

Note
The current version of NDK (at least up to r5b) on Windows depends on Cygwin (a Unix-like environment and command-line interface for Microsoft Windows), or specifically "shell" (bash) and "make" (gmake) programs available through Cygwin. To run `ndk-build` on Windows, we first need to run `bash` and then execute `ndk-build`. It is important that both `c:\path\to\cygwin\bin` and `c:\path\to\ndk` be defined in our `Path`.

2.5.5.1 Controlling CPU Application Binary Interface (ABI)

- By default, the NDK will generate machine code for the *armeabi* (i.e. ARMv5TE with support for Thumb-1) ABI
  - The target application library is packaged as `lib/armeabi/lib<name>.so`
  - Upon installation, it is copied to `/data/data/<package>/lib/lib<name>.so`
- We could add support for ARMv7-A (including hardware FPU/VFPv3-D16, Thumb-2, VFPv3-D32/ThumbEE, and SIMD/NEON) via `APP_ABI` in a separate `Application.mk` file

  - `APP_ABI := armeabi-v7a`
    * ARMv7-a only
    * Packaged as `lib/armeabi-v7a/lib<name>.so`
  - `APP_ABI := armeabi armeabi-v7a x86`
    * Builds **three** versions of the library, for ARMv5TE, ARMv7-a, and x86 in a single "fat binary"
    * Packaged as
      · `lib/armeabi/lib<name>.so`
      · `lib/armeabi-v7a/lib<name>.so`
      · `lib/x86/lib<name>.so`
    * The Android system knows at runtime which ABI(s) it supports
      · *primary* ABI for the device corresponds to the machine code of the system image
      · *secondary* ABI (optional) corresponds to to another ABI that is also supported by the system image
      · Upon installation, Android first scans `lib/<primary-abi>/lib<name>.so` then `lib/<secondary-abi>/lib<name>` and installs the first that it finds to `/data/data/<package>/lib/lib<name>.so`

- Support for `x86` ABI (i.e. IA-32 instruction set) was added in NDK r6 (July 2011)

2.5.6 Fibonacci - Client

We can now build the "client" of our library (in this case a simple activity) to use our `FibLib` library.

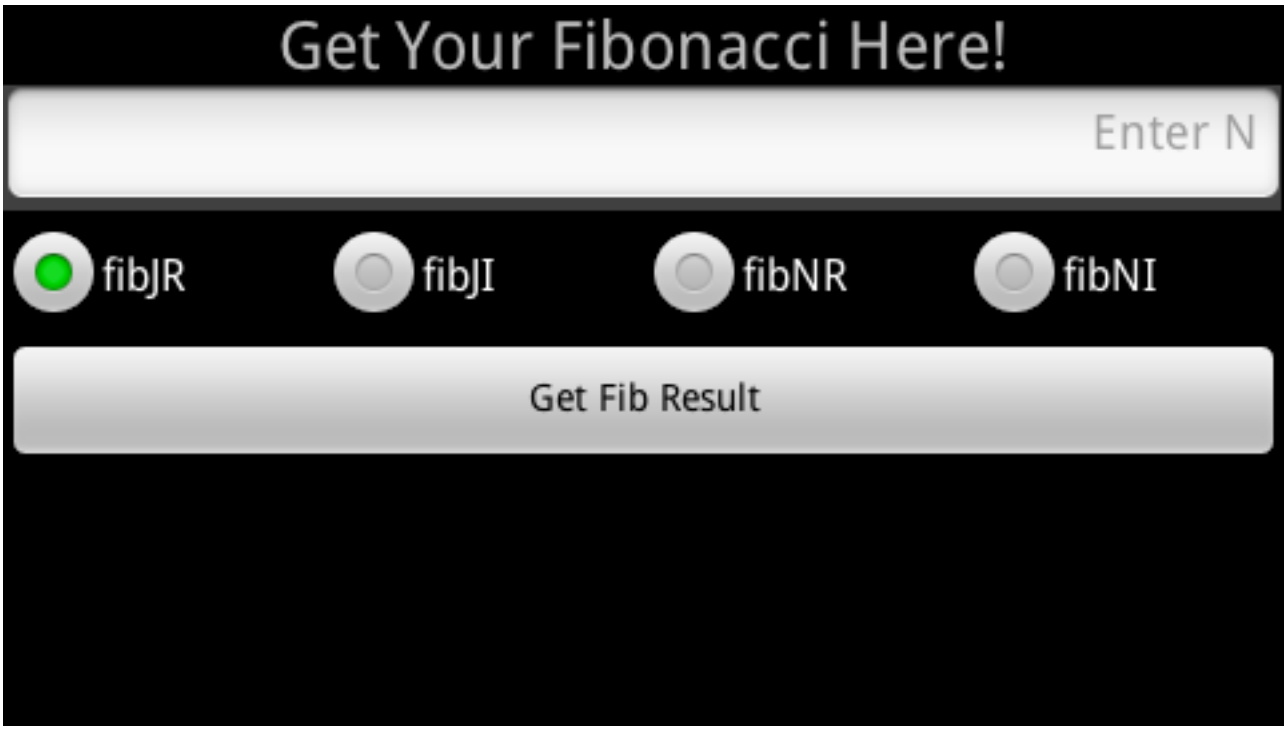
2.5.6.1 Fibonacci - String Resources

FibonacciNative/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <string name="hello">Get Your Fibonacci Here!</string>
        <string name="app_name">Fibonacci Native</string>
        <string name="input_hint">Enter N</string>
        <string name="input_error">Numbers only!</string>
        <string name="button_text">Get Fib Result</string>
        <string name="progress_text">Calculating...</string>

        <string name="type_fib_jr">fibJR</string>
        <string name="type_fib_ji">fibJI</string>
        <string name="type_fib_nr">fibNR</string>
        <string name="type_fib_ni">fibNI</string>
</resources>
```

2.5.6.2 Fibonacci - User Interface (Layout)



FibonacciNative/res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TextView android:text="@string/hello" android:layout_height="wrap_content"
                android:layout_width="fill_parent" android:textSize="25sp"
                android:gravity="center" /> <!-- ❶ -->
        <EditText android:layout_height="wrap_content"
                android:layout_width="match_parent" android:id="@+id/input"
                android:hint="@string/input_hint" android:inputType="number"
                android:gravity="right" />  <!-- ❷ -->
        <RadioGroup android:orientation="horizontal"
                android:layout_width="match_parent" android:id="@+id/type"
                android:layout_height="wrap_content">  <!-- ❸ -->
                <RadioButton android:layout_height="wrap_content"
                        android:checked="true" android:id="@+id/type_fib_jr"  ↩
                            android:text="@string/type_fib_jr"
                        android:layout_width="match_parent" android:layout_weight="1"  ↩
                            />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_ji" android:text="@string/ ↩
                            type_fib_ji"
                        android:layout_width="match_parent" android:layout_weight="1"  ↩
                            />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_nr" android:text="@string/ ↩
                            type_fib_nr"
```

```
                        android:layout_width="match_parent" android:layout_weight="1"  ↩
                            />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_ni" android:text="@string/ ↩
                            type_fib_ni"
                        android:layout_width="match_parent" android:layout_weight="1"  ↩
                            />
        </RadioGroup>
        <Button android:text="@string/button_text" android:id="@+id/button"
                android:layout_width="match_parent" android:layout_height=" ↩
                    wrap_content" />  <!-- ❹ -->
        <TextView android:id="@+id/output" android:layout_width="match_parent"
                android:layout_height="match_parent" android:textSize="20sp"
                android:gravity="center" />  <!-- ❺ -->
</LinearLayout>
```

❶ This is just a simple title ("Get Your Fibonacci Here!")

❷ This is the entry box for our number *n*

❸ This radio group allows the user to select the fibonacci implementation type

❹ This button allows the user to trigger fibonacci calculation

❺ This is the output area for the fibonacci result

2.5.6.3 Fibonacci - `FibonacciActivity`

FibonacciNative/src/com/marakana/android/fibonaccinative/FibonacciActivity.java

```
package com.marakana.android.fibonaccinative;

import android.app.Activity;
import android.app.ProgressDialog;
import android.os.AsyncTask;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;

public class FibonacciActivity extends Activity implements OnClickListener {

    private EditText input; // our input n

    private RadioGroup type; // fibonacci implementation type

    private TextView output; // destination for fibonacci result

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        super.setContentView(R.layout.main);
```

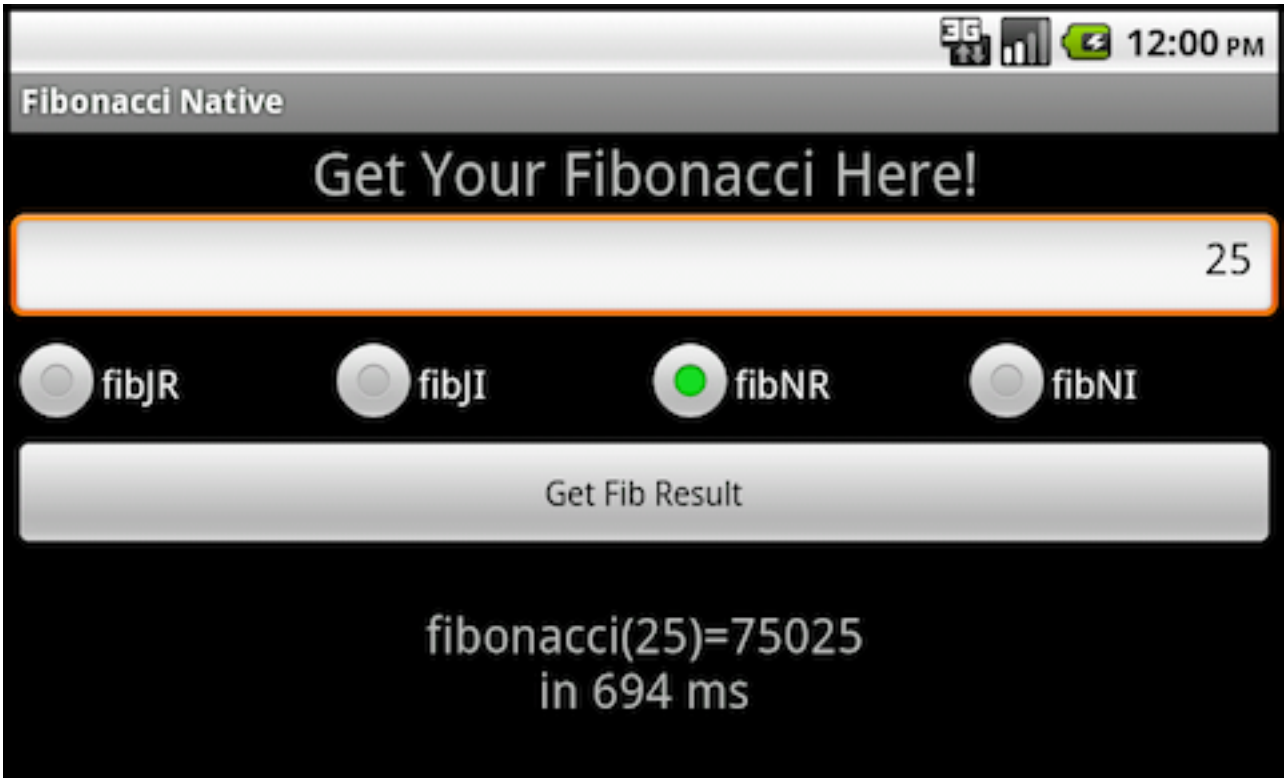
```
        // connect to our UI elements
        this.input = (EditText)super.findViewById(R.id.input);
        this.type = (RadioGroup)super.findViewById(R.id.type);
        this.output = (TextView)super.findViewById(R.id.output);
        Button button = (Button)super.findViewById(R.id.button);
        // request button click call-backs via onClick(View) method
        button.setOnClickListener(this);
    }

    // handle button clicks
    public void onClick(View view) {
        // parse n from input (or report errors)
        final long n;
        String s = this.input.getText().toString();
        if (TextUtils.isEmpty(s)) {
            return;
        }
        try {
            n = Long.parseLong(s);
        } catch (NumberFormatException e) {
            this.input.setError(super.getText(R.string.input_error));
            return;
        }
        // showing the user that the calculation is in progress
        final ProgressDialog dialog = ProgressDialog.show(this, "", super
                .getText(R.string.progress_text), true);
        // since the calculation can take a long time, we do it in a separate
        // thread to avoid blocking the UI
        new AsyncTask<Void, Void, String>() {
            @Override
            protected String doInBackground(Void... params) {
                // this method runs in a background thread
                long result = 0;
                long t = System.currentTimeMillis(); // measure the time
                // call into our library (based on the type selection)
                switch (FibonacciActivity.this.type.getCheckedRadioButtonId()) {
                    case R.id.type_fib_jr:
                        result = FibLib.fibJR(n);
                        break;
                    case R.id.type_fib_ji:
                        result = FibLib.fibJI(n);
                        break;
                    case R.id.type_fib_nr:
                        result = FibLib.fibNR(n);
                        break;
                    case R.id.type_fib_ni:
                        result = FibLib.fibNI(n);
                        break;
                }
                // measure the time difference
                t = System.currentTimeMillis() - t;
                // generate the result
                return String.format("fibonacci(%d)=%d\nin %d ms", n, result, t);
            }

            @Override
```

```
            protected void onPostExecute(String result) {
                // get rid of the dialog
                dialog.dismiss();
                // show the result to the user
                FibonacciActivity.this.output.setText(result);
            }
        }.execute(); // run our AsyncTask
    }
}
```

2.5.7 Fibonacci - Result



2.6 NDK's Stable APIs

2.6.1 Android-specific Log Support

- Include `<android/log.h>` to access various functionality that can be used to send log messages to the kernel (i.e. logcat buffers) from our native code
- Requires that our code be linked to `/system/lib/liblog.so` with `LOCAL_LDLIBS += -llog` in our `Android.mk` file

2.6.2 ZLib Compression Library

- Include `<zlib.h>` and `<zconf.h>` to access ZLib compression library

  - See `http://www.zlib.net/manual.html` for more info on ZLib
- Requires that our code be linked to `/system/lib/libz.so` with `LOCAL_LDLIBS += -lz` in our `Android.mk` file

2.6.3 The OpenGL ES 1.x Library

- Include `<GLES/gl.h>` and `<GLES/glext.h>` to access OpenGL ES 1.x rendering calls from native code
  - The "1.x" here refers to both versions 1.0 and 1.1
    - Using 1.1 requires OpenGL-capable GPU
    - Using 1.0 is universally supported since Android includes software renderer for GPU-less devices
    - Requires that we include `<uses-feature>` tag in our manifest file to indicate the actual OpenGL version that we expect
- Requires that our code be linked to `/system/lib/libGLESv1_CM.so` with `LOCAL_LDLIBS += -lGLESv1_CM.so` in our `Android.mk` file
- Since API 4 (Android 1.6)

2.6.4 The OpenGL ES 2.0 Library

- Include `<GLES2/gl2.h>` and `<GLES2/gl2ext.h>` to access OpenGL ES 2.0 rendering calls from native code
  - Enables the use of vertex and fragment shaders via the GLSL language
  - Since not all devices support OpenGL 2.0, we should include `<uses-feature>` tag in our manifest file to indicate this requirement
- Requires that our code be linked to `/system/lib/libGLESv2.so` with `LOCAL_LDLIBS += -lGLESv2.so` in our `Android.mk` file
- Since API 4 (Android 2.0)

2.6.5 The `jnigraphics` Library

- Include `<android/bitmap.h>` to reliably access the pixel buffers of Java bitmap objects from native code
- Requires that our code be linked to `/system/lib/libjnigraphics.so` with `LOCAL_LDLIBS += -ljnigraphics` in our `Android.mk` file
- Since API 8 (Android 2.2)

2.6.6 The OpenSL ES native audio Library

- Include `<SLES/OpenSLES.h>` and `<SLES/OpenSLES_Platform.h>` to perform audio input and output from native code
  - Based on Khronos Group OpenSL ES™ 1.0.1
- Requires that our code be linked to `/system/lib/libOpenSLES.so` with `LOCAL_LDLIBS += -lOpenSLES` in our `Android.mk` file
- Since API 9 (Android 2.3)

2.6.7 The Android native application APIs

- Makes it possible to write our entire application in native code
  - Mainly added for gaming
  - Our code still depends on the Dalvik VM since most of the platform features are managed in the VM and accessed via JNI (Native → Java)
- Include `<android/native_activity.h>` to write an Android activity (with its life-cycle callbacks) in native code
  - A native activity would serve as the main entry point into our native application
- Include `<android/looper.h>`, `<android/input.h>`, `<android/keycodes.h>`, and `<android/sensor.h>` to listen to input events and sensors directly from native code
- Include `<android/rect.h>`, `<android/window.h>`, `<android/native_window.h>`, and `<android/native_window` for window management from native code
  - Includes ability to lock/unlock the pixel buffer to draw directly into it
- Include `<android/configuration.h>`, `<android/asset_manager.h>`, `<android/storage_manager.h>`, and `<android/obb.h>` for direct access to the assets embedded in our `.apk` files Opaque Binary Blob (OBB) files
  - All access is read-only
- Requires that our code be linked to `libandroid.so` with `LOCAL_LDLIBS += -landroid` in our `Android.mk` file
- Since API 9 (Android 2.3)

Caution
With the exception of the libraries listed above, the native system libraries in the Android platform are not considered "stable" and may change in future platform versions. Unless our library is being built for a specific Android ROM, we should only make use of the stable libraries provided by the NDK.

Note
All the header files are available under `/path/to/ndk-installation-dir/platforms/android-9/arch-arm/usr/incl`

Note
See `/path/to/ndk-installation-dir/docs/STABLE-APIS.html` for the complete reference of NDK's stable APIs.

2.7 Lab: NDK

Create a simple Android application that allows the end-user to log to Android's logcat via functionality provided by `/system/lib/liblog.so` (i.e. you cannot use `android.util.Log`).

For example, this could be your `LogLib`:

```
package com.marakana.android.loglib;
public class LogLib {
        public static native void log(int priority, String tag, String message);
        /* You'll need to load your library here */
}
```

It is up to you to design the UI, but it could be as simple as three `TextView` widgets (one for `priority`, one for `tag`, and one for `message`) and one `Button` widget to submit the message to the log.

Tip
Don't forget to convert `tag` and `message` strings from the Java format (`jstring`) to native format (`char *`) before trying to use them in `__android_log_write`. Also, don't forget to *free* the native strings before returning from the native method.

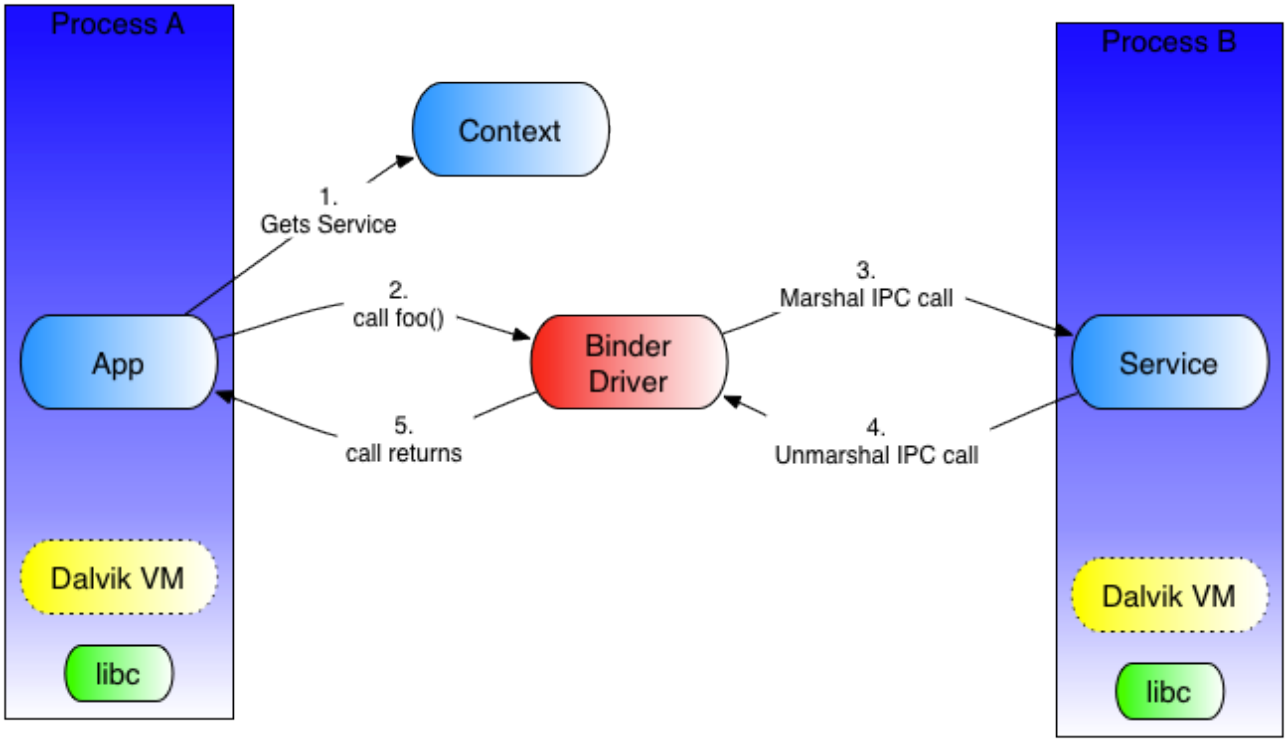
Chapter 3

Android Binder Inter Process Communication (IPC) with AIDL

3.1 Why IPC?

- Each Android application runs in a separate process
  - Android application-framework services also run in a separate process called `systemserver`
- Often we wish to make use of services offered by other applications, or we simply wish to expose our applications' capabilities to 3rd party apps
  - Even Android's `Intent`-based IPC-like mechanism (used for starting activities and services, as well as delivering events), is internally based on Binder
- By design (for security reasons), processes cannot directly access each other's data
- To cross the process boundaries, we need support of a inter-process communication transport mechanism, which handles passing of data from one process (caller) to another (callee)
  - Caller's data is *marshaled* into tokens that IPC understands, copied to callee's process, and finally *unmarshaled* into what callee expects
  - Callee's response is also *marshaled*, copied to caller's process where it is *unmarshaled* into what caller expects
  - Marshaling/unmarshaling is automatically provided by the IPC mechanism

3.2 What is Binder?



- Binder provides a lightweight remote procedure call (RPC) mechanism designed for high performance when performing in-process and cross-process calls (IPC)
- Binder-capable services are described in Android Interface Definition Language (AIDL), not unlike other IDL languages
- Since Binder is provided as a Linux driver, the services can be written in both C/C++ as well as Java
  - Most Android services are written in Java
- All caller calls go through Binder's `transact()` method, which automatically marshals the arguments and return values via `Parcel` objects
  - `Parcel` is a generic buffer of data (decomposed into primitives) that also maintains some meta-data about its contents - such as object references to ensure object identity across processes
  - Caller calls to `transact()` are by default synchronous - i.e. provide the same semantics as a local method call
    - On callee side, the Binder framework maintains a pool of transaction threads, which are used to handle the incoming IPC requests (unless the call is local, in which case the same thread is used)
    - Callee methods can be marked as `oneway`, in which case caller calls do not block (i.e. calls return immediately)
- Callee' mutable state needs to be thread-safe - since callee's can accept concurrent requests from multiple callers
- The Binder system also supports recursion across processes - i.e. behaves the same as recursion semantics when calling methods on local objects

3.3 What is AIDL?

- Android Interface Definition Language is a Android-specific language for defining Binder-based service interafaces
- AIDL follows Java-like interface syntax and allows us to declare our "business" methods
- Each Binder-based service is defined in a separate `.aidl` file, typically named `IFooService.aidl`, and saved in the `src/` directory

  src/com/example/app/IFooService.aidl

```
package com.example.app;
import com.example.app.Bar;
interface IFooService {
    void save(inout Bar bar);
    Bar getById(int id);
    void delete(in Bar bar);
    List<Bar> getAll();
}
```

- The `aidl` build tool (part of Android SDK) is used to extract a real Java interface (along with a `Stub` providing Android's `android.os.IBinder`) from each `.aidl` file and place it into our `gen/` directory

  gen/com/example/app/IFooService.java

```
package com.example.app;
public interface IFooService extends android.os.IInterface
{
    public static abstract class Stub extends android.os.Binder
        implements com.example.app.IFooService {
        ...
        public static com.example.app.IFooService asInterface(
            android.os.IBinder obj) {
            ...
        }
        public android.os.IBinder asBinder() {
            return this;
        }
        ...
    }
    void save(com.example.app.Bar bar) throws android.os.RemoteException;
    com.example.app.Bar getById(int id) throws android.os.RemoteException;
    void delete(com.example.app.Bar bar) throws android.os.RemoteException;
    java.util.List<Bar> getAll() throws android.os.RemoteException;
}
```

> **Note**
> Eclipse ADT automatically calls `aidl` for each `.aidl` file that it finds in our `src/` directory

- AIDL supports the following types
  - Java primitives: `boolean`, `char`, `short`, `int`, `long`, `float`, `double`, and `void`
  - `java.lang.String` and `java.lang.CharSequence`
  - An array of other supported types

- `java.util.List` where all elements are of supported AIDL types
  - Generic definitions are supported
    - Internally, Binder always uses `java.util.ArrayList` as the concrete implementation
- `java.util.Map` where all elements are of supported AIDL types
  - Generic definitions are *not* supported
    - Internally, Binder always uses `java.util.HashMap` as the concrete implementation
- Any custom classes implementing `android.os.Parcelable` interface
  src/com/example/app/Bar.java

```
package com.example.app;

import android.os.Parcel;
import android.os.Parcelable;

public class Bar implements Parcelable {
    private int id;
    private String data;

    public Bar(Parcel parcel) {
        this.id = parcel.readInt();
        this.data = parcel.readString();
    }

    // getters and setters omitted
    ...

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeInt(this.id);
        parcel.writeString(this.data);
    }

    public static final Parcelable.Creator<Bar> CREATOR = new Parcelable.Creator< ↩
        Bar>() {
        public Bar createFromParcel(Parcel in) {
            return new Bar(in);
        }
        public Bar[] newArray(int size) {
            return new Bar[size];
        }
    };
}
```

  - These custom classes have to be declared in their own (simplified) `.aidl` files
    src/com/example/app/Bar.aidl

```
package com.example.app;
parcelable Bar;
```

> **Note**
> AIDL-interfaces have to `import` parcelable custom classes even if they are in the same package. In the case of the previous example, `src/com/example/app/IFooService.aidl` would have to `import com.example.app.Bar;` if it makes any references to `com.example.app.Bar` even though they are in the same package.

- AIDL-defined methods can take zero or more parameters, and must return a value or `void`
  - All non-primitive parameters require a *directional tag* indicating which way the data goes: one of: `in`, `out`, or `inout`
    - Direction for primitives is always `in` (can be omitted)
    - The direction tag tells binder when to marshal the data, so its use has direct consequences on performance
- All `.aidl` comments are copied over to the generated Java interface (except for comments before the import and package statements).
- Static fields are not supported in `.aidl` files

3.4 Building a Binder-based Service and Client

- To demonstrate an Binder-based service and client, we'll create three separate projects:
  1. `FibonacciCommon` library project - to define our AIDL interface as well as custom types for parameters and return values
  2. `FibonacciService` project - where we implement our AIDL interface and expose it to the clients
  3. `FibonacciClient` project - where we connect to our AIDL-defined service and use it

3.5 FibonacciCommon - Define AIDL Interface and Custom Types

- We start by creating a new Android (library) project, which will host the common API files (an AIDL interface as well as custom types for parameters and return values) shared by the service and its clients
  - Project Name: `FibonacciCommon`
  - Build Target: Android 2.2 (API 8)
  - Package Name: `com.marakana.android.fibonaccicommon`
  - Min SDK Version: 8
  - No need to specify Application name or an activity
- To turn this into a *library project* we need to access project properties → `Android` → `Library` and check `Is Library`
  - We could also manually add `android.library=true` to `FibonacciCommon/default.properties` and refresh the project
- Since library projects are never turned into actual applications (APKs)
  - We can simplify our manifest file:
    FibonacciCommon/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.marakana.android.fibonaccicommon" android:versionCode="1"
        android:versionName="1.0">
</manifest>
```

  - And we can remove everything from `FibonacciCommon/res/` directory (e.g. `rm -fr FibonacciCommon/res/*`)

- We are now ready to create our AIDL interface

  FibonacciCommon/src/com/marakana/android/fibonaccicommon/IFibonacciService.aidl

```
package com.marakana.android.fibonaccicommon;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;

interface IFibonacciService {
    long fibJR(in long n);
    long fibJI(in long n);
    long fibNR(in long n);
    long fibNI(in long n);
    FibonacciResponse fib(in FibonacciRequest request);
}
```

- Our interface clearly depends on two custom Java types, which we have to not only implement in Java, but define in their own `.aidl` files

  FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciRequest.aidl

```
package com.marakana.android.fibonaccicommon;

parcelable FibonacciRequest;
```

  FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciRequest.java

```
package com.marakana.android.fibonaccicommon;

import android.os.Parcel;
import android.os.Parcelable;

public class FibonacciRequest implements Parcelable {

    public static final int RECURSIVE_JAVA_TYPE = 1;

    public static final int ITERATIVE_JAVA_TYPE = 2;

    public static final int RECURSIVE_NATIVE_TYPE = 3;

    public static final int ITERATIVE_NATIVE_TYPE = 4;

    private final long n;

    private final int type;

    public FibonacciRequest(long n, int type) {
        this.n = n;
        if (type < RECURSIVE_JAVA_TYPE || type > ITERATIVE_NATIVE_TYPE) {
            throw new IllegalArgumentException("Invalid type: " + type);
        }
        this.type = type;
    }

    public FibonacciRequest(Parcel parcel) {
        this(parcel.readLong(), parcel.readInt());
    }
```

```
    public long getN() {
        return n;
    }

    public int getType() {
        return type;
    }

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeLong(this.n);
        parcel.writeInt(this.type);
    }

    public static final Parcelable.Creator<FibonacciRequest> CREATOR = new Parcelable ↩
        .Creator<FibonacciRequest>() {
        public FibonacciRequest createFromParcel(Parcel in) {
            return new FibonacciRequest(in);
        }

        public FibonacciRequest[] newArray(int size) {
            return new FibonacciRequest[size];
        }
    };
}
```

FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciResponse.aidl

```
package com.marakana.android.fibonaccicommon;

parcelable FibonacciResponse;
```

FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciResponse.java

```
package com.marakana.android.fibonaccicommon;

import android.os.Parcel;
import android.os.Parcelable;

public class FibonacciResponse implements Parcelable {

    private final long result;

    private final long timeInMillis;

    public FibonacciResponse(long result, long timeInMillis) {
        this.result = result;
        this.timeInMillis = timeInMillis;
    }

    public FibonacciResponse(Parcel parcel) {
        this(parcel.readLong(), parcel.readLong());
    }
```

```
    public long getResult() {
        return result;
    }

    public long getTimeInMillis() {
        return timeInMillis;
    }

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeLong(this.result);
        parcel.writeLong(this.timeInMillis);
    }

    public static final Parcelable.Creator<FibonacciResponse> CREATOR = new  ↩
        Parcelable.Creator<FibonacciResponse>() {
        public FibonacciResponse createFromParcel(Parcel in) {
            return new FibonacciResponse(in);
        }

        public FibonacciResponse[] newArray(int size) {
            return new FibonacciResponse[size];
        }
    };
}
```

- Finally we are now ready to take a look at our generated Java interface

  FibonacciCommon/gen/com/marakana/android/fibonaccicommon/IFibonacciService.java

```
package com.marakana.android.fibonaccicommon;
public interface IFibonacciService extends android.os.IInterface
{
    public static abstract class Stub extends android.os.Binder
        implements com.marakana.android.fibonacci.IFibonacciService {
        ...
        public static com.marakana.android.fibonacci.IFibonacciService asInterface(
            android.os.IBinder obj) {
            ...
        }
        public android.os.IBinder asBinder() {
            return this;
        }
        ...
    }

    public long fibJR(long n) throws android.os.RemoteException;
    public long fibJI(long n) throws android.os.RemoteException;
    public long fibNR(long n) throws android.os.RemoteException;
    public long fibNI(long n) throws android.os.RemoteException;
    public com.marakana.android.fibonaccicommon.FibonacciResponse fib(
        com.marakana.android.fibonaccicommon.FibonacciRequest request)
        throws android.os.RemoteException;
```

```
}
```

3.6 FibonacciService - Implement AIDL Interface and Expose It To Our Clients

- We start by creating a new Android project, which will host the our AIDL Service implementation as well as provide a mechanism to access (i.e. bind to) our service implementation
  - Project Name: `FibonacciService`
  - Build Target: Android 2.2 (API 8)
  - Package Name: `com.marakana.android.fibonacciservice`
  - Application name: Fibonacci Service
  - Min SDK Version: 8
  - No need to specify an Android activity
- We need to link this project to the `FibonacciCommon` in order to be able to access the common APIs: project properties → `Android` → `Library` → `Add...` → `FibonacciCommon`
  - As the result, `FibonacciService/default.properties` now has `android.library.reference.1=../Fibonacci` and `FibonacciService/.classpath` and `FibonacciService/.project` also link to `FibonacciCommon`
- Our service will make use of the `com.marakana.android.fibonaccinative.FibLib`, which provides the actual implementation of the Fibonacci algorithms
- We copy (or move) this Java class (as well as the `jni/` implementation) from the `FibonacciNative` project
  - Don't forget to run `ndk-build` under `FibonacciService/` in order to generate the required native library

3.7 Implement AIDL Interface

- We are now ready to implement our AIDL-defined interface by extending from the auto-generated `com.marakana.android.fibona` (which in turn extends from `android.os.Binder`)

FibonacciService/src/com/marakana/android/fibonacciservice/IFibonacciServiceImpl.java

```
package com.marakana.android.fibonacciservice;

import android.os.RemoteException;
import android.util.Log;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;
import com.marakana.android.fibonaccicommon.IFibonacciService;
import com.marakana.android.fibonaccinative.FibLib;

public class IFibonacciServiceImpl extends IFibonacciService.Stub {
    private static final String TAG = "IFibonacciServiceImpl";

    public long fibJI(long n) throws RemoteException {
        Log.d(TAG, "fibJI()");
        return FibLib.fibJI(n);
    }
```

```
    public long fibJR(long n) throws RemoteException {
        Log.d(TAG, "fibJR()");
        return FibLib.fibJR(n);
    }

    public long fibNI(long n) throws RemoteException {
        Log.d(TAG, "fibNI()");
        return FibLib.fibNI(n);
    }

    public long fibNR(long n) throws RemoteException {
        Log.d(TAG, "fibNR()");
        return FibLib.fibNR(n);
    }

    public FibonacciResponse fib(FibonacciRequest request) throws RemoteException {
        Log.d(TAG, "fib()");
        long timeInMillis = System.currentTimeMillis();
        long result;
        switch (request.getType()) {
            case FibonacciRequest.ITERATIVE_JAVA_TYPE:
                result = FibLib.fibJI(request.getN());
                break;
            case FibonacciRequest.RECURSIVE_JAVA_TYPE:
                result = FibLib.fibJR(request.getN());
                break;
            case FibonacciRequest.ITERATIVE_NATIVE_TYPE:
                result = FibLib.fibNI(request.getN());
                break;
            case FibonacciRequest.RECURSIVE_NATIVE_TYPE:
                result = FibLib.fibNR(request.getN());
                break;
            default:
                return null;
        }
        timeInMillis = System.currentTimeMillis() - timeInMillis;
        return new FibonacciResponse(result, timeInMillis);
    }
}
```

3.8 Expose our AIDL-defined Service Implementation to Clients

- In order for clients (callers) to use our service, they first need to bind to it.
- But in order for them to *bind* to it, we first need to expose it via our own `android.app.Service`'s `onBind(Intent)` implementation

FibonacciService/src/com/marakana/android/fibonacciservice/FibonacciService.java

```
package com.marakana.android.fibonacciservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
```

```
import android.util.Log;

public class FibonacciService extends Service { // ❶

    private static final String TAG = "FibonacciService";

    private IFibonacciServiceImpl service; // ❷

    @Override
    public void onCreate() {
        super.onCreate();
        this.service = new IFibonacciServiceImpl(); // ❸
        Log.d(TAG, "onCreate()'ed"); // ❹
    }

    @Override
    public IBinder onBind(Intent intent) {
        Log.d(TAG, "onBind()'ed"); // ❺
        return this.service; // ❻
    }

    @Override
    public boolean onUnbind(Intent intent) {
        Log.d(TAG, "onUnbind()'ed"); // ❼
        return super.onUnbind(intent);
    }

    @Override
    public void onDestroy() {
        Log.d(TAG, "onDestroy()'ed");
        this.service = null;
        super.onDestroy();
    }
}
```

❶ We create yet another "service" object by extending from `android.app.Service`. The purpose of `FibonacciService` object is to provide access to our Binder-based `IFibonacciServiceImpl` object.

❷ Here we simply declare a local reference to `IFibonacciServiceImpl`, which will act as a singleton (i.e. all clients will share a single instance). Since our `IFibonacciServiceImpl` does not require any special initialization, we could instantiate it at this point, but we choose to delay this until the `onCreate()` method.

❸ Now we instantiate our `IFibonacciServiceImpl` that we'll be providing to our clients (in the `onBind(Intent)` method). If our `IFibonacciServiceImpl` required access to the `Context` (which it doesn't) we could pass a reference to `this` (i.e. `android.app.Service`, which implements `android.content.Context`) at this point. Many Binder-based services use `Context` in order to access other platform functionality.

❻ This is where we provide access to our `IFibonacciServiceImpl` object to our clients. By design, we chose to have only one instance of `IFibonacciServiceImpl` (so all clients share it) but we could also provide each client with their own instance of `IFibonacciServiceImpl`.

❹, ❺, ❼ We just add some logging calls to make it easy to track the life-cycle of our service.

- Finally, we register our `FibonacciService` in our `AndroidManifest.xml`, so that clients can find it

FibonacciService/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.marakana.android.fibonacciservice" android:versionCode="1"
        android:versionName="1.0">
        <uses-sdk android:minSdkVersion="8" />
        <application android:icon="@drawable/icon" android:label="@string/app_name">
                <service android:name=".FibonacciService">
                        <intent-filter>
                                <action android:name="com.marakana.android. ↩
                                    fibonaccicommon.IFibonacciService" /> <!-- ❶ -->
                        </intent-filter>
                </service>
        </application>
</manifest>
```

❶ The name of this action is arbitrary, but it is a common convention to use the fully-qualified name of our AIDL-derived interface.

3.9 FibonacciClient - Using AIDL-defined Binder-based Services

- We start by creating a new Android project, which will server as the client of the AIDL Service we previously implemented
  - Project Name: `FibonacciClient`
  - Build Target: Android 2.2 (API 8)
  - Package Name: `com.marakana.android.fibonacciclient`
  - Application name: Fibonacci Client
  - Create activity: `FibonacciActivity`
    - We'll repurpose most of this activity's code from `FibonacciNative`
  - Min SDK Version: 8
- We need to link this project to the `FibonacciCommon` in order to be able to access the common APIs: project properties → `Android` → `Library` → `Add...` → `FibonacciCommon`
  - As the result, `FibonacciClient/default.properties` now has `android.library.reference.1=../FibonacciC` and `FibonacciClient/.classpath` and `FibonacciClient/.project` also link to `FibonacciCommon`
  - As an alternative, we could've avoided creating `FibonacciCommon` in the first place
    - `FibonacciService` and `FibonacciClient` could have each had a copy of: `IFibonacciService.aidl`, `FibonacciRequest.aidl`, `FibonacciResponse.aidl`, `FibonacciResult.java`, and `FibonacciResponse.j`
    - But we don't like duplicating source code (even though the binaries do get duplicated at runtime)
- Our client will make use of the string resources and layout definition from `FibonacciNative` application

FibonacciClient/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <string name="hello">Get Your Fibonacci Here!</string>
        <string name="app_name">Fibonacci Client</string>
        <string name="input_hint">Enter N</string>
        <string name="input_error">Numbers only!</string>
        <string name="button_text">Get Fib Result</string>
```

```
        <string name="progress_text">Calculating...</string>
        <string name="fib_error">Failed to get Fibonacci result</string>
        <string name="type_fib_jr">fibJR</string>
        <string name="type_fib_ji">fibJI</string>
        <string name="type_fib_nr">fibNR</string>
        <string name="type_fib_ni">fibNI</string>
</resources>
```

FibonacciClient/res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TextView android:text="@string/hello" android:layout_height="wrap_content"
                android:layout_width="fill_parent" android:textSize="25sp"  ↩
                    android:gravity="center"/>
        <EditText android:layout_height="wrap_content"
                android:layout_width="match_parent" android:id="@+id/input"
                android:hint="@string/input_hint" android:inputType="number"
                android:gravity="right" />
        <RadioGroup android:orientation="horizontal"
                android:layout_width="match_parent" android:id="@+id/type"
                android:layout_height="wrap_content">
                <RadioButton android:layout_height="wrap_content"
                        android:checked="true" android:id="@+id/type_fib_jr"  ↩
                            android:text="@string/type_fib_jr"
                        android:layout_width="match_parent" android:layout_weight="1" ↩
                             />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_ji" android:text="@string/ ↩
                            type_fib_ji"
                        android:layout_width="match_parent" android:layout_weight="1" ↩
                             />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_nr" android:text="@string/ ↩
                            type_fib_nr"
                        android:layout_width="match_parent" android:layout_weight="1" ↩
                             />
                <RadioButton android:layout_height="wrap_content"
                        android:id="@+id/type_fib_ni" android:text="@string/ ↩
                            type_fib_ni"
                        android:layout_width="match_parent" android:layout_weight="1" ↩
                             />
        </RadioGroup>
        <Button android:text="@string/button_text" android:id="@+id/button"
                android:layout_width="match_parent" android:layout_height=" ↩
                    wrap_content" />
        <TextView android:id="@+id/output" android:layout_width="match_parent"
                android:layout_height="match_parent" android:textSize="20sp"  ↩
                    android:gravity="center|top"/>
</LinearLayout>
```

- We are now ready to implement our client

  FibonacciClient/src/com/marakana/android/fibonacciclient/FibonacciActivity.java

```
package com.marakana.android.fibonacciclient;

import android.app.Activity;
import android.app.ProgressDialog;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.text.TextUtils;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;
import android.widget.Toast;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;
import com.marakana.android.fibonaccicommon.IFibonacciService;

public class FibonacciActivity extends Activity implements OnClickListener,  ↩
    ServiceConnection {

    private static final String TAG = "FibonacciActivity";

    private EditText input; // our input n

    private Button button; // trigger for fibonacci calcualtion

    private RadioGroup type; // fibonacci implementation type

    private TextView output; // destination for fibonacci result

    private IFibonacciService service; // reference to or service

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        super.setContentView(R.layout.main);
        // connect to our UI elements
        this.input = (EditText)super.findViewById(R.id.input);
        this.button = (Button)super.findViewById(R.id.button);
        this.type = (RadioGroup)super.findViewById(R.id.type);
        this.output = (TextView)super.findViewById(R.id.output);
        // request button click call-backs via onClick(View) method
        this.button.setOnClickListener(this);
        // the button will be enabled once we connect to the service
        this.button.setEnabled(false);
    }

    @Override
```

```
    protected void onResume() {
        Log.d(TAG, "onResume()'ed");
        super.onResume();
        // Bind to our FibonacciService service, by looking it up by its name
        // and passing ourselves as the ServiceConnection object
        // We'll get the actual IFibonacciService via a callback to
        // onServiceConnected() below
        if (!super.bindService(new Intent(IFibonacciService.class.getName()), this,
                BIND_AUTO_CREATE)) {
            Log.w(TAG, "Failed to bind to service");
        }
    }

    @Override
    protected void onPause() {
        Log.d(TAG, "onPause()'ed");
        super.onPause();
        // No need to keep the service bound (and alive) any longer than
        // necessary
        super.unbindService(this);
    }

    public void onServiceConnected(ComponentName name, IBinder service) {
        Log.d(TAG, "onServiceConnected()'ed to " + name);
        // finally we can get to our IFibonacciService
        this.service = IFibonacciService.Stub.asInterface(service);
        // enable the button, because the IFibonacciService is initialized
        this.button.setEnabled(true);
    }

    public void onServiceDisconnected(ComponentName name) {
        Log.d(TAG, "onServiceDisconnected()'ed to " + name);
        // our IFibonacciService service is no longer connected
        this.service = null;
        // disabled the button, since we cannot use IFibonacciService
        this.button.setEnabled(false);
    }

    // handle button clicks
    public void onClick(View view) {
        // parse n from input (or report errors)
        final long n;
        String s = this.input.getText().toString();
        if (TextUtils.isEmpty(s)) {
            return;
        }
        try {
            n = Long.parseLong(s);
        } catch (NumberFormatException e) {
            this.input.setError(super.getText(R.string.input_error));
            return;
        }

        // build the request object
        final FibonacciRequest request;
```

```
        switch (FibonacciActivity.this.type.getCheckedRadioButtonId()) {
            case R.id.type_fib_jr:
                request = new FibonacciRequest(n, FibonacciRequest. ↩
                    RECURSIVE_JAVA_TYPE);
                break;
            case R.id.type_fib_ji:
                request = new FibonacciRequest(n, FibonacciRequest. ↩
                    ITERATIVE_JAVA_TYPE);
                break;
            case R.id.type_fib_nr:
                request = new FibonacciRequest(n, FibonacciRequest. ↩
                    RECURSIVE_NATIVE_TYPE);
                break;
            case R.id.type_fib_ni:
                request = new FibonacciRequest(n, FibonacciRequest. ↩
                    ITERATIVE_NATIVE_TYPE);
                break;
            default:
                return;
        }

        // showing the user that the calculation is in progress
        final ProgressDialog dialog = ProgressDialog.show(this, "", super
                .getText(R.string.progress_text), true);
        // since the calculation can take a long time, we do it in a separate
        // thread to avoid blocking the UI
        new AsyncTask<Void, Void, String>() {
            @Override
            protected String doInBackground(Void... params) {
                // this method runs in a background thread
                try {
                    FibonacciResponse response = FibonacciActivity.this.service.fib( ↩
                        request);
                    // generate the result
                    return String.format("fibonacci(%d)=%d\nin %d ms", n, response. ↩
                        getResult(),
                            response.getTimeInMillis());
                } catch (RemoteException e) {
                    Log.wtf(TAG, "Failed to communicate with the service", e);
                    return null;
                }
            }

            @Override
            protected void onPostExecute(String result) {
                // get rid of the dialog
                dialog.dismiss();
                if (result == null) {
                    // handle error
                    Toast.makeText(FibonacciActivity.this, R.string.fib_error, Toast. ↩
                        LENGTH_SHORT)
                            .show();
                } else {
                    // show the result to the user
                    FibonacciActivity.this.output.setText(result);
                }
```

```
            }
        }.execute(); // run our AsyncTask
    }
}
```

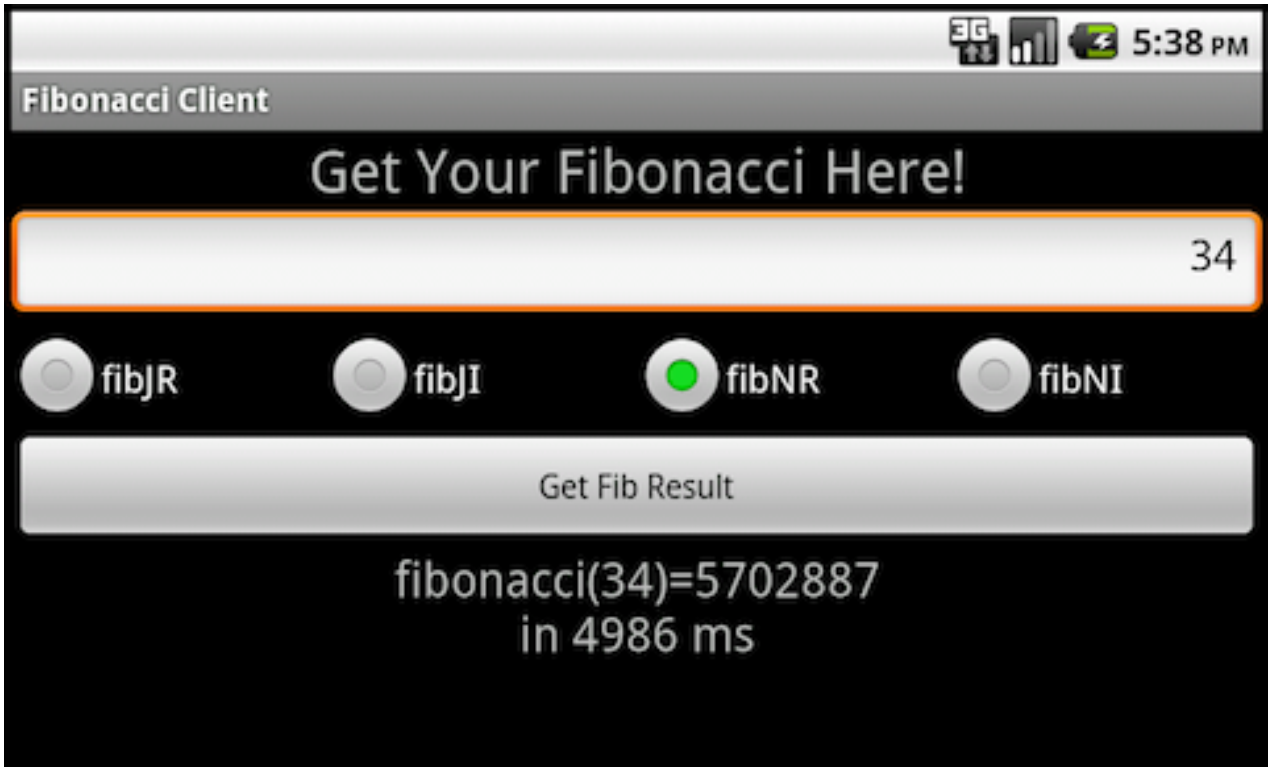
- Our activity should already be registered in our `AndroidManifest.xml` file

  FibonacciClient/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        android:versionCode="1"
        android:versionName="1.0" package="com.marakana.android.fibonacciclient">
        <uses-sdk android:minSdkVersion="8" />
        <uses-permission android:name="com.marakana.android.fibonacci7.GET_FIBONACCI" ↩
            />

        <application android:icon="@drawable/icon" android:label="@string/app_name">
                <activity android:name="com.marakana.android.fibonacciclient. ↩
                    FibonacciActivity" android:label="@string/app_name">
                        <intent-filter>
                                <action android:name="android.intent.action.MAIN" />
                                <category android:name="android.intent.category. ↩
                                    LAUNCHER" />
                        </intent-filter>
                </activity>
        </application>
</manifest>
```

- And the result should look like



3.10 Lab: Binder-based Service with AIDL

Create an AIDL-described `ILogService` that provides the following functionality:

```
package com.marakana.android.logservice;
public interface ILogService {
        public void log(LogMessage logMessage);
}
```

where `LogMessage` is defined as follows:

```
package com.marakana.android.logservice;
public class LogMessage ... {
        ...
        public LogMessage(int priority, String tag, String message) {
                ...
        }
        ...
}
```

Create a simple Android client that allows the user to submit a `LogMessage` request to the remote `ILogService` running in a separate process.

Tip
Your implementation can simply use `android.util.Log.println(int priority, String tag, String msg)` to do the logging.

Chapter 4

Marakana Android Security

4.1 Overview

- Android Security Architecture
- Android Application Signing
- Understanding Android User IDs
- File Access Permissions on Android
- Using Permissions in Android Applications
- Android Permission Enforcement
  - Permissions enforced by the Linux kernel
  - Permissions enforced by the Application Framework Services
- Custom Android Permissions
  - Declaring Custom Permissions
  - Requiring Custom Permissions
  - Enforcing Permissions Dynamically
- ContentProvider URI Permissions
- Developer DOs and DON'Ts
  - Public vs. Private Components
  - Intent Broadcast Permissions
  - Pending Intents
- Encryption on Android
  - Application Data Encryption: OpenSSL, JCE/BouncyCastle
  - Whole Disk Encryption via dm-crypt

- Rooting Android Devices
  - Controlling access to `/system/bin/sh`
  - Understanding Android Exploits
    - UDEV exploit (exploid)
    - ADB setuid exhaustion attack (rageagainstthecage)
    - Buffer-overrun of vold (softbreak/gingerbreak)
    - Webkit exploits
  - To root or not to root
  - Malware rootkits
- Address Space Layout Randomization (ASLR) on Android
- Tap-Jacking on Android
- Device Administration (management) on Android
- Building Anti-malware Applications for Android
- Other Security Concerns
  - Bootloaders
  - Security of GMail accounts affecting security of Android devices
  - Social-engineering vectors of attack
  - Encryption of communication
  - Firewall
  - SE-Linux
  - Recovery mode

4.2 Android Security Architecture

- Android is privilege-separated operating system
  - Each app runs with a separate process with a distinct system identity (user/group ID), as do parts of the system
  - OS (Linux) provides app isolation (sandboxing)
- By default, no app can do anything to adversely affect other apps, the system, or the user
  - E.g. reading/writing user data, modifying other apps'/system' files and settings, accessing network, keeping the device awake, etc.
- Android provides fine-grained permission system that restricts what apps can do if they want to play outside the sandbox
  - Apps statically declare permissions they need (use)
  - The Android system prompts the user for consent at app installation-time (and on update if changed)
  - No support for dynamic (run-time) granting of permissions (complicates user experience)
- Apps can explicitly share resources/data (usually via ContentProviders), and even "logic" (via AIDL)

- Linux kernel is the sole mechanism of application (i.e. process) sandboxing
  - Dalvik VM is not a security boundary
  - Native code (via NDK) is subject to the same restrictions (i.e. no extra privileges)
- All apps are created equal
  - Sandboxed in the same way
  - Same level of security from each other

4.3 Application Signing

- All apps (`.apk` files) *must be* digitally signed prior to installation on a device with a certificate whose private key is kept confidential by the developer of the application
- Android uses the certificate as a means of:
  - Identifying the author of an application
    - Used to ensure the authenticity of future application updates
  - Establishing trust relationships between applications
    - Applications signed with the same certificate can share signature-level permissions, runtime process, user-id, and data
- The signing process is based on public-key cryptography, as defined by Java's JAR specification
  - The private/signing key is kept confidential by the developer of the application
  - The certificate (including the public key) is embedded in the application itself (`META-INF/CERT.RSA`), along with the signature generated with the private key (`META-INF/CERT.SF`)
    - The certificate can be self-signed - **no trusted 3rd party (i.e. certification authority) is required**
    - The certificate's validity period should be 25 years or longer
      - The certificate validity is only verified during the installation/update process (not at startup) - so apps with expired certificates would continue to function normally, but could not be updated
  - Key/certificate management, app signing, and key verification can be accomplished using a number of tools:
    - Java's `keytool` - used to create/manage keys (in keystores)

```
$ keytool -genkey -v -keystore marakana.keystore -alias android -keyalg RSA - ↩
    keysize 2048 -validity 10000 \
  -dname "CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San  ↩
      Francisco, ST=California, C=US"
```

    - Java's `jarsigner` - used to sign/verify `.apk` files (signing is done in-place, replacing any existing signatures)

```
$ jarsigner -keystore marakana.keystore MyApp.apk android
$ jarsigner -verify MyApp.apk
jar verified.
```

    - Android's ADT Export Wizard for Eclipse - offers a wizard UI over the whole signing process (wraps `keytool`/`jarsigner` functionality)
  - During the development, Android allows the use of the debug key (`~/.android/debug.keystore`), but applications *must be* signed using a real key prior to publishing
- Since signatures are applied to the individual files, not the `.apk` itself, developers often use Android's `zipalign` tool to optimize the final `.apk` package

4.4 User IDs

- Each app (package) is assigned an arbitrary but distinct OS user ID at installation time
  - Typically something like `app_XX` (e.g. `app_79`)
- Does not change during app's life on a device
- All app resources are owned by its UID
- Each app process runs under its own UID
- Apps that are signed with the same certificate can share data, user ID, as well as run in a single process
  - They just need to specify the same `sharedUserId` and `process`

AndroidManifest.xml

```
<manifest package="com.marakana.android.myapp"
          android:sharedUserId="myapp"
          android:sharedUserLabel="@string/myapp_uid" ... >
  <application android:process="myapp" ... >
        ...
  </application>
</manifest>
```

4.5 File Access

- Files created by apps are owned by apps' distinct user/group ID and are not world-accessible
- Exceptions
  - `/mnt/sdcard` is FAT32, so free-for-all (though it requires `android.permission.WRITE_TO_EXTERNAL_STORAGE` permission)
  - Apps can create files, preferences, database with `MODE_WORLD_READABLE` and/or `MODE_WORLD_WRITABLE`, which affect world access, but not file ownership

4.6 Using Permissions

- By default, apps cannot do anything outside their sandbox
- To access restricted features of the system, apps require to use permissions, in `AndroidManifest.xml` with explicit `<uses-permission>` elements
- Subject to one-time user-approval, permissions are granted at install time with no support for run-time per-use user approval
- Attempts to access restricted resources without holding the appropriate permission
  - Fail with `SecurityException` (explicit failures)
  - Ignored but logged by the system (implicit failures)

- For example, an app that wishes to track user by GPS and report location via network/SMS would require three permissions: access (fine) location, access the internet, and send SMS
- Its `AndroidManifest.xml` would look like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com. ↩
    marakana.android.trackapp">
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.SEND_SMS" />
  ...
</manifest>
```

4.7 Permission Enforcement

There are a number of trigger points for security/permission checks:

4.7.1 Kernel / File-system Permission Enforcement

- Any kind of system call (file/device I/O, network requests, etc.) is usually enforced by the kernel via group IDs
- Through the use of Android's paranoid network security feature, as well as file system permissions, access to various system resources is restricted to users belonging to pre-determined groups (like `inet`, `camera`, `log`, `sdcard_rw`, etc) defined in `system/core/include/private/android_filesystem_config.h`
- Application's logical permissions (i.e. ones defined via `<uses-permission>` in `AndroidManifest.xml`) are mapped to system groups at the install time

  frameworks/base/data/etc/platform.xml (Android source code)

```
<permissions>
  ...
  <permission name="android.permission.INTERNET" >
    <group gid="inet" />
  </permission>

  <permission name="android.permission.CAMERA" >
    <group gid="camera" />
  </permission>

  <permission name="android.permission.READ_LOGS" >
    <group gid="log" />
  </permission>

  <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
    <group gid="sdcard_rw" />
  </permission>
  ...
</permissions>
```

4.7.2 Application Framework Permission Enforcement

- Modifying system settings
- Accessing/using system services
- Starting an activity
- Sending/receiving broadcasts (enforce receivers as well as senders)
- Reading from or writing to content providers
- Binding to or starting/stopping a service
- List of built-in permissions:
  http://developer.android.com/reference/android/Manifest.permission.html

4.8 Declaring Custom Permissions

Before we can enforce our own permissions, we have to declare them using one or more `<permission>` in

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com. ↩
    marakana.android.myapp" >
  <permission
    android:name="com.example.app.DO_X"
    android:label="@string/do_x_label"
    android:description="@string/do_x_desc"
    android:permissionGroup="android.permission-group.PERSONAL_INFO"
    android:protectionLevel="dangerous" />
  ...
</manifest>
```

4.8.1 Permission components

- `name` - arbitrary, but has to be unique (name-spaced)
- `protectionLevel` - specifies if/how will Android inform the user when another app uses this permission
  - `normal` (0) - A low-risk permission that is automatically granted (by default), though users have an option to review these before installing (users often just ignore)
  - `dangerous` (1) - A higher-risk permission that requires explicit user approval at install time (preferred)
  - `signature` (2) - A permission that will be granted automatically if the requesting app shares the signature of the declaring app (preferred for application "suites")
  - `signatureOrSystem` (3) - A permission that will be granted only to packages in `/system/app/` (i.e. burned to ROM) or that are signed with the same certificates.
    * This is useful in special-cases where different vendors supply apps to be built into system image, and those apps need to work together
- `label` - a short description of the functionality protected by the permission (ignored when `protectionLevel=signature*`)

- `description` - a longer description (warning) of what can go wrong if the permission is abused (ignored when `protectionLevel=signature*`)
- `permissionGroup` - helps organize (group) permissions by some pre-determined categories (e.g. `....COST_MONEY`, `....PERSONAL_INFO`, `....SYSTEM_TOOLS`, etc.)
  - See http://d.android.com/reference/android/Manifest.permission_group.html for the existing Android categories

---

Note
We can list permissions currently defined an an Android device/emulator: `$ adb shell pm list permissions -s`

4.9 Requiring Permissions

- In AndroidManifest.xml via `andriod:permission` attribute
  - On `<activity>`, controls who can `Context.startActivity()` and `startActivityForResult()`
  - On `<service>`, controls who can `Context.startService()`, `stopService()`, and `bindService()`
  - On `<provider>`, `android:readPermission` controls who can use `ContentResolver.query()`; `android:writePerm` controls who can use `ContentResolver.insert()`, `ContentResolver.update()`, and `ContentResolver.delete()`
  - On `<receiver>`, controls who can send broadcasts to the receiver
    - Receivers can also be registered programmatically, so the sender's permission requirement can be specified via `Context.registerReceiver(BroadcastReceiver receiver, IntentFilter filter, String broadcastPermission, Handler scheduler)` method
    - Senders can also programmatically require that the receivers hold the appropriate permission via `Context.sendBroadcast(Int intent, String receiverPermission)`
- For example

```
<manifest ...>
  ...
  <application ...>
    ...
    <activity android:name=".GetPasswordActivity"
      android:permission="com.marakana.android.permission.GET_PASSWORD_FROM_USER"  ↩
          ... >
      ...
    </activity>
    <service android:name=".UserAuthenticatorService"
      android:permission="com.marakana.android.permission.AUTHENTICATE_USER" ... >
      ...
    </service>
    <provider android:name=".EnterpriseDataProvider"
      android:readPermission="com.marakana.android.permission.READ_ENTERPRISE_DATA"
      android:writePermission="com.marakana.android.permission.WRITE_ENTERPRISE_DATA" ↩
          ... >
      ...
    </provider>
    <receiver android:name=".UserAuthStatusReceiver"
      android:permission="com.marakana.android.permission.SEND_USER_AUTH_STATUS">
      ...
```

```
        </receiver>
  </application>
</manifest>
```

4.10 Enforcing Permissions Dynamically

- Permissions can also be enforced programmatically
  - `android.content.Context.checkCallingPermission(String permission)`
  - `android.content.Context.checkPermission(String permission, int pid, int uid)`
  - `android.content.pm.PackageManager.checkPermission(String permName, String pkgName)`
- All these return `PackageManager.PERMISSION_GRANTED` or `PackageManager.PERMISSION_DENIED`
- This is how many of the application framework services enforce their permissions

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

```
package com.android.server;
...
public class VibratorService extends IVibratorService.Stub {
  ...
  public void vibrate(long milliseconds, IBinder token) {
    if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
      != PackageManager.PERMISSION_GRANTED) {
      throw new SecurityException("Requires VIBRATE permission");
    }
    ...
  }
  ...
}
```

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

```
package com.android.server;
...
public class LocationManagerService extends ILocationManager.Stub implements Runnable ↩
     {
  ...
  private static final String ACCESS_FINE_LOCATION =
      android.Manifest.permission.ACCESS_FINE_LOCATION;
  private static final String ACCESS_COARSE_LOCATION =
      android.Manifest.permission.ACCESS_COARSE_LOCATION;
  ...
  private void checkPermissionsSafe(String provider) {
    if ((LocationManager.GPS_PROVIDER.equals(provider)
            || LocationManager.PASSIVE_PROVIDER.equals(provider))
        && (mContext.checkCallingOrSelfPermission(ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
        throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION permission");
    }
    if (LocationManager.NETWORK_PROVIDER.equals(provider)
        && (mContext.checkCallingOrSelfPermission(ACCESS_FINE_LOCATION)
```

```
            != PackageManager.PERMISSION_GRANTED)
        && (mContext.checkCallingOrSelfPermission(ACCESS_COARSE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
        throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION ↩
                    permission");
    }
  }
  ...
  private Location _getLastKnownLocationLocked(String provider) {
    checkPermissionsSafe(provider);
    ...
  }
  ...
  public Location getLastKnownLocation(String provider) {
    ...
    _getLastKnownLocationLocked(provider);
    ...
  }
}
```

- Not very common in application code
  - Can be used in bound services to differentiate access to specific methods (e.g. "administrative" operations)

4.11 `ContentProvider` URI Permissions

- Simple `ContentProvider` read/write permissions on are not always flexible enough
  - E.g. an image viewer app wants to get access to an email attachment for viewing – it would be an overkill to give it read-permission over all of email (assuming that email is exposed via a content provider)
  - E.g. a contact picker (selector) activity wants to select a contact
- Use per-URI permissions instead
  - When starting another Activity, caller sets `Intent.FLAG_GRANT_READ_URI_PERMISSION` or `Intent.FLAG_GRANT_WRITE`
  - This grants the receiving activity permission to access the specific URI regardless of whether it holds a permission over the ContentProvider managing the data behind the URI
- ContentProviders need to add explicit support for URI permissions via `android:grantUriPermission`

4.12 Public vs. Private Components

- Android is based on Inter-component communication (via Intents), but the components can span applications
- Components can be public or private
  - Private by default
  - Public if intent-filters are defined
  - Can be explicitly controlled via a simple boolean `android:exported="true|false"` attribute of individual components in `AndroidManifest.xml`

- Components may be unknowingly be leaked to other applications (assuming no permissions are used)
- Unless absolutely designed for external use, explicitly set the `android:exported="false"`

```
<manifest ...>
  ...
    <application ...>
          ...
      <activity android:name=".GetPasswordActivity" android:exported="false" ... >
        <intent-filter>
          ...
        </intent-filter>
      </activity>
  </application>
</manifest>
```

- Or use permissions and don't trust component inputs!

4.13 Intent Broadcast Permissions

- Broadcast senders can specify which permission the `BroadcastReceiver`-s must hold to access the intent
- If they don't, the broadcast intent is leaked to all applications on the system
- Always specify read permission via `Context.sendBroadcast(Intent i, String receiverPermission)` unless we use an explicit destination

4.14 Pending Intents

- PendingIntent allows delayed triggering of our intent in another application
  - E.g. Notification
  - E.g. Alarms
- The other application can fill-in unspecified values of our intent, which may influence destination and/or integrity of our intent's data
- Best to only use PendingIntents as triggers for private components (i.e. explicitly specify your component's class in the Intent)

4.15 Lab

In the provided Fibonacci example:

1. Restrict access to the `com.marakana.android.fibonacciservice.FibonacciService` to applications that hold *USE_FIBONACCI_SERVICE* custom permission
   a. Start by creating a custom permission (make sure to name-space it)
   b. Then require the permission on the service

c. Test that a client without the required permission cannot bind to the service
   i. Look for an exception stack trace in `adb logcat` when you launch the client
d. Have the client use the required permission
e. Test again - the client should now be able to bind to the service as before

2. Restrict access to "slow" operations of `com.marakana.android.fibonaccicommon.IFibonacciService` (e.g. recursive operations when `n > 10`) to applications that hold *USE_SLOW_FIBONACCI_SERVICE* permission

a. Create another custom permission (make sure to name-space it)
b. Dynamically enforce your permission

> **Tip**
> For you to do this, you'll need access to a `android.content.Context` object inside the provided `com.marakana.android.fibonacciservice.IFibonacciServiceImpl`. Conveniently enough, `com.marakana.android.fibonacciservice.FibonacciService` extends `android.app.Service`, which in turn implements `android.content.Context`.

c. Test that a client without the required permission cannot execute "slow" operations
d. Have the client use the required permission
e. Test again - the client should now be able to use the service as before

4.16 Encryption

4.16.1 Data encryption

- Privacy and integrity of data can be achieved using encryption
- Data being transported off device is usually encrypted via TLS/SSL, which Android supports
  - At the native level via OpenSSL
  - In Java using Java Cryptography Extension (JCE), which on Android is implicitly provided via BouncyCastle (http://www.bouncycastle.c
)
- On-device data encryption is usually also done via JCE:

```
package com.marakana.android.securenote;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
```

```
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.SecretKeySpec;

import android.util.Base64;

public class CryptUtil {
    private static final String ENCRYPTION_ALGORITHM = "AES";

    private static final int KEY_SIZE = 256;

    private static final String RANDOM_ALGORITHM = "SHA1PRNG";

    private static final String CHARSET = "UTF-8";

    public static Cipher getCipher(int mode, byte[] secret) throws  ↩
        NoSuchAlgorithmException,
            NoSuchPaddingException, InvalidKeyException, UnsupportedEncodingException ↩
                {
        // generate an encryption/decryption key from random data seeded with
        // our secret (i.e. password)
        SecureRandom secureRandom = SecureRandom.getInstance(RANDOM_ALGORITHM);
        secureRandom.setSeed(secret);
        KeyGenerator keyGenerator = KeyGenerator.getInstance(ENCRYPTION_ALGORITHM);
        keyGenerator.init(KEY_SIZE, secureRandom);
        Key key = new SecretKeySpec(keyGenerator.generateKey().getEncoded(),  ↩
            ENCRYPTION_ALGORITHM);
        // get a cipher based on the specified encryption algorithm
        Cipher cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        // tell the cipher if it will be used for encryption or decryption
        // (i.e. cipher mode) and give it our key
        cipher.init(mode, key);
        return cipher;
    }

    public static byte[] encrypt(byte[] input, byte[] secret) throws  ↩
        InvalidKeyException,
            IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException,
            NoSuchPaddingException, UnsupportedEncodingException {
        return getCipher(Cipher.ENCRYPT_MODE, secret).doFinal(input);
    }

    public static byte[] decrypt(byte[] input, byte[] secret) throws  ↩
        InvalidKeyException,
            IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException,
            NoSuchPaddingException, UnsupportedEncodingException {
        return getCipher(Cipher.DECRYPT_MODE, secret).doFinal(input);
    }

    public static String encrypt(String input, String secret) throws  ↩
        InvalidKeyException,
            IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException,
            NoSuchPaddingException, UnsupportedEncodingException {
        return Base64.encodeToString(encrypt(input.getBytes(CHARSET), secret.getBytes ↩
            (CHARSET)),
```

```
                Base64.DEFAULT);
    }

    public static String decrypt(String input, String secret) throws  ↩
        InvalidKeyException,
            IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException,
            NoSuchPaddingException, UnsupportedEncodingException {
        return new String(decrypt(Base64.decode(input.getBytes(), Base64.DEFAULT),  ↩
            secret
                .getBytes(CHARSET)), CHARSET);
    }

    public static OutputStream encrypt(OutputStream out, byte[] secret) throws  ↩
        InvalidKeyException,
            NoSuchAlgorithmException, NoSuchPaddingException,  ↩
                UnsupportedEncodingException {
        return new CipherOutputStream(out, getCipher(Cipher.ENCRYPT_MODE, secret));
    }

    public static InputStream decrypt(InputStream in, byte[] secret) throws  ↩
        InvalidKeyException,
            NoSuchAlgorithmException, NoSuchPaddingException,  ↩
                UnsupportedEncodingException {
        return new CipherInputStream(in, CryptUtil.getCipher(Cipher.DECRYPT_MODE,  ↩
            secret));
    }
}
```

Note
In May 2011, Google was caught with their pants down. They passed authentication tokens from their Android client applications to their backend services, including contacts, calendars, and photos (picassa) over a plain-text (i.e. unencrypted) channel. This enabled potential attackers to get access and modify private content of users whose auth tokens were captured.
See http://money.cnn.com/2011/05/18/technology/android_security/?section=money_latest

4.16.2 Whole Disk Encryption

- Android 3.0 (Honeycomb) release introduced a new feature, Settings → Location & Security → Encryption → Encrypt tablet, which enables transparent encryption of the `/data` partition using Linux kernel's dm-crypt functionality (http://www.saout.de/misc/dm-crypt/)
- dm-crypt, which presents itself as a block-device, wraps another block device such as eMMC and similar flash devices (but not raw flash chips, so no yaffs2) and offers on-the-fly encryption/decryption of the underlying data
  - Note that at the moment encryption is done "in software", rather than by an optimized set of hardware instructors on the SoC
- To avoid issues with GPL, Android does not use dm-crypt's `cryptsetup` command and `libdevmapper` shared library, but rather moves that functionality to the volume daemon (`vold`), which directly `ioctl`-calls on the dm-crypt's kernel device
- Additionally, Android's `init` process had to be extended to support password-entry at boot

  - With encryption enabled, `init` gets the password and then restarts into the normal boot process with `/data` properly initialized as a real filesystem
  - See http://source.android.com/tech/encryption/android_crypto_implementation.html for details
- The rest of the Android OS as well as all the apps are unaware of any encryption of the underlying `/data` partition
- At present, the Android uses 128 AES with CBC and ESSIV:SHA256 for `/data`, and 128 bit AES for the master key
- On an unencrypted tablet:
  - The `/data` partition is mounted as a memory block device:

```
$ adb shell mount |grep /data
/dev/block/mmcblk0p8 /data ext4 rw,nosuid,nodev,noatime,barrier=1,data=ordered 0 0
```

  - Writing a 1GB file takes on average 103.919 seconds (10,333,296 bytes/sec)

```
cd /data/local/tmp/
time dd if=/dev/zero of=out bs=4096 count=262144
262144+0 records in
262144+0 records out
1073741824 bytes transferred in 104.006 secs (10,323,845 bytes/sec)
    1m44.02s real     0m0.45s user     0m8.42s system
rm out
time dd if=/dev/zero of=out bs=4096 count=262144
...
rm out
time dd if=/dev/zero of=out bs=4096 count=262144
...
```

  - Reading a 1GB file takes on average 45.99 seconds (23,348,197 byes/sec)

```
time dd if=out of=/dev/null bs=4096 count=262144
262144+0 records in
262144+0 records out
1073741824 bytes transferred in 45.692 secs (23499558 bytes/sec)
    0m45.70s real     0m0.54s user     0m8.84s system
time dd if=out of=/dev/null bs=4096 count=262144
...
time dd if=out of=/dev/null bs=4096 count=262144
...
rm out
```

- After encryption
  - The `/data` partition is mounted as a dm-crypt device-mapper target, which wraps the original memory block device:

```
$ adb shell mount |grep /data
/dev/block/dm-0 /data ext4 rw,nosuid,nodev,noatime,barrier=1,data=ordered 0 0
```

  - Writing a 1GB file takes on average 107.288 seconds (10,014,686 bytes/sec), a mare 3.2% degradation in performance, mostly because writing to NAND is slow
  - Reading a 1GB file takes on average of 70.616 seconds (15,209,002 byes/sec), a significant **54% degradation** in performance
    * Note, that at some point dm-crypt could be optimized to take advantage of any special encryption facilities offered in the hardware (on the SoC)

- Android's whole-disk encryption is still vulnerable to various attacks:
  - "Evil maid attack"
    - Only the `/data` is encrypted, so we don't know if we can trust the bootloader and `/system` not to contain any "keyloggers"
    - Everything along the boot path would have to be encrypted, which it is not at the moment
  - Cold-boot attack (http://citp.princeton.edu/memory/)
    - Since dm-crypt stores the encryption keys in RAM, it would theoretically be possible to reboot the device with something like `msramdmp` (McGrew Security RAM Dumper) or `ram2usb` to dump the contents of RAM (containing the encryption key) to a USB drive
      - The device has to be running in order for this to work
      - Even if the host device itself does not support booting from an alternative device (or USB), the RAM could be cooled (so that it retains its state), and then transferred to a device that would support alternative boot methods
      - This is a problem for almost all disk-encryption "solutions" in popular OSs, like Windows, Mac OS X, and Linux
    - To protect against this, we would need encryption key to be stored somewhere other than RAM, like the CPU (debug) registers, which may be hardware-dependent (e.g. AES-NI on new Intel chips works well), requires changes to the Linux kernel (see `TRESOR` patch), and is not supported by dm-crypt/Android at the moment

Caution
Breaks during 3.0 to 3.1 OS upgrade. A 3.0-encrypted device had to be master-reset (i.e. all data on `/data` had to be wiped) on upgrading to 3.1.

Caution
While Honeycomb's whole-disk encryption based on dm-crypt is clearly a step in the right direction, it is far from being a NIST FIPS 140-2-certified solution, which requires two-factor encryption and is mandated for most of DOD applications.

Note
Apple's iOS 4 256-bit hardware encryption was cracked in May 2011 by ElcomSoft through a "simple" brute-force attack in as little as 30 mins using CPU and GPUs of modern host machines. See http://www.geek.com/articles/chips/apples-ios-4-hardware-encryption-has-been-cracked-20110525/ for more info.
Also, according to Nguyen from Symantec, iOS encryption key is stored on the device, but itself is not encrypted by the user's master key. This means that if a potential attackers successfully jailbreak the device, they would be able to access the data without knowing the passcode.

4.17 Rooting an Android device

To obtain "root" on an Android device usually involves a number of steps:

- Exploit a vulnerability of the system to give us root once (see below)
- Remount the `/system` partition read-write:

```
$ mount -o remount,rw -t yaffs2 /dev/block/mtdblock3 /system
```

- Create a setuid version of `/system/bin/sh`

```
$ cat /system/bin/sh > /system/bin/su
$ chmod 4755 /system/bin/su
```

- Remount the `/system` partition read-only

```
$ mount -o remount,ro -t yaffs2 /dev/block/mtdblock4 /system
```

- Use `/system/bin/su` to become root at any time afterwards

4.17.1 Controlling access to `/system/bin/su` with *Superuser*

- In the real world, most people would not leave `su` wide open to anyone (or any application) - for obvious security implications
- Instead they use something like *Superuser* (http://code.google.com/p/superuser/)
  - Superuser is a combination of a custom `/system/bin/su` binary, as well as a `Superuser.apk` (application).
  - When a 3rd-party application wants super-user access, it runs `/system/bin/su` (possibly with parameters of what it wishes to execute)
    i. The `su` binary first checks whether the calling process is white-listed
       A. First, it checks in the SQLite database owned by the Superuser application: `/data/data/com.koushikdutta.superus`
          * Its `whitelist` table stores three values: the calling process UID, process name (usually just the package name), and a flag (-1=not allowed, 1=temporary allowed, 10000=always allowed)
       B. If the calling process is already not-approved `su` exits
       C. If the calling process is not already approved,
          I. `su` launches a simple activity dialog of the Superuser application by passing the calling process` UID and PID as extra intent parameters: `am start -a android.intent.action.MAIN -n com.koushikdutta.sup --ei uid %d --ei pid %d > /dev/null`
          II. Now `com.koushikdutta.superuser` application starts and `SuperuserRequestActivity` prompts the user to make their selection
          III. The user's selection is saved into the database and `SuperuserRequestActivity` terminates
          IV. `su` now checks the database again, and if the process is not approved, it exits
       D. `su` then `setuid(uid)`-s and `setgid(gid)`-s the calling process
       E. `su` finally executes `/system/bin/sh` with the same parameters that the original `su` was invoked with
    ii. The 3rd party application can now use `sh` to pass any commands that it wishes to run as `root`

4.17.2 Some Of The Past Android Exploits (Getting root the first time)

4.17.2.1 UDEV exploit (CVE-2009-1185)

- On a standard Linux OS, `udev` enables dynamic management of devices - specifically ones that can be hot-plugged while the system is running (like USB)
  - When a new device is detected, Linux kernel passes a message (containing executable code) to the `udev` daemon, which runs as root and acts on this event
  - Prior to version 1.4.1, `udev` did not verify that the message came from the kernel, which made it possible for a rouge application to fake a device and have `udev` execute arbitrary code

  - Newer kernels sent authenticated messages, but `udev` needs to be smart to verify them

- On Android, `udev` functionality is rolled int Android's `init` process (i.e. it is not as a stand-alone executable), which still runs as root

  - This "exploid2" roughly works as follows:
    i. The user `copies` to the device (e.g. `adb push exploid2 /data/local/tmp/exploid2`)
    ii. The user then runs the "exploid2" process (e.g. adb shell `/data/local/tmp/exploid2`)
    iii. On the first run, the "exploid2" copies itself to `/sqlite_stmt_journals/exploid2`
    iv. The "exploid2" then sends a `NETLINK_KOBJECT_UEVENT` message (via a local unix socket) to `init` (i.e., `udev` code within `init`) to tell it to run a copy of itself next time a device is plugged in (basically it presents itself as `FIRMWARE` update for this device)
    v. The user then "hot-plugs" a device by clicking Settings → Wireless → Airplane, WiFi, etc. or plugs in a USB device (if USB host port is available)
    vi. The "exploid2" runs again, this time as `root` (as part of `init`) and it then
        A. Remounts the `/system` in read-write mode
        B. Copies itself to `/system/bin/rootshell` and sets its permission as `04711` (i.e. setuid-bit enabled)
    vii. If the user now wants root, the user simply runs `/system/bin/rootshell`, which then
        A. Switches to `root` via a simple `setuid(0); setgid(0);`
        B. Executes `/system/bin/sh` (now as root) with the parameters passed to `/system/bin/rootshell`

4.17.2.2 ADB setuid exhaustion attack (CVE-2010-EASY)

- Android Debug Bridge Daemon (`adbd`) starts as root, but `setuid` to the `shell` when forking itself to execute remote requests (i.e. to run `/system/bin/shell`)
- In this case, a program called `rageagainstthecage` tries to exploit a race condition such that it can preempt `adbd`-s call to `setuid`
- The program "fork-bombs" `adbd` by creating client requests to it (which causes it to fork) until the system reaches the maximum number of processes (typically around 2-5K)
- At this point, `rageagainstthecage` tries to fill the last slot with its connection to `adbd` while it is still running as `root` before `adbd` has a chance to `setuid()` itself
- The problem is that `adbd` does not check whether its call to `setuid()` succeeded, which leaves `rageagainstthecage` running with `root` access

4.17.2.3 Buffer Overrun `vold`

- On Android, `vold` (volume daemon running as root) is used for operations such as SD-Card mounting/unmounting, as well as encryption of `/data` partition
- Here, an application called Softbreak/Gingerbreak is first uploaded to the device (e.g. to `/data/local/tmp/softbreak`)
- When executed (e.g. via `adb shell`) it tries to exploit an out of bounds array access in `vold` and thus inject code to be executed by `root`
- Because `vold` is configurable by the OEMs (and its memory state changes), this attack is not guaranteed to work every time - in fact, it often causes `vold` to segfault
- A malicious application on the device can exploit the same vulnerability to gain root

4.17.2.4 WebKit exploit

- Initially discovered on iOS, which also uses WebKit
- Based on buffer-overruns
- Enables the browser (or any app using a WebKit via Android's `WebView` or directly via `libwebcore`) to execute arbitrary code
- A proof of concept creates a remote shell
- But, not a root exploit, because the application is sandboxed
  - Still, access to bookmarks, SSL sessions, stored passwords, etc.
- Patched in 2.2

4.17.3 To Root or Not To Root?

- Dangers on already rooted devices
- A malicious app can gain root access and inject a loadable kernel module into the kernel
- Very hard to detect
- Can open network-channels to leak information from the device

4.17.4 Malware Rootkits

- If we can root our own phone, so can malicious applications (e.g. trojans)
- But they have to be installed first
  - Easy, when they are repackaged versions of legit apps, so they look and feel "official"
  - Users are often confused into installing applications from Market that are not authentic, since they have no easy way to verify
  - Google has ability to both pull apps from Market as well as remotely uninstall them from users' devices (via C2DM) but this process is reactive - not proactive
- OEMs/Carriers are often too slow to patch the devices out in the field - so users remain vulnerable to these root exploits

4.18 Address Space Layout Randomization (ASLR) on Android

- Research by Hristo Bojinov and Dan Boneh from Stanford University and Rich Cannings and Iliyan Malchev from Google
  - http://bojinov.org/professional/wisec2011-mobileaslr-paper.pdf
- When employed, ASLR randomizes base addresses of shared libraries, base executables, and process stack and heap
- ASLR's goal is to make certain types of control-hijacking attacks (e.g. buffer/stack-overflow to return to libc) more difficult to execute reliably as the executable code location is unknown (or at least hard to guess)

  - Designed to help against remote exploits from network attacks and network-facing services
    * A website with malicious content
    * Any remote network service that exploits a vulnerability of a local [network] protocol parser (e.g. DHCP, HTTP, NTP, or even app "protocols" like iCal, vCard, etc)
    * A rogue access point
    * A rogue SMS packet
    * A malicious audio/video file that targets a flow in a codec decoder
  - Does not help against local code that has access to the actual memory locations of binary code (even if randomized)
- But, on Android, pre-linking, limited processing power, and restrictive update processes make ASLR hard to use
  - The Android OS prelinks shared libraries to speed up the boot process (by ~ 5%)
    * Prelinking (hard-coding memory addresses in library code) at OS build time
    * Prelinked libraries cannot be relocated (i.e randomized) in process memory - their location is well-known to potential attackers
  - Android's `/system` partition is mounted read-only (for security reasons), but this prevents post-build modifications of the system image (e.g. randomization of library offsets in binary code)
- ASLR on Android
  - Randomize all executable code (including libraries, base executables, and the linker) via *retouching*
    * Random offsets are applied by modifying existing binaries (i.e. "every" device looks different to potential attackers) at system upgrade time (i.e. in the recovery mode) when the `/system` is read-write
    * Requires "undo" capability, in order to roll-back to virgin state (prior to randomization), to support diff-based OTA updates, which depend on known system images
    * Minimal additional storage requirements (2%) to store "undo" info at the end of executables
    * Requires no kernel changes
  - Prevent brute-forcing
    * Given the millions of deployed devices, a successful exploit on one device can potentially work on a large enough subset of other devices
    * For example, with a 8-bit randomization offset, for any one device, there are 1 in 256 devices just like it
    * Use cloud-based analysis of crash reports to reliably detect attempts to bypass ASLR by guessing random offset used on each device
    * Attack on a single device is hard to detect, but an attack across a range of devices tends to have a well-known signature
    * Once detected, issue patches, or isolate vulnerable code that lead to the attack in the first place
  - Non pre-linked libraries already use standard Linux ASLR PaX approach: by randomizing the mmap base
  - Android kernel already performs stack randomization for each process
  - Android's use of `dlmalloc` already offers some overflow (randomization) protection for heap allocated chunks

4.19 Tap-Jacking on Android

- Similar to "click-jacking" in web browsers
- Example scenario

a. A malicious application starts a "sensitive" activity from another application (e.g. system settings)
b. The malicious application then overlays a customized notification dialog on top of the legitimate activity (say something that looks like a game)
c. User interacts with the custom notification dialog, but user touch events are passed down to the legitimate activity (say to inadvertently enable some "insecure" settings mode: like side-loading)
   – Proof of concept: http://www.youtube.com/watch?v=gCLU7YUXUAY

- Another scenario

a. A malicious application starts a background service
b. User launches a legitimate "secure" application (say a banking app)
c. This service launches a transparent custom notification dialog
d. User enters password/pin/gesture to access the secure application
e. The custom notification dialog captures user taps, records them, and passes them down to the secure app
f. The secure app has no knowledge of the fact that the user taps were recorded

- How to prevent?

  – Set `android:filterTouchesWhenObscured="true"` on views which are deemed "secure" (e.g. `EditText` used for password entry)
    * When set to `true` the view system discards touch events that are received whenever the view's window is obscured by another visible window
    * The view will not receive touches whenever a toast, dialog, or other window appears above the view's window
  – For custom views, consider overriding `View.onFilterTouchEventForSecurity(MotionEvent)` to implement your own security policy
  – See http://developer.android.com/reference/android/view/View.html#Security for more info.
  – Only available as for Android API 9 (Gingerbread)

4.20 Android Device Administration

4.20.1 Device Administration Overview

The Android Device Administration API, introduced in Android 2.2, allows you to create security-aware applications that are useful in enterprise settings, such as:

- Email clients
- Security applications that do remote wipe
- Device management services and applications

You use the Device Administration API to write device admin applications that users install on their devices. The device admin application enforces desired *security policies*. Here's how it works:

- A system administrator writes a device admin application that enforces remote/local device security policies.
- The application is installed on a user's device.

- The system prompts the user to enable the device admin application.
- Once the users enable the device admin application, they are subject to its policies.

When enabled, in addition to enforcing security policies, the admin application can:

- Prompt the user to set a new password
- Lock the device immediately
- Perform a factory reset on the device, wiping the user data (if it has permission)

4.20.2 Device Administration Overview (cont.)

If a device contains multiple enabled admin applications, the strictest policy is enforced.

If users do not enable the device admin app, it remains on the device, but in an inactive state.

- Users will not be subject to its policies, but the application may disable some or all of its functionality.

If a user fails to comply with the policies (for example, if a user sets a password that violates the guidelines), it is up to the application to decide how to handle this.

- For example, the application may prompt the user to set a new password or disable some or all of its functionality.

To uninstall an existing device admin application, users need to first deactivate the application as a device administrator.

- Upon deactivation, the application may disable some or all of its functionality, delete its data, and/or perform a factory reset (if it has permission).

4.20.3 Security Policies

An admin application may enforce security policies regarding the device's screen lock PIN/password, including:

- The maximum inactivity time to trigger the screen lock
- The minimum number of PIN/password characters
- The maximum number of failed password attempts
- The minimum number of uppercase letters, lowercase letters, digits, and/or special password characters (Android 3.0)
- The password expiration period (Android 3.0)
- A password history restriction, preventing users from reusing the last *n* unique passwords (Android 3.0)

Additionally, a security policy can require device storage encryption as of Android 3.0.

4.20.4 The Device Administration Classes

The Device Administration API includes the following classes:

`DeviceAdminReceiver`
Base class for implementing a device administration component. This class provides a convenience for interpreting the raw intent actions that are sent by the system. Your Device Administration application must include a `DeviceAdminReceiver` subclass.

`DevicePolicyManager`
A class for managing policies enforced on a device. Most clients of this class must have published a `DeviceAdminReceiver` that the user has currently enabled. The `DevicePolicyManager` manages policies for one or more `DeviceAdminReceiver` instances.

`DeviceAdminInfo`
This class is used to specify metadata for a device administrator component.

4.20.5 Creating the Manifest

The manifest of your admin application must register your `DeviceAdminReceiver` as a `<receiver>`.

The `<receiver>` should set `android:permission="android.permission.BIND_DEVICE_ADMIN"` to ensure that only the system is allowed to interact with the broadcast receiver.

The `<receiver>` must have an `<intent-filter>` child element including one or more of the following `<action>`s, as defined in the `DeviceAdminReceiver` class:

`ACTION_DEVICE_ADMIN_ENABLED`
(Required) This is the primary action that a device administrator must implement to be allowed to manage a device. This is sent to a device administrator when the user enables it for administration.

`ACTION_DEVICE_ADMIN_DISABLE_REQUESTED`
Action sent to a device administrator when the user has requested to disable it, but before this has actually been done.

`ACTION_DEVICE_ADMIN_DISABLED`
Action sent to a device administrator when the user has disabled it.

`ACTION_PASSWORD_CHANGED`
Action sent to a device administrator when the user has changed the password of their device.

`ACTION_PASSWORD_EXPIRING`
Action periodically sent to a device administrator when the device password is expiring.

`ACTION_PASSWORD_FAILED`
Action sent to a device administrator when the user has failed at attempted to enter the password.

`ACTION_PASSWORD_SUCCEEDED`
Action sent to a device administrator when the user has successfully entered their password, after failing one or more times.

```
<receiver android:name="MyDeviceAdminReceiver"
        android:permission="android.permission.BIND_DEVICE_ADMIN">
        <intent-filter>
                <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
                <action android:name="android.app.action. ↩
                    ACTION_DEVICE_ADMIN_DISABLE_REQUESTED" />
                <action android:name="android.app.action.ACTION_DEVICE_ADMIN_DISABLED" ↩
                     />
        </intent-filter>
        <!-- ... -->
</receiver>
```

4.20.6 Creating the Manifest (cont.)

Your `<receiver>` element must also include a `<meta-data>` child element specifying an XML resource declaring the policies used by your admin application.

- The `android:name` attribute must be `android.app.device_admin`.
- The `android:resource` must reference an XML resource in your application.
- For example:

```
<meta-data android:name="android.app.device_admin"
           android:resource="@xml/device_admin_sample" />
```

An example XML resource requesting all policies would be:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
        <uses-policies>
                <limit-password />
                <watch-login />
                <reset-password />
                <force-lock />
                <wipe-data />
                <expire-password />
                <encrypted-storage />
        </uses-policies>
</device-admin>
```

Your application needs to list only those policies it actually uses.

4.20.7 The `DeviceAdminReceiver` Class

The `DeviceAdminReceiver` class defines a set of methods that you can override to handle the device administration events broadcast by the system:

`void onEnabled(Context context, Intent intent)`
: Called after the administrator is first enabled, as a result of receiving `ACTION_DEVICE_ADMIN_ENABLED`. At this point you can use `DevicePolicyManager` to set your desired policies.

`CharSequence onDisableRequested(Context context, Intent intent)`
Called when the user has asked to disable the administrator, as a result of receiving `ACTION_DEVICE_ADMIN_DISABLE_REQUEST`. You may return a warning message to display to the user before being disabled, or `null` for no message.

`void onDisabled(Context context, Intent intent)`
Called prior to the administrator being disabled, as a result of receiving `ACTION_DEVICE_ADMIN_DISABLED`. Upon return, you can no longer use the protected parts of the `DevicePolicyManager` API.

`void onPasswordChanged(Context context, Intent intent)`
Called after the user has changed their password, as a result of receiving `ACTION_PASSWORD_CHANGED`.

`void onPasswordExpiring(Context context, Intent intent)`
Called periodically when the password is about to expire or has expired, as a result of receiving `ACTION_PASSWORD_EXPIRING`. (API 11)

`void onPasswordFailed(Context context, Intent intent)`
Called after the user has failed at entering their current password, as a result of receiving `ACTION_PASSWORD_FAILED`.

`void onPasswordSucceeded(Context context, Intent intent)`
Called after the user has succeeded at entering their current password, as a result of receiving `ACTION_PASSWORD_SUCCEEDED`.

4.20.8 Testing Whether the Admin Application is Enabled

You can query the `DevicePolicyManager` to test if your admin application is enabled:

```
DevicePolicyManager devicePolicyManager
        = (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName deviceAdminComponentName
        = new ComponentName(this, MyDeviceAdminReceiver.class);
boolean isActive = devicePolicyManager.isAdminActive(deviceAdminComponentName);
```

You could then enable or disable features of your application depending on whether it is an active device administrator.

4.20.9 Enabling the Application

Your application must explicitly request the user to enable it for device administration. To do so:

1. Create an implicit Intent with the `DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN` action:

   ```
   Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
   ```

2. Add an extra identifying your `DeviceAdminReceiver` component:

   ```
   ComponentName deviceAdminComponentName
           = new ComponentName(this, MyDeviceAdminReceiver.class);
   intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, deviceAdminComponentName);
   ```

3. Optionally, provide an explanation as to why the user should activate the admin application:

   ```
   intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION, "Your boss told you to ↩
       do this");
   ```

4. Use the Intent with `startActivityForResult()` to display the activation dialog:

```
startActivityForResult(intent, ACTIVATION_REQUEST);
```

5. You can test for successful activation in your Activity's `onActivityResult()` method:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case ACTIVATION_REQUEST:
            if (resultCode == Activity.RESULT_OK) {
                Log.i("DeviceAdminSample", "Administration enabled!");
            } else {
                Log.i("DeviceAdminSample", "Administration enable FAILED!");
            }
            return;
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

4.20.10 Setting Password Quality Policies

`DevicePolicyManager` includes APIs for setting and enforcing the device screen lock password policy.

- The `setPasswordQuality()` lets you set basic password requirements for your admin application, using these constants:

  `PASSWORD_QUALITY_ALPHABETIC`
  The user must enter a password containing at least alphabetic (or other symbol) characters.

  `PASSWORD_QUALITY_ALPHANUMERIC`
  The user must enter a password containing at least both numeric and alphabetic (or other symbol) characters.

  `PASSWORD_QUALITY_NUMERIC`
  The user must enter a password containing at least numeric characters.

  `PASSWORD_QUALITY_SOMETHING`
  The policy requires some kind of password, but doesn't care what it is.

  `PASSWORD_QUALITY_UNSPECIFIED`
  The policy has no requirements for the password.

  `PASSWORD_QUALITY_COMPLEX`
  (API 11) The user must have entered a password containing at least a letter, a numerical digit and a special symbol.

- Once you have set the password quality, you may also specify a minimum length (except with `PASSWORD_QUALITY_SOMETHING` and `PASSWORD_QUALITY_UNSPECIFIED`) using the `setPasswordMinimumLength()` method.

- For example:

```
devicePolicyManager.setPasswordQuality(deviceAdminComponentName,  ↩
    PASSWORD_QUALITY_ALPHANUMERIC);
devicePolicyManager.setPasswordMinimumLength(deviceAdminComponentName, 6);
```

Your application's policy metadata resource must request the `<limit-password />` policy to control password quality; otherwise these methods throw a security exception.

4.20.11 Setting Password Quality Policies, API 11

Beginning with Android 3.0, the `DevicePolicyManager` class includes methods that give you greater control over the contents of the password. Here are the methods for fine-tuning a password's contents:

- `setPasswordMinimumLetters()`
- `setPasswordMinimumLowerCase()`
- `setPasswordMinimumUpperCase()`
- `setPasswordMinimumNonLetter()`
- `setPasswordMinimumNumeric()`
- `setPasswordMinimumSymbols()`

You can also set the password expiration timeout, and prevent users from reusing the last *n* unique passwords:

- `setPasswordExpirationTimeout()`
- `setPasswordHistoryLength()`

Additionally, Android 3.0 introduced support for a policy requiring the user to encrypt the device, which you can set with:

- `setStorageEncryption()`

Your application's policy metadata resource must request the `<limit-password />` policy to control password quality; otherwise these methods throw a security exception.

Similarly, it must request the `<expire-password />` and `<encrypted-storage />` policies to control those features without throwing a security exception.

4.20.12 Setting the Device Password

You can test if the current device password meets the quality requirements by calling `DevicePolicyManager.isActivePasswordSu`
which returns a boolean result.

If necessary, you can start an activity prompting the user to set a password as follows:

```
Intent intent = new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
startActivity(intent);
```

Your application can also perform a password reset on the device using `DevicePolicyManager.resetPassword()`. This can be useful if your application is designed to support remote administration, with a new password being provided from a central administration system.

Your application's policy metadata resource must request the `<reset-password />` policy to reset the password; otherwise `resetPassword()` throws a security exception.

4.20.13 Locking and Wiping the Device

Your application can lock the device programmatically using `DevicePolicyManager.lockNow()`.

You can wipe the user data of the device, performing a factory reset, using `DevicePolicyManager.wipeData()`.

Additionally, you can set the maximum number of allowed failed password attempts before the device is wiped automatically by calling `DevicePolicyManager.setMaximumFailedPasswordsForWipe()`

Your application's policy metadata resource must request the `<wipe-data />` policy to wipe the data either explicitly or set the maximum failed passwords for wipe; otherwise a security exception is thrown. `setMaximumFailedPasswordsForWipe()` also requires the `<watch-login />` policy.

The `lockNow()` method requires your application to request the `<force-lock />` policy to avoid throwing a security exception.

4.21 Anti-malware

- Usually, anti-malware apps are reactive
  - Use `android.content.pm.PackageManager.getInstalledPackages(int)` for the initial scan of apps/packages against a known black-list
  - Listen for `android.intent.action.PACKAGE_ADDED` broadcasts and verify new apps
  - Once a malicious app is found, offer the user a chance to delete it

```
Uri packageURI = Uri.parse("package:com.malicous.app");
Intent uninstallIntent = new Intent(Intent.ACTION_DELETE, packageURI);
startActivity(uninstallIntent);
```

- Some of the popular apps include
  - Norton Mobile Security
  - Lookout Mobile Security

4.22 Other Security Concerns

- Push-based installation of apps from Market (based on the Google account)
  - If the Google account associated with the device is compromised, malicious applications could be pushed directly to affected owner's devices
- Social-engineering vectors of attack
  - Lack of the developer-user trust model
  - Reactive vs preventative security of Market-installed apps
- Firewall

  - Android uses all-or-nothing access to networking
    * Requested via the INTERNET permission, mapped to `inet` group, enforced via `ANDROID_PARANOID_NETWORK` kernel extension
    * `iptables` is available, but not exposed to the user
    * No easy way to setup a firewall policy controlling/limiting access to network resources
  - WhisperMonitor is a 3rd party custom ROM that tries to address this shortcoming, providing a full-fledged application firewall manageable by the end user
- Encryption of communication
  - Whisper Systems' RedPhone application uses ZRTP-encrypted SMS messages to establish calls over a VOIP connection (hidden behind an alternative dialer)
    * ZRTP was designed by PGP inventor Phillip Zimmerman
- Enable SE-Linux (Security Enhanced Linux) access rights controls at the Android kernel level
  - Experimental work by Hitachi (http://www.youtube.com/watch?v=e2yWDvcWu6I)
  - Challenges
    * Extended file attributes (`xattr`) not available on yaffs2, yet required by SE-Linux. Use ext3 or ext4 instead.
    * Only two domains can be used: processes started directly off `init` and those run from `app_process`, but we should be able to set up different policies for individual apps.
    * Default SE-Linux policy files are way too big. Optimize for Android.
  - http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5342408
- Rogue applications signed using platform keys (targeting custom ROMs): jSMSHider
  - http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/
- App obfuscation
  - Proguard
- Recovery mode
  - State of the device while in the recovery mode
- Controlling access to private content with a privacy manager
  - North Carolina State University's TISSA (Taming Information-Stealing Smartphone Applications) privacy manager controls access to user's private data (Location, Phone Identity, Contacts, and Call Log)
    * On an app-by-app basis, users decide how their data is shared: Trusted (unlimited), Bogus (false), Anonymized (filtered), or Empty (pretend that there is no data)
    * http://mobile.engadget.com/2011/04/19/ncsu-teases-tissa-for-android-a-security-manager-that-keeps-per/

Chapter 5

Building Android From Source

5.1 Why Build Android From Source?

- To understand how Android works
- Build custom ROMs
  - Enable custom services - beyond what's possible with APKs along
  - Support custom hardware

5.2 Setting up the Build Environment

- Before we can download and build Android from source, we need to setup our build environment
- Requirements:
  - Linux (specifically Ubuntu 10.04 or later) and Mac OS X are officially supported
    - Windows is explicitly not supported
  - Approximately 12GB of disk space (2.6GB for sources and 10GB for a complete build)
  - A case-sensitive file-system
    - Most Unix/Linux-based file systems are already case-sensitive
    - Mac OS Extended (HFS+) file system is case-insensitive by default - we need to create "case sensitive, journaled" volume (e.g. via "Disk Utility")
  - Python 2.4 — 2.7
  - Java Development Kit (official Sun/Oracle release is recommended)
    - JDK 5 for ⇐ Froyo
    - JDK 6 for >= Gingerbread
  - Git 1.5.4 or newer
  - Build tools and libraries: flex, bison, gcc, make, zlib, libc-dev, ncurses, 32-bit dev libs, etc.
  - See http://source.android.com/source/initializing.html for the easy copy+paste installation instructions for the required binaries

5.3 Downloading the Source Tree

- First we need to get `repo`, a simple tool that makes it easier to work with Git in the context of Android

```
$ mkdir ~/bin
$ export PATH=~/bin:$PATH
$ curl http://android.git.kernel.org/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

  - For more info on `repo`, see http://source.android.com/source/version-control.html

- The next step is to set up a local working directory (on a case-sensitive file-system)

```
$ mkdir android-src
$ cd android-src
```

- Next, we `repo init` to update repo itself as well as specify where we are going to download the sources from (including the particular version/branch)

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git -b gingerbread
```

  - Here, `gingerbread` points to a particular branch - i.e. Android 2.3.3
  - For the complete list of available branches (e.g. "donut", "eclair", "froyo", "gingerbread", etc.), see "heads" under http://android.git.kernel.org/?p=platform/manifest.git;a=summary
  - For more info on codenames, tags, and build numbers see http://source.android.com/source/build-numbers.html
  - To understand Android Branches and Releases (i.e. code-lines), see http://source.android.com/source/code-lines.html

---

Note
Official Android repository `android.git.kernel.org` is not the only game in town.
For example, to get Android sources for TI's SOCs, we would:
`repo init -u git://gitorious.org/rowboat/manifest.git -m TI-Android-FroYo-DevKit-V2.2.xml`
for OMAP3 and
`repo init -u git://gitorious.org/rowboat/manifest.git -m TI-Android-FroYo-DSP-DevKit-V2.x`
for DM37x EVM
(the `.xml` files come from TI) to check out Froyo build optimized for their hardware.

- The final step is to pull the actual files from the repository (2.6GB download)

```
$ repo sync
```

- See http://source.android.com/source/downloading.html for more details on getting Android sources

5.4 Android Source Code Structure

- `bionic/` - the home of the Bionic (libc) library
- `bootable/` - the home of Android's bootloader, diskinstaller, and recovery image support
- `build/` - the home of the Android build system

- `cts/` - Android's Compatibility Test Suite
  - See http://source.android.com/compatibility/cts-intro.html for more info.
- `dalvik/` - the home of the Dalvik VM
- `development/` - development tools, configuration files and sample apps
- `device/` - device-specific binaries (like kernel and device drivers) and source
  - Gingerbread tree supports HTC Nexus One and Samsung Nexus S
- `external/` - 3rd party libraries (mostly native, but also Java), which are synced from their own repositories
- `frameworks/` - Android-specific native utilities (e.g. `app_process`, `bootanimation`, etc.), daemons (e.g. `installd`, `servicemanager`, `system_server`), and libraries (including JNI wrappers and HAL support), as well as Java APIs (i.e. all of `android.*`) and services (all all Application Framework support)
- `hardware/` - hardware-abstraction-layer (HAL) definitions (`libhardware` and `libhardware_legacy`) and some device-specific implementations (e.g. msm7k's `libaudio.so` and TI OMAP3's `libstagefrighthw.so`) both in source code and binaries
- `libcore/` - Apache Harmony (see `libcore/luni/src/main/java/`) as well as test/support libraries
- `ndk/` - the home of NDK
- `out/` - the location where binaries built by `make` go
- `packages/` - the home of the built-in applications (e.g. Phone, Browser, Gallery, etc), content providers (e.g. Contact Provider, Media Provider, etc.), wall papers (including live wall papers), input methods (e.g. LatinIME), etc.
- `prebuilt/` - pre-built kernels (mostly for QEMU) as well as other binaries (mostly 3rd party development tools)
- `sdk/` - the home of Android SDK tools (`ddms`, `traceview`, `ninepatch`, etc.)
- `system/` - the home of the Android root file system, configuration files, `init` and `init.rc`, as well as some of the native daemons

5.5 Android Build System

- Android's build system is based on *make*
- Android OS is a collection of modules (executables, libraries, applications, etc.) - each with its own unique makefile (`Android.mk`)
- The build system was designed around some specific objectives:
  - Allows building for multiple targets (emulator, various devices) and on multiple hosts (Linux and MacOS X)
  - Uses make non-recursively - helps avoid slow/unpredictable build times
    - The build system still allows makefiles from sub-directories (one level) to be included via `include $(call all-subdir-makefiles)`
  - Allows selective (individual component) builds - to speed up compile-test cycles
  - Enables configuration via environment settings as well as configuration files - makes the build system more flexible
  - Separates built binaries from the source files - enables fast cleans
  - Enables automatic dependency resolution - so no need for explicit dependancies in makefiles
  - Enables hiding of command lines - so the output of the build is not too verbose
  - Enables multiple targets in one directory - allows for modules to be more compact

5.6 Initializing the Build Environment

- Before we can run the actual build, we need to initialize the build environment by sourcing `build/envsetup.sh` into our shell
- Running this command adds special functions to our shell: `help`, `croot`, `m`, `mm`, `mmm`, `cgrep`, `jgrep`, `resgrep`, `godir`, `lunch`, etc. - as these are used later for the actual build
- Finally, this script also includes vendor/device-specific functions as well (which are used to register targets to build)

```
$ source build/envsetup.sh
$ help
Invoke ". build/envsetup.sh" from your shell to add the following functions to your  ↩
    environment:
- croot:   Changes directory to the top of the tree.
- m:       Makes from the top of the tree.
- mm:      Builds all of the modules in the current directory.
- mmm:     Builds all of the modules in the supplied directories.
- cgrep:   Greps on all local C/C++ files.
- jgrep:   Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.
- godir:   Go to the directory containing a file.

Look at the source to view more functions. The complete list is:
add_lunch_combo cgrep check_product check_variant choosecombo chooseproduct  ↩
    choosetype choosevariant cproj croot
findmakefile gdbclient get_abs_build_var get_build_var getbugreports getprebuilt  ↩
    gettop godir help isviewserverstarted
jgrep lunch m mm mmm pid print_lunch_menu printconfig resgrep runhat runtest  ↩
    set_java_home set_sequence_number
set_stuff_for_environment setpaths settitle smoketest startviewserver stopviewserver  ↩
    systemstack tapas tracedmdump
```

5.7 Choosing the Build Target

- Next, we choose our *target* using the `lunch` command (actually, it's one of the functions added by `build/envsetup.sh` to our shell)

```
$ lunch

You're building on Darwin

Lunch menu... pick a combo:
     1. generic-eng
     2. full_passion-userdebug
     3. full_crespo-userdebug
     4. full_crespo4g-userdebug

Which would you like? [generic-eng]

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
```

```
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=darwin
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
```

- We could also run `lunch 1` to choose the first target or we can explicitly specify a particular target as `lunch generic-eng`
- We could also use `choosecombo` command, which internally uses `chooseproduct`, `choosetype`, and, `choosevariant` to prompt us for product, type, and variant separately

```
$ choosecombo
Only device builds are supported for Darwin
     Forcing TARGET_SIMULATOR=false

Press enter:



Build type choices are:
     1. release
     2. debug

Which would you like? [1]

Which product would you like? [generic]


Variant choices are:
     1. user
     2. userdebug
     3. eng
Which would you like? [eng]

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=darwin
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
```

- Either way, the result of `lunch` or `choosecombo` sets up the environmental variables which the build system reads to know what to build

- These environmental variables can also be printed using `printconfig` command

- Targets take the form of *CodeName-BuildType*
  - *CodeName* is mapped to a specific device as follows
    - `passion` - HTC Nexus One
    - `crespo` - Samsung Nexus S
    - `crespo4g` - Samsung Nexus S 4G
    - `generic` - QEMU Emulator

    Note
    Building AOSP on Linux-x86 also supported a `simulator` target, which was a partial build of Android, compiled as a single Linux x86 binary, meant to run as a stand-alone process. It would enable testing Dalvik, OpenCORE/Stagefright, or WebKit under Valgrind (instrumentation/debugging framework). But `simulator` has been dropped as a valid lunch target since it's no longer actively maintained by the AOSP team.

  - *BuildType* specifies the intended use (security restrictions)
    - `user`
      - Intended to be the final release
      - Installs modules tagged as `user`
      - Installs non-APK modules that have no tags specified
      - Installs APKs according to the product definition files (tags are ignored for APK modules)
      - Sets `ro.secure=1`
      - Sets `ro.debuggable=0`
      - `adbd` is disabled by default (i.e. has to be enabled via Settings → Applications → Development → USB Debugging)
    - `userdebug` - the same as `user`, except:
      - Intended for limited debugging
      - Installs modules tagged with `debug`
      - Sets `ro.debuggable=1`
      - `adbd` is enabled by default
    - `eng`
      - Intended for platform-level debugging
      - Installs modules tagged with: `eng`, `debug`, `user`, and/or `development`
      - Installs non-APK modules that have no tags specified
      - Installs APKs according to the product definition files, in addition to tagged APKs
      - Sets `ro.secure=1`
      - Sets `ro.debuggable=0`
      - Sets `ro.kernel.android.checkjni=1`
      - `adbd` is enabled by default

Note
Additional *CodeName-BuildType* combinations may be available based on what `build/envsetup.sh` finds (usually in the `device/` folder)

Note
Building for real hardware usually requires that we get proprietary binaries (user-space HAL). For Nexus phones, we can go to http://code.google.com/android/nexus/drivers.html, which allows us to download scripts, which in turn extract binaries from the connected devices (via `adb pull`) into `vendor/` directory tree structure.

5.8 Compiling Android

- We can build the entire code-base with GNU's `make` command (note that this can take 20-120 mins)

```
$ make -j4
```

  - GNU Make supports `-jN` argument, which allows us to parallelize compilation across multiple hardware threads
    - Typically, `N` should be set to the number of hardware threads available on the host machine plus 2
    - For example, use `make -j10` to build on a quad-core i7 with two hardware threads per core (i.e. hyperthreaded)
    - At some point, disk I/O becomes the bottleneck

- To speed up rebuilds, enable compiler cache (`prebuilt/<host>/ccache/ccache`):

```
$ export USE_CCACHE=1
$ make -j4
```

5.8.1 Makefile targets

- In addition to the default `all` target, Android's `Makefile` (actually `build/core/main.mk`) supports additional targets
- `make sdk` - build the tools that are part of Android SDK (`adb`, `fastboot`, etc.)
- `make snod` - build the system image from the current software binaries
- `make services`
- `make runtime`
- `make droid` - make droid is the normal build
- `make all` - make everything, whether it is included in the product definition or not
- `make clean` - remove all built files (prepare for a new build). Same as rm -rf out/<configuration>/
- `make clobber`
- `make dataclean`
- `make installclean`
- `make modules` - shows a list of submodules that can be built (List of all LOCAL_MODULE definitions)
- `make <local_module>` - make a specific module (note that this is not the same as directory name. It is the LOCAL_MODULE * definition in the Android.mk file)
- `make clean-<local_module>` - clean a specific module

5.9 Examining the Built Images

- Upon successful `generic` (emulator) build, the `make` command will create the following images
  - `out/target/product/generic/ramdisk.img` - the temporary root file system used during the booting process to setup our mount points and support the `init` process
  - `out/target/product/generic/system.img` - this `/system` image with
    * `/system/bin` - system binaries (including various daemons started by `init` via `init.rc`)
    * `/system/xbin` - debugging/profiling binaries
    * `/system/usr` - mostly configuration/mapping files (e.g. keylayouts)
    * `/system//etc` - configuration files (e.g. permissions, library mappings, networking/DHCP, etc.)
    * `/system/lib` - system/framework native libraries (including support for user-space HAL)
    * `/system/framework` - system/framework Dalvik libraries (in dex bytecode format but in `.jar` files)
    * `/system/app` - built-in system apps (including content providers, input providers, etc.)
    * `/system/fonts` - built-in system fonts
  - `out/target/product/generic/userdata.img` - the contents of the `/data` partition (empty)
  - Since the emulator is based on QEMU, it uses its own built-in `/boot` partition with a pre-compiled kernel on it
- Upon successful `full_crespo-userdebug` build (i.e. device-specific), the `make` command will create the following images
  - `out/target/product/crespo/boot.img` - a custom Android "filesystem" consisting of a 2KB header, followed by a gzipped kernel, followed by a ramdisk, followed by an optional second stage loader (not typically used)
    * This image (as well as `recovery.img`) is created using Android's `out/host/<arch>/bin/mkbootimg` command, and produces a file of the following structure:
      http://android.git.kernel.org/?p=platform/system/core.git;a=blob;f=mkbootimg/bootimg.h

```
52 /*
53 ** +-----------------+
54 ** | boot header     | 1 page
55 ** +-----------------+
56 ** | kernel          | n pages
57 ** +-----------------+
58 ** | ramdisk         | m pages
59 ** +-----------------+
60 ** | second stage    | o pages
61 ** +-----------------+
62 **
63 ** n = (kernel_size + page_size - 1) / page_size
64 ** m = (ramdisk_size + page_size - 1) / page_size
65 ** o = (second_size + page_size - 1) / page_size
66 **
67 ** 0. all entities are page_size aligned in flash
68 ** 1. kernel and ramdisk are required (size != 0)
69 ** 2. second is optional (second_size == 0 -> no second)
70 ** 3. load each element (kernel, ramdisk, second) at
71 **    the specified physical address (kernel_addr, etc)
72 ** 4. prepare tags at tag_addr.  kernel_args[] is
73 **    appended to the kernel commandline in the tags.
74 ** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
75 ** 6. if second_size != 0: jump to second_addr
76 **    else: jump to kernel_addr
77 */
```

  - `out/target/product/crespo/ramdisk.img` - see above (included in `boot.img`)
    * For more info on ramdisk, see http://android.git.kernel.org/?p=kernel/common.git;a=blob;f=Documentation/filesystems/-ramfs-rootfs-initramfs.txt
  - `out/target/product/crespo/system.img` - similar to `generic` system.img, but with additional content
    * `/system/modules` - custom kernel modules (i.e. mostly drivers)
    * `/system/media/audio` - custom audio ringtones, notifications, etc.
  - `out/target/product/crespo/userdata.img` - the initial contents of the `/data` partition
  - `out/target/product/crespo/recovery.img` - like `boot.img`, it has its own boot header, kernel, and ramdisk, but it also includes a custom `/sbin/recovery` program used during system recovery
    * While Android system can reboot itself into recovery mode, we can usually also boot into recovery through special key combination - usually by holding down volume up or home key on power-up
  - `out/target/product/crespo/ramdisk-recovery.img` - see above (included in `recovery.img`)
- See http://source.android.com/source/building.html for more info on building Android

5.10 Running Custom Android Build on Emulator

- Run `emulator` right from the source directory (requires `lunch` to be run first)

```
$ source build/envsetup.sh
$ lunch ...
$ make ...
$ emulator
```

- Alternatively, we can run our image in one of our Android Virtual Devices (AVD)
  - Create an AVD - unless we already have one from before
    * We could use "Android SDK and AVD Manager" GUI to create an AVD (based on the same "target" as the image we compiled)
    * Or, we could create an AVD from the command line (assuming we are in the source directory and SDK's `android` is in our `PATH`):

```
$ android create avd --sdcard 16M --target android-10 --name custom-avd --path ↩
    custom-avd
Android 2.3.3 is a basic Android platform.
Do you wish to create a custom hardware profile [no]
Created AVD 'custom-avd' based on Android 2.3.3,
with the following hardware config:
hw.lcd.density=240
vm.heapSize=24
hw.ramSize=256
```

  - We are now ready to run it (assuming we are in the source directory and SDK's `emulator` is in our `PATH`):

```
$ emulator -avd custom-avd -system out/target/product/generic/system.img -ramdisk ↩
    out/target/product/generic/ramdisk.img &
```

- And the result is

5.11 Running Custom Android Build on Real Hardware

- First we reboot our device into fastboot mode (assuming SDK's `adb` is in our `PATH`)

```
$ adb reboot bootloader
```

  - We could also reboot using the special key combination on power-up, but that's often device-specific (as mentioned above)
    - Nexus S and Nexus S 4G: Press and hold Volume Up, then press and hold Power

    * Nexus One: Press and hold the trackball, then press Power
    * G1 and MyTouch 3G: Press and hold Back, then press Power
  - Android requires that every device support "fastboot" protocol - a mechanism for communicating with bootloaders over USB
  - For more info on Fastboot, see `bootable/bootloader/legacy/fastboot_protocol.txt` (relative to the source directory)
- We may need to unlock our bootloader (which may not be permitted on many devices)

```
$ out/host/darwin-x86/bin/fastboot oem unlock
```

Note
Nexus One (passion) does not allow the bootloader to be locked again with `fastboot oem lock`

- We are now ready to flash our device

```
$ out/host/darwin-x86/bin/fastboot flashall -w
```

Caution
Running this command will **wipe** our device! The `-w` switch is used to also wipe the `/data` partition, which is sometimes necessary (on first-flash, or on major updates), but is otherwise not required.

Note
Nexus One (passion) does not support the `-w` switch, so instead we can `fastboot erase cache` and `fastboot erase userdata` before flashing.

- Finally, we reboot the device to run our custom build

5.12 Building the Linux Kernel

- As of 2009, Linux kernel is no longer cloned as part of the standard `repo sync`
  - The kernel's build system is separate from the one Android uses for the rest of the platform
  - Most system integrators get the pre-built kernel from their SoC provider

Note
It is easiest to build the Linux kernel on a Linux OS. While other host OSs can also be used, they are not trivial to setup.

5.12.1 Building Kernel for the Emulator (Goldfish)

1. Start the emulator
2. Get the existing kernel version from `/proc/version` (since `uname` does not exist on Android)

```
$ adb shell cat /proc/version
Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com) (gcc ↩
    version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
```

3. Get the corresponding kernel version (here, we are getting the kernel for goldfish, the emulator)

```
$ git clone git://android.git.kernel.org/kernel/common.git
$ cd common
$ git checkout -t origin/archive/android-gldfish-2.6.29 -b goldfish
```

Note
Alternatively, we could directly clone the goldfish 2.6.29 branch (yes, there is a typo in the branch name)
`$ git clone git://android.git.kernel.org/kernel/common.git -b archive/android-gldfish-2.6.29`

---

4. Specify the target architecture and cross compiler

```
$ export ARCH=arm
$ export CROSS_COMPILE=/path/to/android-src/prebuilt/linux-x86/toolchain/arm-eabi ↩
    -4.4.0/bin/arm-eabi-
```

Caution
Android's own build system uses `ARCH` and `CROSS_COMPILE` env vars, so these need to be cleared before building AOSP.
As an alternative, these can be also passed directly to the make commands: `$ make ... ARCH=... CROSS_COMPILE=...`

---

5. Get the existing kernel configuration file (with all of the Android/emulator specific options) by pulling it from the running emulator:

```
$ adb pull /proc/config.gz .
$ gunzip config.gz
$ mv config .config
```

Note
As an alternative, we could generate the Goldfish `.config` file (even without the emulator) by running:
`$ make goldfish_defconfig`

---

6. You can optionally take a look at the existing configuration options and make changes as desired

```
$ make menuconfig
```

- For example, you could set your *General setup* → *Local version - append to kernel release* to something like `-marakana-example`

7. Now we are ready to compile

```
$ make
```

8. The resulting kernel will be compressed to `arch/arm/boot/zImage`
9. Run the emulator with our new kernel

```
$ emulator -kernel /path/to/common/arch/arm/boot/zImage
```

10. And the result is

Chapter 6

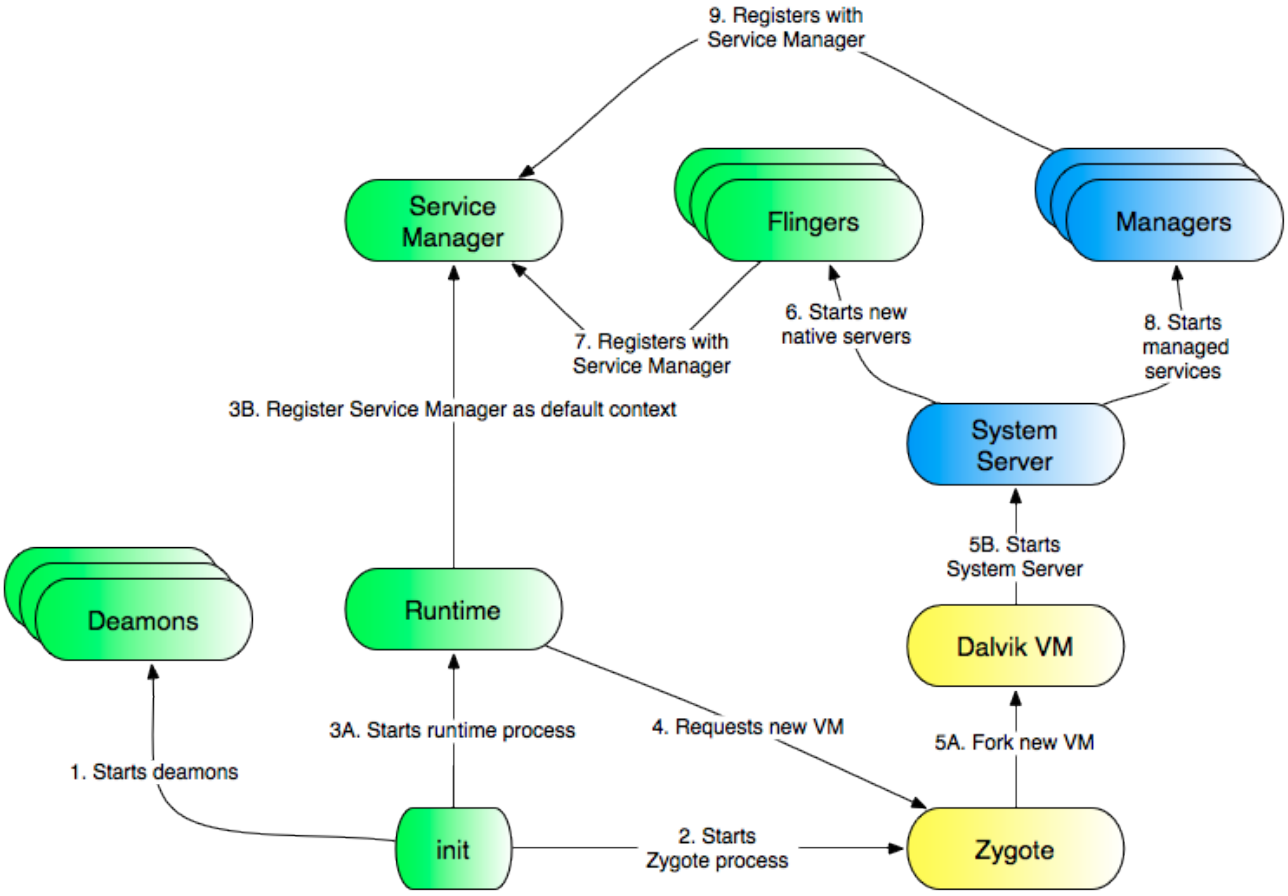
Android Startup

6.1 Bootloading the Kernel

1. On power-up, CPU is uninitialized - wait for stable power
2. Execute Boot ROM (hardwired into CPU)
    a. Locate the first-stage boot loader
    b. Load the first-stage boot loader into internal RAM
    c. Jump to first-stage boot loader's memory location to execute it
3. First-stage boot loader runs
    a. Detect and initialize external RAM
    b. Locate the second-stage boot loader
    c. Load the second-stage boot loader into external RAM
    d. Jump to the second-stage boot loader's memory location to execute it
4. Second-stage boot loader runs
    a. Setup file systems (typically on Flash media)
    b. Optionally setup display, network, additional memory, and other devices
    c. Enable additional CPU features
    d. Enable low-level memory protection
    e. Optionally load security protections (e.g. ROM validation code)
    f. Locate Linux Kernel
    g. Load Linux Kernel into RAM
    h. Place Linux Kernel boot parameters into memory so that kernel knows what to run upon startup
    i. Jump to Linux Kernel memory address to run it
5. Linux Kernel runs
    a. Build a table in RAM describing the layout of the physical memory

b. Initialize and setup input devices
c. Initialize and setup disk (typically MTD) controllers and map available block devices in RAM
d. Initialize Advanced Power Management (APM) support
e. Initialize interrupt handlers: Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT), and Programmable Interrupt Controllers (PIC)
f. Reset the floating-point unit (FPU)
g. Switch from real to protected mode (i.e. enable memory protection)
h. Initialize segmentation registers and a provisional stack
i. Zero uninitialized memory
j. Decompress the kernel image
k. Initialize provisional kernel page tables and enable paging
l. Setup kernel mode stack for process `0`
m. Fill the IDT with null interrupt handlers
n. Initialize the first page frame with system parameters
o. Identify the CPU model
p. Initialize registers with the addresses of the GDT and IDT
q. Initialize and start the kernel
    i. Scheduler
    ii. Memory zones
    iii. Buddy system allocator
    iv. IDT
    v. SoftIRQs
    vi. Date and Time
    vii. Slab allocator
    viii. ...
r. Create process `1` (`/init`) and run it

6.2 Android's `init` Startup



- The granddaddy of all other processes on the system (PID=1)
- Custom initialization script (with its own language)
  - Different than more traditional `/etc/inittab` and SysV-init-levels initialization options
- When started by the kernel, `init` parses and executes commands from two files:
  - `/init.rc` - provides generic initialization instructions (see `system/core/rootdir/init.rc`)
  - `/init.<board-name>.rc` - provides machine-specific initialization instructions - sometimes overriding `/init.rc`
    - `/init.goldfish.rc` for the emulator
    - `/init.trout.rc` for HTC's ADP1
    - `/init.herring.rc` for Samsung's Nexus S (see `device/samsung/crespo/init.herring.rc`)
- The `init` program never exists (if it were, the kernel would panic), so it continues to monitor started services
- The init's language consists of four broad classes of statements (see `system/core/init/readme.txt`)
  - Actions - named sequences of commands queued to be executed on a unique trigger

```
on <trigger>
   <command>
   <command>
   <command>
```

- Triggers - named events that trigger actions
  * `early-init`, `init`, `early-fs`, `fs`, `post-fs`, `early-boot`, `boot` - built-in stages of init
  * `<name>=<value>` - fires when a system property is set to `<name>=<value>`
  * `device-added-<path>` - fires when a device node at `<path>` is added
  * `device-removed-<path>` - fires when a device node at `<path>` is removed
  * `service-exited-<name>` - fires when a service by `<name>` exists
- Commands - a command to be queued and run
  * `chdir <directory>` - change working directory
  * `chmod <octal-mode> <path>` - change file access permissions
  * `chown <owner> <group> <path>` - change file user and group ownership
  * `chroot <directory>` - change process root directory
  * `class_start <serviceclass>` - start all non-running services of the specified class
  * `class_stop <serviceclass>` - stop all running services of the specified class
  * `domainname <name>` - set the domain name.
  * `exec <path> [ <argument> ]*` - fork and execute `<path> <argument> ...` (blocks `init` until exec returns)
  * `export <name> <value>` - set globally visible environment variable `<name> =<value>`
  * `hostname <name>` - set the host name
  * `ifup <interface>` - bring the network interface <interface> online
  * `import <filename>` - parse and process `<filename>` init configuration file (extends the current script)
  * `insmod <path>` - install the kernel module at `<path>`
  * `loglevel <level>` - initialize the logger to `<level>`
  * `mkdir <path> [mode] [owner] [group]` - create a directory at `<path>` and optionally change its default permissions (`755`) and user (`root`) / group (`root`) ownership
  * `mount <type> <device> <dir> [ <mountoption> ]*` - attempt to mount named `<device>` at `<dir>` with the optional `<mountoption>`s
  * `setprop <name> <value>` - set system property `<name>=<value>`
  * `setrlimit <resource> <cur> <max>` - set the rlimit for a `<resource>`
  * `start <service>` - start named `<service>` if it is not already running
  * `stop <service>` - stop name `<service>` if it is currently running
  * `symlink <target> <path>` - symbolically link `<path>` to `<target>`
  * `sysclktz <mins_west_of_gmt>` - set the system clock base (0 for GMT)
  * `trigger <event>` - trigger an event - i.e. call one action from another
  * `write <path> <string> [ <string> ]*` - write arbitrary `<string>`s to file by specified `<path>`
- Services - persistent daemon programs to be launched (optionally) restarted if they exit

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

  - Options - modifiers to services, modifying how/when they are run re/launched
    * `critical` - a device-critical service - devices goes into recovery if the service exits more than 4 times in 4 minutes
    * `disabled` - not started by default - i.e. it has to be started explicitly by name
    * `setenv <name> <value>` - set the environment variable `<name>=<value>` in the service
    * `socket <name> <dgram|stream|seqpacket> <perm> [ <user> [ <group> ] ]` - create a unix domain socket named /dev/socket/<name> and pass its fd to the service
    * `user <username>` - change service's effective user ID to `<username>`
    * `group <groupname> [ <groupname> ]*` - change service's effective group ID to `<groupname>` (additional groups are supplemented via `setgroups()`)
    * `oneshot` - ignore service exist (default is to auto-restart)
    * `class <name>` - set the class name of the service (defaults to `default`) so that all services of a particular class can be started/stopped together
    * `onrestart` - execute a command on restart

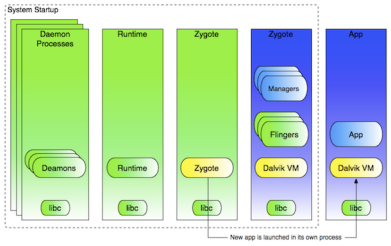
- The default `init.rc` script

  1. Starts `ueventd`
  2. Initializes the system clock and logger
  3. Sets up global environment
  4. Sets up the file system (mount points and symbolic links)
  5. Configures kernel timeouts and scheduler
  6. Configures process groups
  7. Mounts the file systems
  8. Creates a basic directory structure on `/data` and applies permissions
  9. Applies permissions on `/cache`
  10. Applies permissions on certain `/proc` points
  11. Initializes local network (i.e. localhost)
  12. Configures the parameters for the low memory killer Section 1.1.7
  13. Applies permissions for `systemserver` and daemons
  14. Defines TCP buffer sizes for various networks
  15. Configures and (optionally) loads various daemons (i.e. services): `ueventd`, `console`, `adbd`, `servicemanager`, `vold`, `netd`, `debuggerd`, `rild`, `zygote` (which in turn starts `system_server`), `mediaserver`, `bootanimation` (one time), and various Bluetooth daemons (like `dbus-daemon`, `bluetoothd`, etc.), `installd`, `racoon`, `mtpd`, `keystore`

- Nexus S' `init.herring.rc` additionally

  1. Sets up product info
  2. Initializes device-driver-specific info, file system structures, and permissions: battery, wifi, phone, uart_switch, GPS, radio, bluetooth, NFC, lights
  3. Initializes and (re)mounts file systems
  4. Loads additional device-specific daemons

6.3 Zygote Startup



1. Zygote starts from `/init.rc`

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start- ↩
    system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

2. This translates to `frameworks/base/cmds/app_process/app_main.cpp:main()`
3. The command `app_process` then launches `frameworks/base/core/java/com/android/internal/os/ZygoteIni` in a Dalvik VM via `frameworks/base/core/jni/AndroidRuntime.cpp:start()`
4. `ZygoteInit.main()` then
    a. Registers for zygote socket
    b. Pre-loads classes defined in `frameworks/base/preloaded-classes` (1800+)
    c. Pre-loads resources `preloaded_drawables` and `preloaded_color_state_lists` from `frameworks/base/cor`
    d. Runs garbage collector (to clean the memory as much as possible)
    e. Forks itself to start `systemserver`
    f. Starts listening for requests to fork itself for other apps

6.4 System Server Startup

1. When Zygote forks itself to launch the `systemserver` process (in `ZygoteInit.java:startSystemServer()`), it executes `frameworks/base/services/java/com/android/server/SystemServer:java.main()`
2. The `SystemServer:java.main()` method loads `android_servers` JNI lib from `frameworks/base/services/jni` and invokes `init1()` native method
3. Before `init1()` runs, the JNI loader first runs `frameworks/base/services/jni/onload.cpp:JNI_OnLoad()`, which registers native services - to be used as JNI counterparts to Java-based service manager loaded later
4. Now `frameworks/base/services/jni/com_android_server_SystemServer.cpp:init1()` is invoked, which simply wraps a call to `frameworks/base/cmds/system_server/library/system_init.cpp:syste`
5. The `system_init.cpp:system_init()` function
    a. First starts native services (some optionally):

i. `frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp`
ii. `frameworks/base/services/sensorservice/SensorService.cpp`
iii. `frameworks/base/services/audioflinger/AudioFlinger.cpp`
iv. `frameworks/base/media/libmediaplayerservice/MediaPlayerService.cpp`
v. `frameworks/base/camera/libcameraservice/CameraService.cpp`
vi. `frameworks/base/services/audioflinger/AudioPolicyService.cpp`

b. Then goes back to `frameworks/base/services/java/com/android/server/SystemServer.java:init2(` again via `frameworks/base/core/jni/AndroidRuntime.cpp:start()` JNI call

6. The `SystemServer.java:init2()` method then starts Java service managers in a separate thread (`ServerThread`), readies them, and registers each one with `frameworks/base/core/java/android/os/ServiceManager:addService` (which in turn delegates to to ServiceManagerNative.java, which effectively talks to `servicemanager` daemon previously started by `init`)

a. `frameworks/base/services/java/com/android/server/PowerManagerService.java`
b. `frameworks/base/services/java/com/android/server/am/ActivityManagerService.java`
c. `frameworks/base/services/java/com/android/server/TelephonyRegistry.java`
d. `frameworks/base/services/java/com/android/server/PackageManagerService.java`
e. `frameworks/base/services/java/com/android/server/BatteryService.java`
f. `frameworks/base/services/java/com/android/server/VibratorService.java`
g. etc

7. Finally `frameworks/base/services/java/com/android/server/am/ActivityManagerService.java:fini` sets `sys.boot_completed=1` and sends out

a. a broadcast intent with `android.intent.action.PRE_BOOT_COMPLETED` action (to give apps a chance to reach to boot upgrades)
b. an activity intent with `android.intent.category.HOME` category to launch the *Home* (or *Launcher*) application
c. a broadcast intent with `android.intent.action.BOOT_COMPLETED` action, which launches applications subscribed to this intent (while using `android.permission.RECEIVE_BOOT_COMPLETED`)

Chapter 7

Customizing Android

- In this section, we will create:
  - a custom Android device (ROM), we'll call it *Marakana Alpha*
  - and a custom Android SDK add-on, that will enable 3rd party developers to develop for our custom device

7.1 Setting up the Directory Structure

- While we could overlay our device's custom components over the existing AOSP source tree, that makes it harder to deal with future OS upgrades
- Instead, we will create a **self-contained** directory structure to host our device
  1. Create our vendor (e.g. *mararkana*) directory: `mkdir device/marakana/`
  2. Now create our device (e.g. *alpha*) sub-directory: `mkdir device/marakana/alpha`
  3. Also create our SDK addon (e.g. *alpha-sdk_addon*) sub-directory: `mkdir device/marakana/alpha-sdk_addon`
  4. Finally, create a sub-directory for common components, shared between the device and the addon: `mkdir device/marakana/alpha-common`

7.2 Registering our Device with Android's Build System

- Remember that `$ source build/envsetup.sh` registers lunch combos that we can later build
- We now want to add our device to that "lunch" list
  1. Create `vendorsetup.sh` file for our device (the name of this file is fixed):
     device/marakana/alpha/vendorsetup.sh

     ```
     add_lunch_combo full_marakana_alpha-eng
     ```

  2. Re-build the lunch list:

```
$ source build/envsetup.sh
including device/htc/passion/vendorsetup.sh
including device/marakana/alpha/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
```

3. Finally, we can check to see that our device now appear in the lunch menu:

```
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. full_passion-userdebug
     3. full_marakana_alpha-eng
     4. full_crespo-userdebug
     5. full_crespo4g-userdebug

Which would you like? [generic-eng]
```

   – We are **not yet ready** to select it here, because we have not yet provided the necessary makefiles for `full_marakana_alpha`. If we do, we'll get:

```
$ lunch 3
build/core/product_config.mk:203: *** No matches for product " ↩
    full_marakana_alpha".  Stop.

** Don't have a product spec for: 'full_marakana_alpha'
** Do you have the right repo manifest?
```

7.3 Adding the Makefile Plumbing for our Device

- We now need to add basic support for building our device

1. Start by creating a our own `AndroidProducts.mk` file, which simply defines the actual makefiles to be used when building our device:
   device/marakana/alpha/AndroidProducts.mk

```
PRODUCT_MAKEFILES := $(LOCAL_DIR)/full_alpha.mk
```

> **Note**
> The only purpose of `AndroidProducts.mk` file (whose name is fixed) is to set `PRODUCT_MAKEFILES` to a list of product makefiles to expose to the build system. The only external variable it can use is `LOCAL_DIR`, whose value will be automatically set to the directory containing this file.

2. Create the main build-file for our device (we call it `full_crespo` following the example from `device/samsung/crespo/ful`
   device/marakana/alpha/full_alpha.mk

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/languages_small.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/generic.mk)

# Discard inherited values and use our own instead.
PRODUCT_NAME := full_marakana_alpha
PRODUCT_DEVICE := alpha
PRODUCT_MODEL := Full Marakana Alpha Image for Emulator

include $(call all-makefiles-under,$(LOCAL_PATH))
```

- Notice that this file:
  - Includes `build/target/product/languages_small.mk`, which defines a list of languages to be supported by our device
  - Includes `build/target/product/generic.mk`, which defines the rules for building the base Android platform, but itself is not specialized for any particular device
  - Includes all make files in the current directory (see below)
  - Defines our custom `PRODUCT_NAME`, `PRODUCT_DEVICE`, and `PRODUCT_MODEL`, which are arbitrarily chosen for our device

3. Next, we'll import some boiler-plate make/support files from the generic board - since our device will run on the emulator:

```
$ cp build/target/board/generic/BoardConfig.mk device/marakana/alpha/.
$ cp build/target/board/generic/AndroidBoard.mk device/marakana/alpha/.
$ cp build/target/board/generic/tuttle2.kcm device/marakana/alpha/.
$ cp build/target/board/generic/tuttle2.kl device/marakana/alpha/.
```

- While `AndroidBoard.mk`, `tuttle2.kcm`, and `tuttle2.kl` are there to configure key-bindings, the one file of interest (which we'll change later) is `BoardConfig.mk` since it defines our device board's kernel/hardware capabilities:
  device/marakana/alpha/BoardConfig.mk

```
# config.mk
#
# Product-specific compile-time definitions.
#

# The generic product target doesn't have any hardware-specific pieces.
TARGET_NO_BOOTLOADER := true
TARGET_NO_KERNEL := true
TARGET_CPU_ABI := armeabi
HAVE_HTC_AUDIO_DRIVER := true
BOARD_USES_GENERIC_AUDIO := true

# no hardware camera
USE_CAMERA_STUB := true
```

4. We are now ready to try out our "custom" device (true, nothing truly custom yet, except for the `PRODUCT_*` settings):
   a. For good measure, re-register our device:

```
$ source build/envsetup.sh
including device/htc/passion/vendorsetup.sh
including device/marakana/alpha/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
```

5. Now we can do the full `lunch` of our device:

```
lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. full_passion-userdebug
     3. full_marakana_alpha-eng
     4. full_crespo-userdebug
     5. full_crespo4g-userdebug

Which would you like? [generic-eng] full_marakana_alpha-eng

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full_marakana_alpha
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=darwin
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
```

6. We can now compile our device:

```
$ export USE_CCACHE=1
$ make -j10
...
Target system fs image: out/target/product/alpha/obj/PACKAGING/ ↩
    systemimage_intermediates/system.img
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

7. Finally, we can run it

```
$ emulator &
```

8. And we should see :screens/MarakanaAlpha-AboutPhone-v1.png

7.4 Adding a Custom Kernel to our Device

- Our device works perfectly fine with the provided QEMU-based (i.e. emulator-specific) kernel: `prebuilt/android-arm/kernel/k` but we could also configure it to use our custom one, say if we needed to add support for a particular driver or kernel feature

1. Build a custom kernel as desired, as outlined in ???TITLE??? section (or specifically ???TITLE???)
2. Copy the compiled kernel file (`zImage`) to our device's common directory (since we'll use it in our SDK addon as well):

```
$ cp /path/to/kernel/arch/arm/boot/zImage device/marakana/alpha-common/kernel
```

3. Enable our custom kernel in `BoardConfig.mk` (double-negation, nice!):
device/marakana/alpha/BoardConfig.mk

```
...
TARGET_NO_KERNEL := false
...
```

4. Create a `alpha.mk` makefile for our common components, where we'll configure our kernel:
device/marakana/alpha-common/alpha.mk

```
MY_PATH := $(LOCAL_PATH)/../alpha-common

include $(call all-subdir-makefiles)

# Enable our custom kernel
LOCAL_KERNEL := $(MY_PATH)/kernel
PRODUCT_COPY_FILES += $(LOCAL_KERNEL):kernel
```

5. Now, we need to include our common `alpha.mk` file in our device's main makefile (`full_alpha.mk`):
device/marakana/alpha/full_alpha.mk

```
...
include device/marakana/alpha-common/alpha.mk
```

6. Test

```
$ make -j10
...
# (re)start the emulator
$ emulator -kernel out/target/product/alpha/kernel &
# wait for the emulator to start
$ adb shell cat /proc/version
Linux version 2.6.29-marakana-example-gb0d93fb (student@ubuntu) (gcc version  ↩
    4.4.0 (GCC) ) #5 Tue Jul 19 22:24:03 PDT 2011
```

Note
We had to run our emulator with `-kernel`, because it defaults to the prebuilt one.

Note
The path to `adb` was added to our `PATH` when we `$ source build/envsetup.sh` (on a Linux host, it comes from `adb`)

7.5 Adding a Custom Native Library and Executable to our Device

- In this section, we'll create a simple native shared library (like a HAL), which will directly interact with a kernel driver (the Android logger driver at `/dev/log/main`) and allow us to
  - Flush the log buffer

  - Get the max size of the log buffer
  - Get the used size of the log buffer

- We'll also create a simple executable that will use our shared library
- Let's get started:

  1. We start by creating a home for our shared libraries:

     ```
     $ mkdir device/marakana/alpha-common/lib/
     ```

  2. Since our actual libraries will be in their own sub-directories, we need to include them in our build system, via a simple makefile:
     device/marakana/alpha-common/lib/Android.mk

     ```
     include $(call all-subdir-makefiles)
     ```

  3. Now we can create an actual directory for our `libmrknlog` library:

     ```
     $ mkdir device/marakana/alpha-common/lib/libmrknlog
     ```

  4. Now we can create a simple header file (`libmrknlog.h`) for our library:
     device/marakana/alpha-common/lib/libmrknlog/libmrknlog.h

     ```
     #ifndef _MRKNLOG_H_
     #define _MRKNLOG_H_

     #include <cutils/logger.h>
     #include <cutils/log.h>
     #include <string.h>
     #include <fcntl.h>
     #include <errno.h>
     #include <sys/ioctl.h>
     #include <sys/stat.h>
     #include <sys/types.h>

     #ifdef __cplusplus
     extern "C" {
     #endif

     extern int mrkn_flush_log();
     extern int mrkn_get_total_log_size();
     extern int mrkn_get_used_log_size();

     #ifdef __cplusplus
     }  /* End of the 'extern "C"' block */
     #endif
     #endif /* End of the _MRKNLOG_H_ block */
     ```

  5. Next, we implement our shared library (`libmrknlog.c`) - using `ioctl()` to talk to the kernel driver:
     device/marakana/alpha-common/lib/libmrknlog/libmrknlog.c

     ```
     #define LOG_FILE "/dev/log/main"
     #define LOG_TAG "MrknLog"

     #include "libmrknlog.h"
     ```

```
static int ioctl_log(int mode, int request) {
    int logfd = open(LOG_FILE, mode);
    if (logfd < 0) {
        LOGE("Failed to open %s: %s", LOG_FILE, strerror(errno));
        return -1;
    } else {
        int ret = ioctl(logfd, request);
        close(logfd);
        return ret;
    }
}

extern int mrkn_flush_log() {
    return ioctl_log(O_WRONLY, LOGGER_FLUSH_LOG);
}

extern int mrkn_get_total_log_size() {
    return ioctl_log(O_RDONLY, LOGGER_GET_LOG_BUF_SIZE);
}

extern int mrkn_get_used_log_size() {
    return ioctl_log(O_RDONLY, LOGGER_GET_LOG_LEN);
}
```

6. We are now ready for the makefile (`Android.mk`):
device/marakana/alpha-common/lib/libmrknlog/Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := libmrknlog.c
LOCAL_SHARED_LIBRARIES := libcutils libutils libc
LOCAL_MODULE := libmrknlog
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)
```

A note about prelinking
Android's linker allows shared libraries to be pre-linked (they are registered to be loaded at fixed virtual memory addresses), which speeds up their loading and generally allows for faster booting.
In this example, we used `LOCAL_PRELINK_MODULE := false` to indicate that our shared library is *not* to be pre-linked, because we wanted to keep our device modifications self-contained (i.e. within `device/marakana/`).
Alternatively, we could pre-link our library (to be compiled as `libmrknlog.so`) by adding the following line to architecture-specific pre-link mapping file:
`build/core/prelink-linux-arm.map`:

```
...
libmrknlog.so           0x9C900000 # libmrknlog
```

Prelinked libraries are aligned on 1MB boundaries, and are listed in descending address order. Here, `0x9C900000` was chosen because the last shared library listed in `build/core/prelink-linux-arm.map` (`liblept.so`) was addressed at `0x9CA00000`, and `libmrknlog.so` is less than 1MB (0x9CA00000 - 0x9C900000 = 0x100000 = 1048576 = 1024 * 1024 = 1MB).

---

7. Optionally, we can compile our library to test that it builds:

```
$ make -j10 libmrknlog
...
target SharedLib: libmrknlog (out/target/product/generic/obj/SHARED_LIBRARIES/ ↩
    libmrknlog_intermediates/LINKED/libmrknlog.so)
target Non-prelinked: libmrknlog (out/target/product/generic/symbols/system/lib/ ↩
    libmrknlog.so)
target Strip: libmrknlog (out/target/product/generic/obj/lib/libmrknlog.so)
Install: out/target/product/generic/system/lib/libmrknlog.so
```

8. Since our library's `Android.mk` states `LOCAL_MODULE_TAGS := optional`, we need to register it with Alpha's `PRODUCT_PACKAGES`, otherwise it will not be included in the final ROM:
device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += libmrknlog
```

9. Before we can create our executable to test our library, let's create a directory for all binaries:

```
$ mkdir device/marakana/alpha-common/bin
```

10. Like with our libraries, we need to "recursively" include makefiles in sub-directories of `.../bin/`, which is where our executables will live:
device/marakana/alpha-common/bin/Android.mk

```
include $(call all-subdir-makefiles)
```

11. Now we can create a directory for our `mrknlog` executable:

```
$ mkdir device/marakana/alpha-common/bin/mrknlog
```

12. Next we provide the implementation (`mrknlog.c`), which will use our shared library (`libmrknlog`):
device/marakana/alpha-common/bin/mrknlog/mrknlog.c

```
#include <stdio.h>
#include <libmrknlog.h>

int main (int argc, char* argv[]) {
    int usedSize = mrkn_get_used_log_size();
    int totalSize = mrkn_get_total_log_size();
    if (totalSize >= 0 && usedSize >= 0) {
        if (mrkn_flush_log() == 0) {
            printf("Flushed log. Previously it was consuming %d of %d bytes\n",
                usedSize, totalSize);
            return 0;
        } else {
            fprintf(stderr, "Failed to flush log: %s", strerror(errno));
        }
    } else {
        fprintf(stderr, "Failed to get log size: %s", strerror(errno));
    }
    return -1;
}
```

13. As with everything else, we need a makefile (`Android.mk`) to build our executable:
device/marakana/alpha-common/bin/mrknlog/Android.mk

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

include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := mrknlog.c
LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../lib/libmrknlog/
LOCAL_SHARED_LIBRARIES := libmrknlog libc
LOCAL_MODULE := mrknlog
include $(BUILD_EXECUTABLE)
```

Note
Here, we need to tell the compiler where to find our library's header file:
`LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../lib/libmrknlog/`
and tell the linker to link against our library:
`LOCAL_SHARED_LIBRARIES := libmrknlog ...`
Finally, unlike last time when we `include $(BUILD_SHARED_LIBRARY)`, this time we:
`include $(BUILD_EXECUTABLE)`
which will produce `system/bin/mrknlog`

---

14. We can compile our executable to test that it builds:

```
$ make -j10 mrknlog
...
target Executable: mrknlog (out/target/product/generic/obj/EXECUTABLES/ ↩
    mrknlog_intermediates/LINKED/mrknlog)
Install: out/target/product/generic/system/lib/libutils.so
Install: out/target/product/generic/system/lib/libmrknlog.so
target Non-prelinked: mrknlog (out/target/product/generic/symbols/system/bin/ ↩
    mrknlog)
target Strip: mrknlog (out/target/product/generic/obj/EXECUTABLES/ ↩
    mrknlog_intermediates/mrknlog)
Install: out/target/product/generic/system/bin/mrknlog
```

15. Just like with our library, since our executable's `Android.mk` states `LOCAL_MODULE_TAGS := optional`, we need to register it with Alpha's `PRODUCT_PACKAGES`, or otherwise it will not be included in the final ROM:
device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += mrknlog
```

16. Now we can rebuild our entire device:

```
$ make -j10
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

17. Finally, we are ready to test our library/executable:

```
# (re)start the emulator
$ emulator -kernel out/target/product/alpha/kernel &
# wait for the emulator to finish
# check out our library
$ adb shell ls -l /system/lib/libmrknlog.so
```

```
-rw-r--r-- root     root         5328 2011-07-09 21:13 libmrknlog.so
# check out our utility
$ adb shell ls -l /system/bin/mrknlog
-rwxr-xr-x root     shell        5476 2011-07-09 21:13 mrknlog
# check if our utility is doing what it is supposed to
$ adb logcat -g
/dev/log/main: ring buffer is 64Kb (33Kb consumed), max entry is 4096b, max  ↩
    payload is 4076b
$ adb shell /system/bin/mrknlog
Flushed log. Previously it was consuming 34530 of 65536 bytes
$ adb logcat -g
/dev/log/main: ring buffer is 64Kb (0Kb consumed), max entry is 4096b, max  ↩
    payload is 4076b
# check again
$ adb shell /system/bin/mrknlog
Flushed log. Previously it was consuming 0 of 65536 bytes
# good :-)
```

7.6 Using our Native Library via a Custom Daemon

- Having a custom utility `mrknlog` is fine, but what if we wanted to have it run periodically, to flush the log buffers at a regular interval - we'd need a daemon
- As we discussed before, daemons are executables (e.g. `mrknlogd`) generally started by `init` (from `init.rc`) as `service`-es
- Let's get started:

1. As before, we need to create a home for our daemon source:

```
$ mkdir device/marakana/alpha-common/bin/mrknlogd
```

2. Next, we create or daemon source (`mrknlogd.c`), by utilizing our `libmrknlog` library:
device/marakana/alpha-common/bin/mrknlogd/mrknlogd.c

```
#define LOG_TAG "MRKN Log Daemon"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <utils/Log.h>
#include <libmrknlog.h>

int main (int argc, char* argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <flush-frequency-in-seconds>\n", argv[0]);
    exit(2);
  } else {
    int frequency = atoi(argv[1]);
    int totalSize = mrkn_get_total_log_size();
    int usedSize;
    int count = 1;
    while(1) {
      usedSize = mrkn_get_used_log_size();
```

```
      if (mrkn_flush_log() == 0) {
        LOGI("Flushed log (%d, %d of %d bytes). Waiting %d seconds before the ↩
            next flush.",
          count, usedSize, totalSize, frequency);
        count++;
      } else {
        LOGE("Failed to flush log. Waiting %d seconds before the next attempt",
          frequency);
      }
      sleep(frequency);
    }
  }
}
```

Note
This program is designed to run as a "daemon" simply by running infinitely in a `while(1) {...}` block.

Tip
We can use `LOGI()`, `LOGE()`, and other such macros, which simplify logging:

```
#define LOGI(...)  __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)
```

---

3. Next, we create our makefile (`Android.mk`), which defines links from our source to our library:
   device/marakana/alpha-common/bin/mrknlogd/Android.mk

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

include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := mrknlogd.c
LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../lib/libmrknlog/
LOCAL_SHARED_LIBRARIES := libmrknlog libc libcutils libutils
LOCAL_MODULE := mrknlogd
include $(BUILD_EXECUTABLE)
```

4. Now we can compile our daemon to test that it builds:

```
$ make -j10 mrknlogd
...
target Executable: mrknlogd (out/target/product/alpha/obj/EXECUTABLES/ ↩
    mrknlogd_intermediates/LINKED/mrknlogd)
target Non-prelinked: mrknlogd (out/target/product/alpha/symbols/system/bin/ ↩
    mrknlogd)
target Strip: mrknlogd (out/target/product/alpha/obj/EXECUTABLES/ ↩
    mrknlogd_intermediates/mrknlogd)
Install: out/target/product/alpha/system/bin/mrknlogd
```

5. Just like with our previous executable/library, we need to register our daemon with Alpha's `PRODUCT_PACKAGES`, or otherwise it will not be included in the final ROM:
   device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += mrknlogd
```

6. Before we can rebuild our ROM, we need to configure `init` to start our `mrknlogd` daemon as a `service`, and to do that, we need to modify the `init.rc` file:
   a. Since we don't want to modify the existing `init.rc` file, we'll copy it into our own directory:

```
$ cp system/core/rootdir/init.rc device/marakana/alpha-common/.
```

   b. Now we can register our service:
   device/marakana/alpha-common/init.rc

```
...
# Marakana's custom log-flushing daemon
service mrknlogd /system/bin/mrknlogd 60
    user system
    group log
    oneshot
```

A note about security
In order to get the size of the log buffer `/dev/log/main`, we need to be able to read from this "file", and for that, we need to be either `root` or belong to group `log` (we chose the latter):

```
$ adb shell ls -l /dev/log/main
crw-rw--w- root     log       10,  58 2011-07-13 11:08 main
```

---

   c. Finally, we need to tell Android to use our custom `init.rc` file (by copying it over the existing one during the build):
   device/marakana/alpha-common/alpha.mk

```
...
# Copy our init.rc file over the existing one (since ours contains extra  ↩
    changes)
PRODUCT_COPY_FILES += $(MY_PATH)/init.rc:root/init.rc
```

Note
Alternatively, we could have used `init.<board-name>.rc` file, but since we did not define a custom board name, we are defaulting to `goldfish` (which comes from the emulator), and `init.goldfish.rc` also already exists, so we'd need to overwrite it the same way as `init.rc`, so no gain there.

---

7. Now we can rebuild our entire device:

```
$ make -j10
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

8. Finally, we are ready to test our daemon:

```
# (re)start the emulator
$ emulator -kernel out/target/product/alpha/kernel &
# wait for the emulator to finish
# check out our daemon
$ adb shell ls -l /system/bin/mrknlogd
-rwxr-xr-x root     shell        5508 2011-07-13 02:25 mrknlogd
# check that it runs
```

```
$ adb shell ps | grep mrknlogd
system    37    1     1044   292   c00520f8 afd0bdac S /system/bin/mrknlogd
$ adb logcat | grep MRKN
I/MRKN Log Daemon(   37): Flushed log (1, 60 of 65536 bytes). Waiting 60 seconds  ↩
    before the next flush.
I/MRKN Log Daemon(   37): Flushed log (2, 34406 of 65536 bytes). Waiting 60  ↩
    seconds before the next flush.
I/MRKN Log Daemon(   37): Flushed log (3, 232 of 65536 bytes). Waiting 60 seconds ↩
     before the next flush.
^C
$ adb logcat -g
/dev/log/main: ring buffer is 64Kb (0Kb consumed), max entry is 4096b, max  ↩
    payload is 4076b
# good :-)
```

7.7 Exposing our Native Library via Java (i.e. JNI)

- Our shared native library (`libmrknlog.so`) is fine if we just develop in C/C++, but to utilize its functionality in the Java layers, we need to wrap it with some JNI love
- As we discussed in Chapter 2 section, here we will:
  - Create a Java "library" (`com.marakana.android.lib.log.LibLog`)
  - Declare some of its methods as `native` (e.g. `public static native void flushLog();`)
  - Extract the C header file (`com_marakana_android_lib_log_LibLog.h`) using `javah -jni ....LibLog`
  - Provide the implementation (`com_marakana_android_lib_log_LibLog.c`) - where we will finally get to utilize our shared library (`libmrknlog.so`)
  - Compile our Java/JNI library as `/system/frameworks/com.marakana.android.lib.log.jar` and `/system/lib/lib`
  - Create a simple Java program and test that our library works
- Let's get started:
  1. Since our Java/JNI library will primarily service the *Application Framework* layer, let's create a home for our framework components:

     ```
     $ mkdir device/marakana/alpha-common/framework
     ```

  2. Like with our libraries and executables, we need to "recursively" include makefiles in sub-directories of `.../framework/`, which is where our Java/JNI library will live:
     device/marakana/alpha-common/framework/Android.mk

     ```
     include $(call all-subdir-makefiles)
     ```

  3. Now, let's create the home for our Java/JNI library and two sub-directories for its Java and JNI parts:

     ```
     $ mkdir device/marakana/alpha-common/framework/mrknlog_jni
     $ mkdir device/marakana/alpha-common/framework/mrknlog_jni/java
     $ mkdir device/marakana/alpha-common/framework/mrknlog_jni/jni
     ```

  4. To include these sub-directories in the compilation (`java/` and `jni/`), we again need one of those "recursive" makefiles:
     device/marakana/alpha-common/framework/mrknlog_jni/Android.mk

```
include $(call all-subdir-makefiles)
```

5. Now, let's create the directory structure that reflects our Java library's package name (`com.marakana.android.lib.log`):

```
$ mkdir -p device/marakana/alpha-common/framework/mrknlog_jni/java/com/marakana/ ↩
    android/lib/log
```

6. Now let's create our Java "library" class (`LibLog.java`) and its accompanying exception (`LibLogException.java`):
device/marakana/alpha-common/framework/mrknlog_jni/java/com/marakana/android/lib/log/LibLog.java

```
package com.marakana.android.lib.log;

public class LibLog {
  public native static void flushLog() throws LibLogException;
  public native static int getTotalLogSize() throws LibLogException;
  public native static int getUsedLogSize() throws LibLogException;

  static {
     System.loadLibrary("mrknlog_jni");
  }
}
```

device/marakana/alpha-common/framework/mrknlog_jni/java/com/marakana/android/lib/log/LibLogException.java

```
package com.marakana.android.lib.log;

public class LibLogException extends RuntimeException {
  public LibLogException(String msg) {
    super(msg);
  }
}
```

7. While we are at it, we might as well create a simple `Main.java` class to test our Java/JNI library:
device/marakana/alpha-common/framework/mrknlog_jni/java/com/marakana/android/lib/log/Main.java

```
package com.marakana.android.lib.log;

/** @hide */
public class Main {
  public static void main (String[] args) {
    try {
      int usedSize = LibLog.getUsedLogSize();
      int totalSize = LibLog.getTotalLogSize();
      LibLog.flushLog();
      System.out.printf("Flushed log. Previously it was consuming %d of %d bytes\ ↩
          n",
          usedSize, totalSize);
    } catch (LibLogException e) {
      System.err.println("Failed to flush the log");
      e.printStackTrace();
    }
  }
}
```

8. For our Java/JNI library (deployed as `/system/framework/com.marakana.android.lib.log.jar`) to be accessible to the running applications, we need to explicitly expose its logical name (`com.marakana.android.lib.log`) via a simple XML mapping file (`com.marakana.android.lib.log.xml`):

device/marakana/alpha-common/framework/mrknlog_jni/java/com.marakana.android.lib.log.xml

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
  <library name="com.marakana.android.lib.log"
    file="/system/framework/com.marakana.android.lib.log.jar"/>
</permissions>
```

Note
This file will be deployed as `/system/etc/permissions/com.marakana.android.lib.log.xml` in the target image and applications that wish to use our library will have to reference it in their via `AndroidManifest.xml` with:
`<uses-library android:name="com.marakana.android.lib.log"`
`android:required="true"/>`

---

9. Now we are ready for the makefile (`Android.mk`):
device/marakana/alpha-common/framework/mrknlog_jni/java/Android.mk

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

# Build the library
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := com.marakana.android.lib.log
LOCAL_SRC_FILES := $(call all-java-files-under,.)
LOCAL_JAVA_LIBRARIES := core
LOCAL_NO_STANDARD_LIBRARIES := true
include $(BUILD_JAVA_LIBRARY)

# Build the documentation
include $(CLEAR_VARS)
LOCAL_SRC_FILES := $(call all-subdir-java-files) $(call all-subdir-html-files)
LOCAL_MODULE:= com.marakana.android.lib.log_doc
LOCAL_DROIDDOC_OPTIONS := com.marakana.android.lib.log
LOCAL_MODULE_CLASS := JAVA_LIBRARIES
LOCAL_DROIDDOC_USE_STANDARD_DOCLET := true
include $(BUILD_DROIDDOC)

# Copy com.marakana.android.lib.log.xml to /system/etc/permissions/
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := com.marakana.android.lib.log.xml
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE_PATH := $(TARGET_OUT_ETC)/permissions
LOCAL_SRC_FILES := $(LOCAL_MODULE)
include $(BUILD_PREBUILT)
```

Note
In this one file, we have three separate targets: to compile the code, to build the documentation (used by the SDK addon), and to copy the `com.marakana.android.lib.log.xml` file to `/system/etc/permissions/` (though this last one seems a too fancy for a simple copy command). Notice that we separate these three targets using the `include $(CLEAR_VARS)` macro.

---

10. For good measure, test that it compiles:

```
$ make -j10 com.marakana.android.lib.log com.marakana.android.lib.log.xml
...
Install: out/target/product/alpha/system/etc/permissions/com.marakana.android.lib ↩
    .log.xml
...
target Java: com.marakana.android.lib.log (out/target/common/obj/JAVA_LIBRARIES/ ↩
    com.marakana.android.lib.log_intermediates/classes)
...
Copying: out/target/common/obj/JAVA_LIBRARIES/com.marakana.android.lib. ↩
    log_intermediates/classes.jar
...
Install: out/target/product/alpha/system/framework/com.marakana.android.lib.log. ↩
    jar
```

11. Next, we need create the C header file for our library (using the `javah -jni` command), and fortunately we can re-use the compiled classes left for us in the `out/` directory by the previous step:

```
$ javah -jni \
    -d device/marakana/alpha-common/framework/mrknlog_jni/jni/ \
    -classpath out/target/common/obj/JAVA_LIBRARIES/com.marakana.android.lib. ↩
        log_intermediates/classes.jar \
    com.marakana.android.lib.log.LibLog
```

12. To check that it worked, we can take a look at the generated C header file (`..._LibLog.h`):
device/marakana/alpha-common/framework/mrknlog_jni/jni/com_marakana_android_lib_log_LibLog.h

```
...
JNIEXPORT void JNICALL Java_com_marakana_android_lib_log_LibLog_flushLog
  (JNIEnv *, jclass);
...
JNIEXPORT jint JNICALL Java_com_marakana_android_lib_log_LibLog_getTotalLogSize
  (JNIEnv *, jclass);
...
JNIEXPORT jint JNICALL Java_com_marakana_android_lib_log_LibLog_getUsedLogSize
  (JNIEnv *, jclass);
...
```

13. Now we can provide our implementation (`....LibLog.c`), which simply wraps calls to functions from `libmrknlog` and provides some basic error handling:
device/marakana/alpha-common/framework/mrknlog_jni/jni/com_marakana_android_lib_log_LibLog.c

```
#include <libmrknlog.h>
#include "com_marakana_android_lib_log_LibLog.h"

static void ThrowLibLogException(JNIEnv *env, const char *message) {
  jclass class = (*env)->FindClass(env, "com/marakana/android/lib/log/ ↩
      LibLogException");
  if (class != NULL) {
    (*env)->ThrowNew(env, class, message);
  }
  (*env)->DeleteLocalRef(env, class);
}

JNIEXPORT void JNICALL Java_com_marakana_android_lib_log_LibLog_flushLog
  (JNIEnv *env, jclass clazz) {
```

```
  if (mrkn_flush_log() != 0) {
    ThrowLibLogException(env, "Failed to flush log");
  }
}

JNIEXPORT jint JNICALL Java_com_marakana_android_lib_log_LibLog_getTotalLogSize
  (JNIEnv *env, jclass clazz) {
  jint result = mrkn_get_total_log_size();
  if (result < 0) {
    ThrowLibLogException(env, "Failed to get total log size");
  }
  return result;
}

JNIEXPORT jint JNICALL Java_com_marakana_android_lib_log_LibLog_getUsedLogSize
  (JNIEnv *env, jclass clazz) {
  jint result = mrkn_get_used_log_size();
  if (result < 0) {
    ThrowLibLogException(env, "Failed to get used log size");
  }
  return result;
}
```

14. Next, we create a makefile (`Android.mk`) to compile our JNI code into a shared library (`libmrknlog_jni.so`):
device/marakana/alpha-common/framework/mrknlog_jni/jni/Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := com_marakana_android_lib_log_LibLog.c
LOCAL_C_INCLUDES += $(JNI_H_INCLUDE) $(LOCAL_PATH)/../../../lib/libmrknlog
LOCAL_SHARED_LIBRARIES := libmrknlog
LOCAL_MODULE := libmrknlog_jni
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)
```

Note
As with our `libmrknlog.so`, we specified `LOCAL_PRELINK_MODULE := false`, which disables prelinking and allows us to keep components code self-contained. Alternatively, we could set `LOCAL_PRELINK_MODULE := true` (or simply omit it) but then we would have to register our new library in `build/core/prelink-linux-arm.map`:

```
...
libmrknlog_jni.so           0x9C800000 # libmrknlog_jni
```

---

15. Now we have all the pieces to compile our JNI library (into `/system/lib/libmrknlog_jni.so`):

```
$ make -j10 libmrknlog_jni
...
target SharedLib: libmrknlog_jni (out/target/product/alpha/obj/SHARED_LIBRARIES/ ↩
    libmrknlog_jni_intermediates/LINKED/libmrknlog_jni.so)
target Non-prelinked: libmrknlog_jni (out/target/product/alpha/symbols/system/lib ↩
    /libmrknlog_jni.so)
target Strip: libmrknlog_jni (out/target/product/alpha/obj/lib/libmrknlog_jni.so)
Install: out/target/product/alpha/system/lib/libmrknlog_jni.so
```

16. As before, since our Java/JNI library's components are marked as `LOCAL_MODULE_TAGS := optional`, in order to get them into the final ROM, we need to register them with Alpha's `PRODUCT_PACKAGES`:
device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += \
    com.marakana.android.lib.log \
    com.marakana.android.lib.log.xml \
    libmrknlog_jni
```

17. Now we can rebuild our entire device:

```
$ make -j10
...
target Prebuilt: com.marakana.android.lib.log.xml (out/target/product/alpha/obj/ ↩
    ETC/com.marakana.android.lib.log.xml_intermediates/com.marakana.android.lib. ↩
    log.xml)
...
Install: out/target/product/alpha/system/lib/libmrknlog_jni.so
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

18. Finally, we are ready to test our Java/JNI library via the `Main.main()` method:

```
# (re)start the emulator
$ emulator -kernel out/target/product/alpha/kernel &
# wait for the emulator to finish
# check out our Java library
$ adb shell ls -l /system/framework/com.marakana.android.lib.log.jar
-rw-r--r-- root     root         1471 2011-07-11 00:01 com.marakana.android.lib. ↩
    log.jar
# check out our Java library registry file
$ adb shell ls -l /system/etc/permissions/com.marakana.android.lib.log.xml
-rw-r--r-- root     root          179 2011-07-10 23:57 com.marakana.android.lib. ↩
    log.xml
# check out our JNI shared library
$ adb shell ls -l /system/lib/libmrknlog_jni.so
-rw-r--r-- root     root         5296 2011-07-11 01:41 libmrknlog_jni.so
# check if our utility is doing what it is supposed to
$ adb logcat -g
/dev/log/main: ring buffer is 64Kb (33Kb consumed), max entry is 4096b, max  ↩
    payload is 4076b
# now run our Java library's Main.main() by directly invoking the Dalvik VM
$ adb shell dalvikvm -cp /system/framework/com.marakana.android.lib.log.jar com. ↩
    marakana.android.lib.log.Main
Flushed log. Previously it was consuming 34346 of 65536 bytes
# check again
$ adb logcat -g
/dev/log/main: ring buffer is 64Kb (0Kb consumed), max entry is 4096b, max  ↩
    payload is 4076b
$ adb shell dalvikvm -cp /system/framework/com.marakana.android.lib.log.jar com. ↩
    marakana.android.lib.log.Main
Flushed log. Previously it was consuming 217 of 65536 bytes
# good :-)
```

7.8 Consuming our a Custom Java/JNI→Native Library via a Custom App

- Now that we have a native library, and a JNI wrapper around it, we could create a custom application to take advantage of its services
- Note that we will later wrap the JNI library with a client-service Binder library/solution, so the app we are creating now is just for demonstration purposes - not the final "solution"
- Let's get started:

1. Create a home directory for our Alpha apps:

```
$ mkdir device/marakana/alpha/app
```

> **Note**
> We created this directory under `alpha/` (and not `alpha-common/`) since we do not need to share this app with the SDK addon (which we'll be creating later).

2. As we should know by now, we need a makefile (`Android.mk`) to include `app/`'s the sub-directories:
device/marakana/alpha/app/Android.mk

```
include $(call all-subdir-makefiles)
```

3. Now we can create a sub-directory for our app (`MrknLogLibClient`), its basic directory structure (`res/`, and `src/`), and the source package (`com.marakana.android.loglibclient`) directory structure:

```
$ mkdir device/marakana/alpha/app/MrknLogLibClient
$ mkdir device/marakana/alpha/app/MrknLogLibClient/res
$ mkdir device/marakana/alpha/app/MrknLogLibClient/res/values
$ mkdir device/marakana/alpha/app/MrknLogLibClient/res/layout
$ mkdir device/marakana/alpha/app/MrknLogLibClient/src
$ mkdir -p device/marakana/alpha/app/MrknLogLibClient/src/com/marakana/android/ ↩
    loglibclient
```

4. We need some basic string resources (used in the UI):
device/marakana/alpha/app/MrknLogLibClient/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Marakana Log Lib Client</string>
    <string name="log_utilization_message">Using %1$d of %2$d bytes of the log  ↩
        buffer</string>
    <string name="flush_log_button">Flush Log Buffer</string>
</resources>
```

5. And we also need a simple layout (`log.xml`), with a text view, to show the current log utilization, and a button, to allow us to clear the log:
device/marakana/alpha/app/MrknLogLibClient/res/layout/log.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView android:layout_width="fill_parent" android:layout_height=" ↩
      wrap_content"
```

```
    android:gravity="center" android:id="@+id/output" />
  <Button android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:id="@+id/button" android:text="@string/flush_log_button"/>
</LinearLayout>
```

6. Now, we are ready to write our one-and only activity (`LogActivity`) utilizing `com.marakana.android.lib.log` Java/JNI library:
device/marakana/alpha/app/MrknLogLibClient/src/com/marakana/android/loglibclient/LogActivity.java

```
package com.marakana.android.loglibclient;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import com.marakana.android.lib.log.LibLog;

public class LogActivity extends Activity
  implements View.OnClickListener, Runnable {

  private TextView output;
  private Handler handler;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    super.setContentView(R.layout.log);
    this.output = (TextView) super.findViewById(R.id.output);
    Button button = (Button) super.findViewById(R.id.button);
    button.setOnClickListener(this);
    this.handler = new Handler();
  }

  private void updateOutput() {
    this.output.setText(
      super.getString(R.string.log_utilization_message,
      LibLog.getUsedLogSize(), LibLog.getTotalLogSize()));
  }

  @Override
  public void onResume() {
    super.onResume();
    this.handler.post(this);
  }

  @Override
  public void onPause() {
    super.onPause();
    this.handler.removeCallbacks(this);
  }

  public void onClick(View view) {
    LibLog.flushLog();
    this.updateOutput();
  }
```

```
  public void run() {
    this.updateOutput();
    this.handler.postDelayed(this, 1000);
  }
}
```

Note
Our activity uses a handler in order to request periodic (1 Hz) call-backs to the `run()` method, via which we simply update our UI with the log utilization data.

7. And, like any other app, we need to provide our own `AndroidManifest.xml` configuration file:
device/marakana/alpha/app/MrknLogLibClient/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana.android.loglibclient"
  android:versionCode="1"
  android:versionName="1.0">
  <uses-sdk android:minSdkVersion="8" />
  <uses-permission android:name="android.permission.READ_LOGS"/>
  <application android:label="@string/app_name">
    <uses-library android:name="com.marakana.android.lib.log" android:required=" ↩
        true"/>
    <activity android:name=".LogActivity" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Note
We need `<uses-permission android:name="android.permission.READ_LOGS"/>` so that our app's user ID gets added to the `log` group membership, or otherwise we won't be able to read from `/dev/log/main` (as discussed earlier).
Additionally, we added `<uses-library android:name="com.marakana.android.lib.log" android:required="true"/>` so that we get run-time access to the `com.marakana.android.lib.log` library.

8. And, like any other component, our app also needs its own makefile (`Android.mk`):
device/marakana/alpha/app/MrknLogLibClient/Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under,src)
LOCAL_JAVA_LIBRARIES := com.marakana.android.lib.log
LOCAL_PACKAGE_NAME := MrknLogLibClient
LOCAL_SDK_VERSION := current
LOCAL_PROGUARD_ENABLED := disabled
include $(BUILD_PACKAGE)
```

Note
We are referencing our `com.marakana.android.lib.log` library here as well, for the sake of the compiler. Also, notice that we are using `include $(BUILD_PACKAGE)` in order to build our code as an app.

---

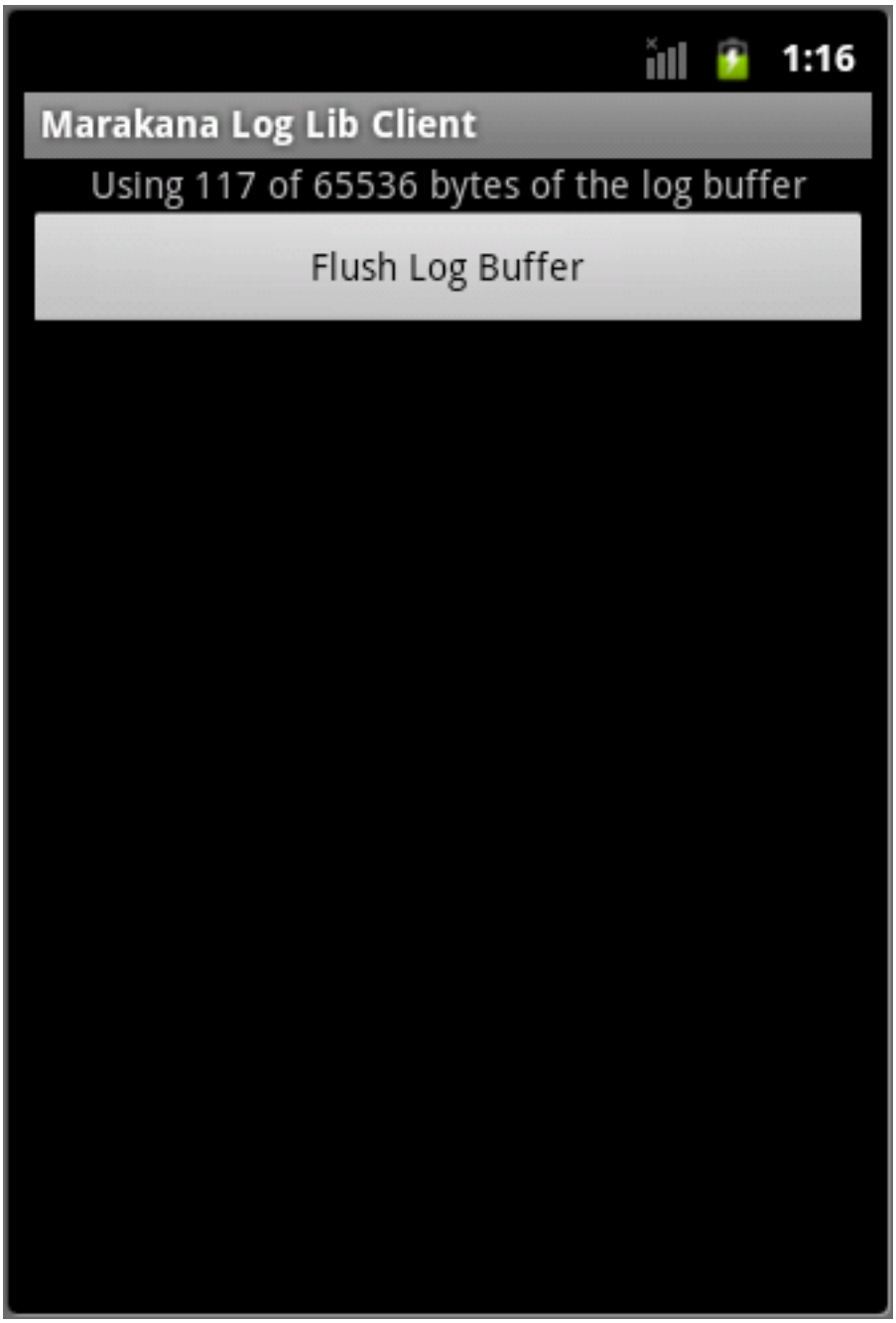
9. And, also like any other component that we wish to include in our ROM, we need to register `MrknLogLibClient` with the device's main makefile (`full_alpha.xml`):
   device/marakana/alpha/full_alpha.mk

```
...
PRODUCT_PACKAGES += MrknLogLibClient
```

10. Now we can compile it:

```
$ make -j10
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

11. And finally, we can test everything:
    a. Restart the emulator
    b. Launch the `Marakana Log Lib Client` app
    c. Test that you can flush the log buffer



7.9 Exposing our Custom Library via a Custom IPC/Binder Service

- Let's suppose that we were able to control access to our "driver" (even though it is not really ours), such that:
  - The "read" operations (getting the used/total sizes of the log buffer) are implicitly permitted
  - The "write" operation (flushing the log buffer) requires an explicit permission check
- We can do this by wrapping "our driver" (and the custom library that enables these operations) with:

  - A custom Binder-based service where we control access to the driver
  - A custom Java-based manager (library) where we enable transparent access to our service

- Let's get started:

  1. We'll start off by creating a directory structure for the custom manager library, since that's where our service descriptor (`ILogService.aidl`) is going to live:

```
$ mkdir device/marakana/alpha-common/framework/mrknlogservice
$ mkdir -p device/marakana/alpha-common/framework/mrknlogservice/com/marakana/ ↩
    android/service/log
```

  2. Let's create a simple AIDL description of our service (`LibLogService.aidl`):
     device/marakana/alpha-common/framework/mrknlogservice/com/marakana/android/service/log/ILogService.aidl

```
package com.marakana.android.service.log;

/**
 * System-private API for talking to the LogService.
 *
 * {@hide}
 */
interface ILogService {
  void flushLog();
  int getTotalLogSize();
  int getUsedLogSize();
}
```

Note
Notice that the methods described by `ILogService.aidl` closely match what our `LibLog` JNI-bridge provides. Also, we are using `{@hide}` to *hide* our AIDL interface from the documentation we'll be producing later for our SDK addon. Why hide it? Because the clients will use our service via a manager proxy that we'll be creating next.

---

  3. We don't want to force the "complexity" of Binder upon our clients, so we provide them with a convenience `LogManager` proxy to our (yet-to-be-created) service:
     device/marakana/alpha-common/framework/mrknlogservice/com/marakana/android/service/log/LogManager.java

```
package com.marakana.android.service.log;

import android.os.IBinder;
import android.os.RemoteException;
import android.os.ServiceManager;
import android.util.Log;

public class LogManager {
  private static final String TAG = "LogManager";
  private static final String REMOTE_SERVICE_NAME = ILogService.class.getName();
  private final ILogService service;

  public static LogManager getInstance() {
    return new LogManager();
  }
```

```
  private LogManager() {
    Log.d(TAG, "Connecting to ILogService by name [" + REMOTE_SERVICE_NAME + "]") ↩
        ;
    this.service = ILogService.Stub.asInterface(ServiceManager.getService( ↩
        REMOTE_SERVICE_NAME));
    if (this.service == null) {
      throw new IllegalStateException("Failed to find ILogService by name [" +  ↩
          REMOTE_SERVICE_NAME + "]");
    }
  }

  public void flushLog() {
    try {
      Log.d(TAG, "Flushing logs. If it works, you won't see this message.");
      this.service.flushLog();
    } catch (RemoteException e) {
      throw new RuntimeException("Failed to flush log", e);
    }
  }

  public int getTotalLogSize() {
    try {
      return this.service.getTotalLogSize();
    } catch (RemoteException e) {
      throw new RuntimeException("Failed to get total log size", e);
    }
  }

  public int getUsedLogSize() {
    try {
      return this.service.getUsedLogSize();
    } catch (Exception e) {
      throw new RuntimeException("Failed to get used log size", e);
    }
  }
}
```

Note
We are using `android.os.ServiceManager` to lookup an object providing `ILogService` implementation. This is going to be our service, which we yet have to create and register with the service-manager.

---

4. For our clients to access `LogManager`, we need to expose it as a Java library - so we create an XML descriptor for it:
   device/marakana/alpha-common/framework/mrknlogservice/com.marakana.android.service.log.xml

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
  <library name="com.marakana.android.service.log"
    file="/system/framework/com.marakana.android.service.log.jar"/>
</permissions>
```

5. Now we are ready ready to create a makefile (`Android.mk`) with rules to compile our library, build its documentation, and copy its `com.marakana.android.service.log.xml` descriptor to `/system/etc/permissions/`:
   device/marakana/alpha-common/framework/mrknlogservice/Android.mk

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

# Build the library
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := com.marakana.android.service.log
LOCAL_SRC_FILES := $(call all-java-files-under,.)
LOCAL_SRC_FILES += com/marakana/android/service/log/ILogService.aidl
LOCAL_JAVA_STATIC_LIBRARIES := android-common
LOCAL_JAVA_LIBRARIES := core
include $(BUILD_JAVA_LIBRARY)

# Build the documentation
include $(CLEAR_VARS)
LOCAL_SRC_FILES := $(call all-subdir-java-files) $(call all-subdir-html-files)
LOCAL_MODULE:= com.marakana.android.service.log_doc
LOCAL_DROIDDOC_OPTIONS := com.marakana.android.service.log
LOCAL_MODULE_CLASS := JAVA_LIBRARIES
LOCAL_DROIDDOC_USE_STANDARD_DOCLET := true
include $(BUILD_DROIDDOC)

# Copy com.marakana.android.service.log.xml to /system/etc/permissions/
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := com.marakana.android.service.log.xml
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE_PATH := $(TARGET_OUT_ETC)/permissions
LOCAL_SRC_FILES := $(LOCAL_MODULE)
include $(BUILD_PREBUILT)
```

6. To test that we did everything correctly, we can compile our library:

```
$ make -j10 com.marakana.android.service.log com.marakana.android.service.log.xml
...
target Prebuilt: com.marakana.android.service.log.xml
...
Install: out/target/product/alpha/system/etc/permissions/com.marakana.android. ↩
    service.log.xml
...
Aidl: com.marakana.android.service.log <= device/marakana/alpha-common/framework/ ↩
    mrknlogservice/com/marakana/android/service/log/ILogService.aidl
...
target Jar: com.marakana.android.service.log
...
Install: out/target/product/alpha/system/framework/com.marakana.android.service. ↩
    log.jar
...
```

7. For our `com.marakana.android.service.log` library to be included in the final ROM, we need to add it to the `alpha.mk` makefile:
   device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += \
    com.marakana.android.service.log \
    com.marakana.android.service.log.xml
```

8. Now we can create our `MrknLogService`, which will implement `ILogService`, but since we'll define our service as an *application*, we need first to create a directory for `alpha-common` apps:

```
$ mkdir device/marakana/alpha-common/app
```

9. And, as we know by now, we need a makefile to include `app/`'s sub-directories into the build:
device/marakana/alpha-common/app/Android.mk

```
include $(call all-subdir-makefiles)
```

10. Next, we create a directory for our service (`MrknLogService`), its basic directory structure (`res/`, and `src/`), and the source package (`com.marakana.android.loglibclient`) directory structure:

```
$ mkdir device/marakana/alpha-common/app/MrknLogService
$ mkdir device/marakana/alpha-common/app/MrknLogService/res
$ mkdir device/marakana/alpha-common/app/MrknLogService/res/values
$ mkdir device/marakana/alpha-common/app/MrknLogService/src
$ mkdir -p device/marakana/alpha-common/app/MrknLogService/src/com/marakana/ ↩
    android/logservice
```

11. We need some basic string resources:
device/marakana/alpha-common/app/MrknLogService/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="flush_log_permission_label">Flush Log</string>
  <string name="flush_log_permission_description">Applications with this  ↩
      permission will be able
    to clear the logs, potentially covering their own tracks of malicious  ↩
       behavior.</string>
</resources>
```

> **Note**
> These strings are used for our custom permission that we'll be creating next.

12. Next, let's define our app's `AndroidManifest.xml` file:
device/marakana/alpha-common/app/MrknLogService/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana.android.logservice"
  android:versionCode="1"
  android:versionName="1.0"
  android:sharedUserId="android.uid.system">
  <uses-sdk android:minSdkVersion="8" />
  <uses-permission android:name="android.permission.READ_LOGS"/>
  <application android:name=".LogServiceApp" android:persistent="true">
    <uses-library android:name="com.marakana.android.service.log"  ↩
       android:required="true"/>
    <uses-library android:name="com.marakana.android.lib.log" android:required=" ↩
       true"/>
  </application>
  <permission android:name="com.marakana.android.logservice.FLUSH_LOG"
    android:protectionLevel="dangerous"
    android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
```

```
    android:label="@string/flush_log_permission_label"
    android:description="@string/flush_log_permission_description"/>
</manifest>
```

Note

In this manifest file, there are a few items of interest:
We required that our app run with the `system` user (so that we can access the `ServiceManager`) by adding `android:sharedUserId="android.uid.system"` attribute to the manifest.
We gave ourselves access to the `log` group (so that we can read from `/dev/log/main`) with the `<uses-permission android:name="android.permission.READ_LOGS"/>` entry.
We defined a custom `LogServiceApp` application class, which once loaded, will in turn load our service (`ILogServiceImpl`) and register it with the `ServiceManager`. We defined our application as `android:persistent="true"`, which means that the `ActivityManager` will automatically launch it on boot, and it will never demote its importance (`/proc/<our-service-app-pid>/oom_adj=-12`), so the low memory killer will never kill it. Additionally, gave our application access to `com.marakana.android.service.log` library to get access to `ILogService` interface and `com.marakana.android.lib.log` library to get access to `LibLog` JNI bridge.
Finally, we defined our custom permission (`....FLUSH_LOG`), which we'll later enforce in our service class implementation (`ILogServiceImpl`).

---

13. Now we can provide the implementation `ILogServiceImpl` for our AIDL-defined interface (`ILogService`) we created earlier:
device/marakana/alpha-common/app/MrknLogService/src/com/marakana/android/logservice/ILogServiceImpl.java

```
package com.marakana.android.logservice;

import android.content.Context;
import android.content.pm.PackageManager;
import android.os.RemoteException;
import android.util.Log;
import com.marakana.android.service.log.ILogService;
import com.marakana.android.lib.log.LibLog;

class ILogServiceImpl extends ILogService.Stub {
  private static final String TAG = "ILogServiceImpl";
  private final Context context;

  ILogServiceImpl(Context context) {
    this.context = context;
  }

  public void flushLog() throws RemoteException {
    if (this.context.checkCallingOrSelfPermission(Manifest.permission.FLUSH_LOG)  ↩
        !=
        PackageManager.PERMISSION_GRANTED) {
      throw new SecurityException("Requires FLUSH_LOG permission");
    }
    Log.d(TAG, "Flushing logs. If it works, you won't see this message.");
    LibLog.flushLog();
  }

  public int getUsedLogSize() throws RemoteException {
    return LibLog.getUsedLogSize();
  }
```

```
  public int getTotalLogSize() throws RemoteException {
    return LibLog.getTotalLogSize();
  }
}
```

Note
For the most part, our service simply wraps our JNI library, except that it requires that the caller be granted our custom `Manifest.permission.FLUSH_LOG` permission before calling the `LibLog.flushLog()` method.

14. For our `ILogServiceImpl` to be accessible to our `LogManager`, we need to instantiate it and register it with the `ServiceManager` - we do this in our custom `LogServiceApp` application class loaded from the manifest file:
device/marakana/alpha-common/app/MrknLogService/src/com/marakana/android/logservice/LogServiceApp.java

```
package com.marakana.android.logservice;

import android.app.Application;
import android.os.ServiceManager;
import android.util.Log;
import com.marakana.android.service.log.ILogService;

public class LogServiceApp extends Application {
  private static final String TAG = "LogServiceApp";
  private static final String REMOTE_SERVICE_NAME = ILogService.class.getName();
  private ILogServiceImpl serviceImpl;

  public void onCreate() {
    super.onCreate();
    this.serviceImpl = new ILogServiceImpl(this);
    ServiceManager.addService(REMOTE_SERVICE_NAME, this.serviceImpl);
    Log.d(TAG, "Registered [" + serviceImpl.getClass().getName() + "] as [" +  ↩
        REMOTE_SERVICE_NAME + "]");
  }

  public void onTerminate() {
    super.onTerminate();
    Log.d(TAG, "Terminated");
  }
}
```

Note
Normally, regular applications cannot get access to `ServiceManager` class (because it is hidden) nor to the `servicemanager` daemon (because it enforces UID-based restrictions). To work around these limitations, we'll compile our app with the framework classes (where `ServiceManager` is *not* hidden), and run it with the `system` user (which *does* have access to `servicemanager` daemon).

15. We are now ready for our makefile (`Android.mk`):
device/marakana/alpha-common/app/MrknLogService/Android.mk

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

```
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under,src)
LOCAL_REQUIRED_MODULES := \
    com.marakana.android.service.log \
    com.marakana.android.lib.log
LOCAL_JAVA_LIBRARIES := \
    com.marakana.android.service.log \
    com.marakana.android.lib.log \
    core \
    framework
LOCAL_PACKAGE_NAME := MrknLogService
LOCAL_SDK_VERSION := current
LOCAL_PROGUARD_ENABLED := disabled
LOCAL_CERTIFICATE := platform
include $(BUILD_PACKAGE)
```

Note
The setting `LOCAL_JAVA_LIBRARIES := ... framework` allows us to reference `ServiceManager` and `LOCAL_CERTIFICATE := platform` makes it possible for us to run with the `system` user and thereby access the `servicemanager` daemon.

---

16. Let's compile our code:

```
$ make -j10 MrknLogService
...
Install: out/target/product/alpha/system/app/MrknLogService.apk
```

17. We want our `MrknLogService` to be included in the final ROM, so we add it to `alpha.mk`:
device/marakana/alpha-common/alpha.mk

```
...
PRODUCT_PACKAGES += MrknLogService
```

18. Compile the entire device:

```
$ make -j10
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

19. It's best to test our new manager→service→library→driver proxy via a real application, so that's what we'll be creating next.

7.10 Building a Custom App Using a Custom Service Manager

- Similar to how we implemented `MrknLogLibClient` app, we are now going to create a new app (`MrknLogServiceClient`), that will take advantage of our IPC/Binder-based APIs
- Since most of the steps are going to be very similar, we'll focus only on the parts that are different:

1. Create a new directory for our new app (`MrknLogServiceClient`), its basic directory structure (`res/`, and `src/`), and the source package (`com.marakana.android.logserviceclient`) directory structure:

```
$ mkdir device/marakana/alpha/app/MrknLogServiceClient
$ mkdir device/marakana/alpha/app/MrknLogServiceClient/res
$ mkdir device/marakana/alpha/app/MrknLogServiceClient/res/values
$ mkdir device/marakana/alpha/app/MrknLogServiceClient/res/layout
$ mkdir device/marakana/alpha/app/MrknLogServiceClient/src
$ mkdir -p device/marakana/alpha/app/MrknLogServiceClient/src/com/marakana/ ↩
    android/logserviceclient
```

2. Just like before, we need some basic string resources (used in the UI):
   device/marakana/alpha/app/MrknLogServiceClient/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Marakana Log Service Client</string>
  <string name="log_utilization_message">Using %1$d of %2$d bytes of the log  ↩
      buffer</string>
  <string name="flush_log_button">Flush Log Buffer</string>
</resources>
```

3. And we also need a simple layout (`log.xml`), which is exactly the same as the one we created for `MrknLogLibClient` before:
   device/marakana/alpha/app/MrknLogServiceClient/res/layout/log.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView android:layout_width="fill_parent" android:layout_height=" ↩
      wrap_content"
    android:gravity="center" android:id="@+id/output" />
  <Button android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:id="@+id/button" android:text="@string/flush_log_button"/>
</LinearLayout>
```

4. Our activity (`LogActivity`) will be similar to the one we wrote before, except that in this case will be using `com.marakana.android.service.log` Java (Binder) library and its `LogManager` APIs:
   device/marakana/alpha/app/MrknLogServiceClient/src/com/marakana/android/logserviceclient/LogActivity.java

```
package com.marakana.android.logserviceclient;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

import com.marakana.android.service.log.LogManager;

public class LogActivity extends Activity implements Runnable, OnClickListener {

  private TextView output;

  private Handler handler;
```

```
  private LogManager logManager;

  public void onCreate(Bundle savedInstanceState) {
    this.logManager = LogManager.getInstance();
    super.onCreate(savedInstanceState);
    super.setContentView(R.layout.log);
    this.output = (TextView)super.findViewById(R.id.output);
    Button button = (Button)super.findViewById(R.id.button);
    button.setOnClickListener(this);
    this.handler = new Handler();
  }

  private void updateOutput() {
    this.output.setText(super.getString(R.string.log_utilization_message,
      this.logManager.getUsedLogSize(), this.logManager.getTotalLogSize()));
  }

  @Override
  public void onResume() {
    super.onResume();
    this.handler.post(this);
  }

  @Override
  public void onPause() {
    super.onPause();
    this.handler.removeCallbacks(this);
  }

  public void onClick(View view) {
    this.logManager.flushLog();
    this.updateOutput();
  }

  public void run() {
    this.updateOutput();
    this.handler.postDelayed(this, 1000);
  }
}
```

5. Like before, we need a `AndroidManifest.xml` file, but this time around we are using the `....FLUSH_LOG` permission, and `com.marakana.android.service.log` library:
device/marakana/alpha/app/MrknLogServiceClient/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana.android.logserviceclient"
  android:versionCode="1" android:versionName="1.0">
  <uses-sdk android:minSdkVersion="8" />
  <uses-permission android:name="com.marakana.android.logservice.FLUSH_LOG" />
  <application android:label="@string/app_name">
    <uses-library android:name="com.marakana.android.service.log"
      android:required="true" />
    <activity android:name=".LogActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
```

```
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

6. And, also like before, our app needs its own makefile (`Android.mk`):
   device/marakana/alpha/app/MrknLogServiceClient/Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under,src)
LOCAL_JAVA_LIBRARIES := com.marakana.android.service.log
LOCAL_PACKAGE_NAME := MrknLogServiceClient
LOCAL_SDK_VERSION := current
LOCAL_PROGUARD_ENABLED := disabled
include $(BUILD_PACKAGE)
```

Note
We are referencing our `com.marakana.android.service.log` library here as well, for the sake of the compiler.

---

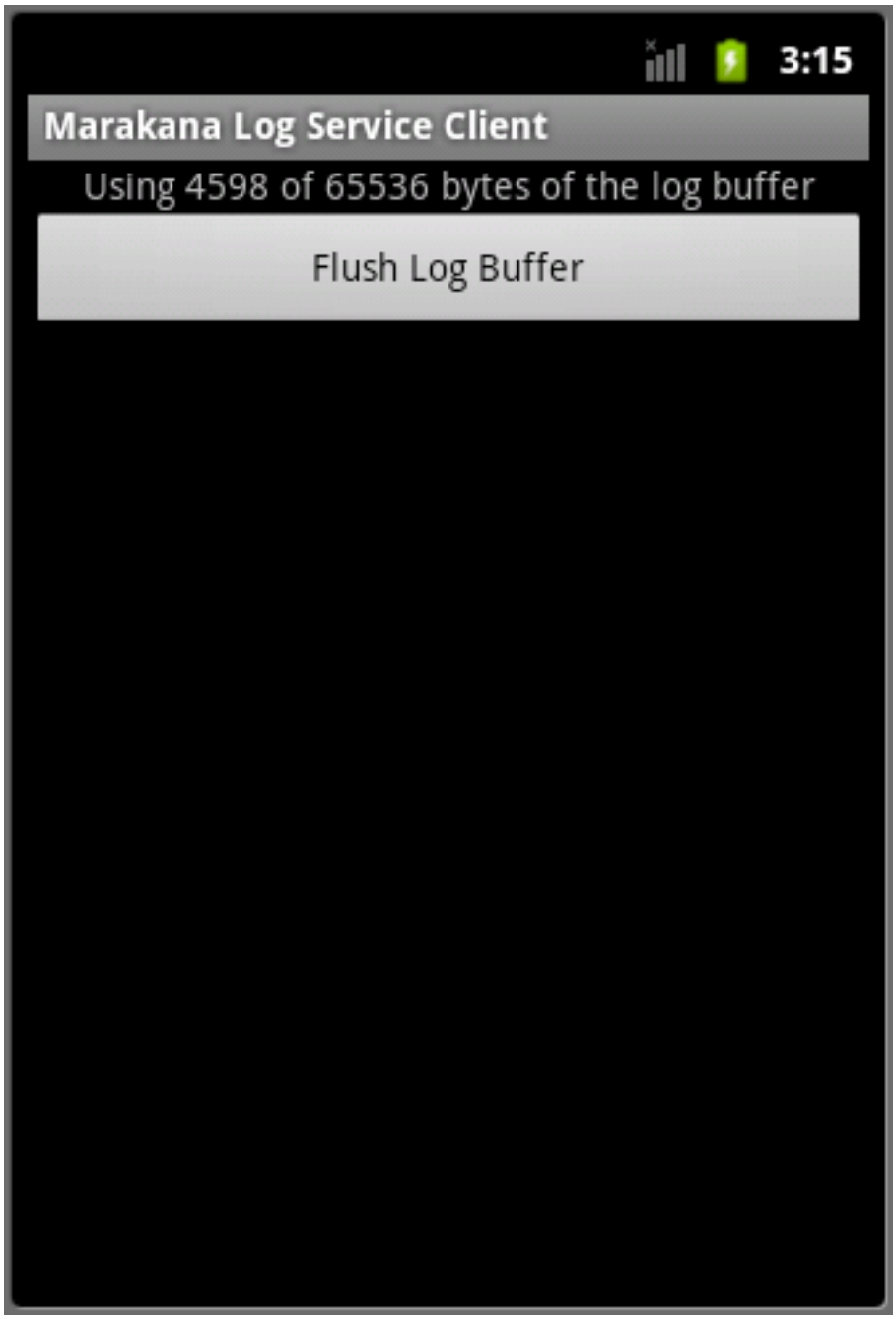
7. And, as before, we need to register `MrknLogServiceClient` with the device's main makefile (`full_alpha.xml`) in order to get it included in the ROM:
   device/marakana/alpha/full_alpha.mk

```
...
PRODUCT_PACKAGES += MrknLogServiceClient
```

8. Now we can compile our entire device:

```
$ make -j10
...
Install system fs image: out/target/product/alpha/system.img
Installed file list: out/target/product/alpha/installed-files.txt
```

9. And finally, we can test everything:
   a. Restart the emulator
   b. Launch the `Marakana Log Service Client` app
   c. Test that you can flush the log buffer



7.11 Creating a Custom SDK Add-on

- We can create apps for our own custom device (utilizing our custom libraries), but what if we wanted to expose those APIs to 3rd part developers? We'd need an SDK Addon
- Android SDK addons allow 3rd party developers to get access to our custom system images (including our custom libraries), compile their apps against our APIs, and test them on an emulator
- Note that custom hardware would have to be simulated (e.g. the way GPS is simulated on the emulator)

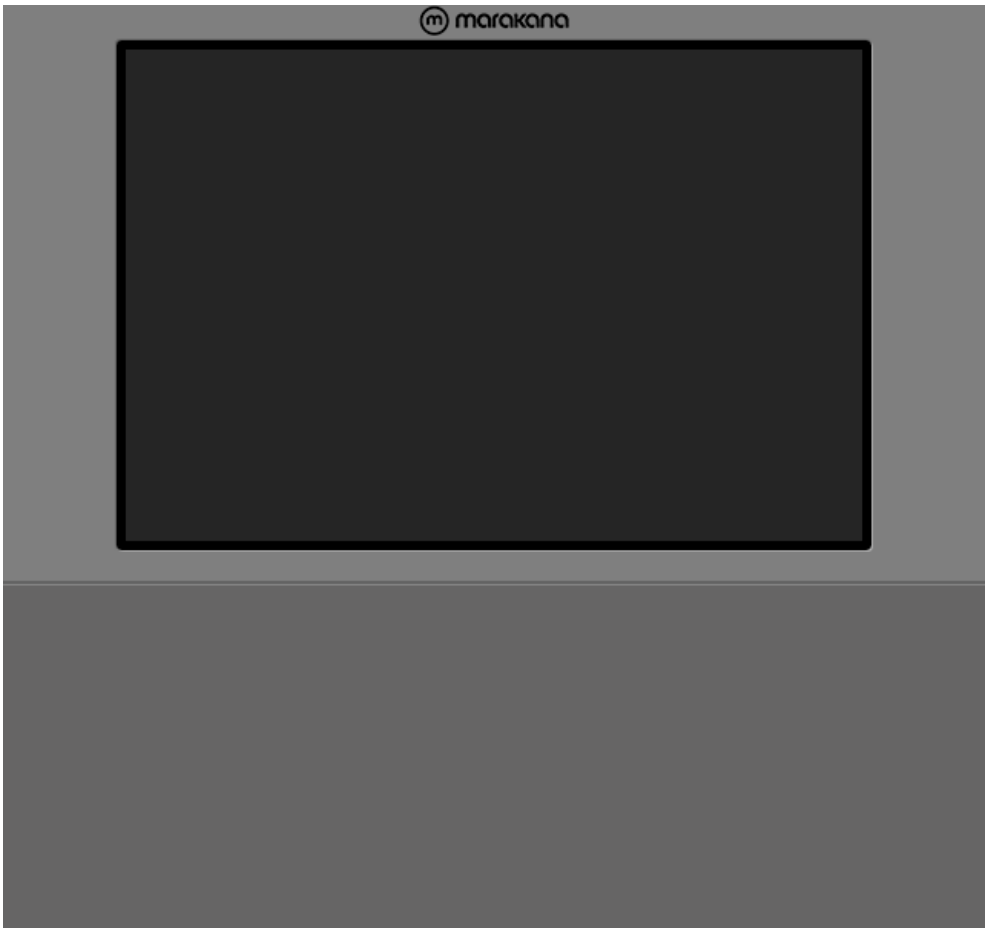
- Fortunately, our SDK addon will be able to utilize most of what we provided in `alpha-common/` so we'll focus only on the differences
- Let's get started:

1. We need a directory for our addon, but we've already created it before: `device/marakana/alpha-sdk_addon`
2. Though this is purely optional, let's create a custom emulator for our add-on, by copying one of the existing ones:

```
$ mkdir device/marakana/alpha-sdk_addon/skins
$ cp -r sdk/emulator/skins/HVGA device/marakana/alpha-sdk_addon/skins/ ↩
    MrknHvgaMdpi
```

   a. We could now customize the portrait background of our skin (`device/marakana/alpha-sdk_addon/skins/MrknHv` - say by adding our custom logo to it

   b. We could also customize the landscape background of our skin (`device/marakana/alpha-sdk_addon/skins/MrknH` - also by adding our custom logo to it



c. We could also customize other images, layout controls (`layout`), and hardware-specific values (`hardware.ini`)

3. To simulate that our devices supports NFC, we would:
   a. Create a directory (`nxp`):

```
$ mkdir device/marakana/alpha-sdk_addon/nxp
```

   b. Add a configuration file (`com.nxp.mifare.xml`) declaring NFC-feature support:
   device/marakana/alpha-sdk_addon/nxp/com.nxp.mifare.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Devices that support MIFARE need to declare NXP's feature constant -->
<permissions>
  <feature name="com.nxp.mifare" />
</permissions>
```

> **Note**
> NFC-support is not really required for our particular product (*Alpha*), but this is an example of how some hardware-specific features are defined.

4. While on the subject of hardware, let's define skin-independent set of hardware properties (`hardware.ini`) for our emulated device:
   device/marakana/alpha-sdk_addon/hardware.ini

```
# Custom hardware options for the add-on.
# Properties defined here impact all AVD targeting this add-on.
# Each skin can also override those values with its own hardware.ini file.
vm.heapSize = 24
```

5. Next, we define the properties of our SDK addon (`manifest.ini`) - name, description, API version, revision, libraries, and skin - as these properties are required by development tools (such as SDK Manager and Eclipse) using our addon:
   device/marakana/alpha-sdk_addon/manifest.ini

```
name=Alpha Add-On
vendor=Marakana
description=Marakana Alpha Add-on
api=10
revision=1
libraries=com.marakana.android.lib.log;com.marakana.android.service.log
com.marakana.android.lib.log=com.marakana.android.lib.log.jar;Marakana Log  ↩
    Library
com.marakana.android.service.log=com.marakana.android.service.log.jar;Marakana  ↩
    Log Service
skin=MrknHvgaMdpi
```

6. Next, we define a list of classes to be included (`+<package-name>.(*|<class-name>)`) or excluded (`-<package-name>.(*|<class-name>)`) from the generated SDK addon's libraries:
   device/marakana/alpha-sdk_addon/alpha_sdk_addon_stub_defs.txt

```
+com.marakana.android.lib.log.*
-com.marakana.android.lib.log.Main
+com.marakana.android.service.log.*
```

> **Note**
> These classes are known as SDK Addon Stub Definitions, and this file is processed by `development/tools/mkstubs` during the build time. The Java tool `mkstubs` loads this file via `PRODUCT_SDK_ADDON_STUB_DEFS` makefile property.

7. Now we are ready for our addon's main makefile (`alpha_sdk_addon.mk`), which will be loaded later by `AndroidProducts.mk`:
   device/marakana/alpha-sdk_addon/alpha_sdk_addon.mk

```
# Include the common stuff
include device/marakana/alpha-common/alpha.mk

# List of modules to include in the the add-on system image
PRODUCT_PACKAGES += \
    com.marakana.android.lib.log_doc \
    com.marakana.android.service.log_doc \

# The name of this add-on (for the SDK)
PRODUCT_SDK_ADDON_NAME := marakana_alpha_addon

# Copy the following files for this add-on's SDK
PRODUCT_SDK_ADDON_COPY_FILES := \
    $(LOCAL_PATH)/manifest.ini:manifest.ini \
    $(LOCAL_PATH)/hardware.ini:hardware.ini \
```

```
    $(call find-copy-subdir-files,*,$(LOCAL_PATH)/skins/MrknHvgaMdpi,skins/ ↩
        MrknHvgaMdpi)


# Copy the jar files for the libraries (APIs) exposed in this add-on's SDK
PRODUCT_SDK_ADDON_COPY_MODULES := \
    com.marakana.android.lib.log:libs/com.marakana.android.lib.log.jar \
    com.marakana.android.service.log:libs/com.marakana.android.service.log.jar

PRODUCT_SDK_ADDON_STUB_DEFS := $(LOCAL_PATH)/alpha_sdk_addon_stub_defs.txt

# Define the name of the documentation to generate for this add-on's SDK
PRODUCT_SDK_ADDON_DOC_MODULE := \
    com.marakana.android.service.log_doc

# Since the add-on is an emulator, we also need to explicitly copy the kernel to  ↩
    images
PRODUCT_SDK_ADDON_COPY_FILES += $(LOCAL_KERNEL):images/kernel-qemu

# This add-on extends the default sdk product.
$(call inherit-product, $(SRC_TARGET_DIR)/product/sdk.mk)

# The name of this add-on (for the build system)
# Use 'make PRODUCT-<PRODUCT_NAME>-sdk_addon' to build the an add-on,
# so in this case, we would run 'make PRODUCT-marakana_alpha_addon-sdk_addon'
PRODUCT_NAME := marakana_alpha_addon
PRODUCT_DEVICE := alpha
PRODUCT_MODEL := Marakana Alpha SDK Addon Image for Emulator
```

Note
As we can see from the comments, this file extends from `build/target/product/sdk.mk`, includes everything from `alpha-commmon`, adds support for library documentation, defines the addon's name (`PRODUCT_SDK_ADDON_NAME`), specifies the files/modules to be copied (`manifest.ini`, `hardware.ini`, skin, libraries, and kernel) into the addon (`PRODUCT_SDK_ADDON_COPY_FILES`), loads stub defs, and defines the addon's product info (like `PRODUCT_NAME`).

---

8. Just like with our `alpha` product, we need to create `AndroidProducts.mk` file, which the build system looks for to get a list of the actual make files (in this just `alpha_sdk_addon.mk`) to process for this "product" (i.e. SDK addon):
   device/marakana/alpha-sdk_addon/AndroidProducts.mk

```
PRODUCT_MAKEFILES := $(LOCAL_DIR)/alpha_sdk_addon.mk
```

9. Just to be on the safe side, let's make sure that make files from all sub-directories also get included:
   device/marakana/alpha-sdk_addon/Android.mk

```
include $(call all-subdir-makefiles)
```

Note
This file is unnecessary in this particular case, but it does not hurt to have it in case we add more addon-specific components down the road, which will most likely be stored in their own sub-directories.

---

10. Now we are ready to compile our addon:

```
$ make -j10 PRODUCT-marakana_alpha_addon-sdk_addon
...
Packaging SDK Addon: out/host/linux-x86/sdk_addon/marakana_alpha_addon-eng. ↩
    student-linux-x86.zip
```

11. Next, to test that it works, we first install our SDK to our Android SDK's `add-ons/` directory:

```
$ unzip out/host/linux-x86/sdk_addon/marakana_alpha_addon-eng.student-linux-x86. ↩
    zip -d /home/student/android/sdk/add-ons/
Archive:  out/host/linux-x86/sdk_addon/marakana_alpha_addon-eng.student-linux-x86 ↩
    .zip
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/images/
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/images/ramdisk.img
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/images/userdata.img
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/images/NOTICE.txt
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/images/kernel-qemu
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/images/system.img
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/docs/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/docs/reference/
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/package-list
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/index-all.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/overview-tree.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/deprecated-list.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/allclasses-frame.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/constant-values.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/allclasses-noframe.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/index.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/stylesheet.css
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/help-doc.html
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/docs/reference/com/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/docs/reference/com/marakana/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
       linux-x86/docs/reference/com/marakana/android/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
```

```
      linux-x86/docs/reference/com/marakana/android/service/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/com/marakana/android/service/log/
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/com/marakana/android/service/log/package-tree.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/com/marakana/android/service/log/LogManager.html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/com/marakana/android/service/log/package-summary. ↩
     html
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/com/marakana/android/service/log/package-frame. ↩
     html
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/docs/reference/resources/
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/docs/reference/resources/inherit.gif
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/hardware.ini
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/manifest.ini
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/skins/
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/skins/MrknHvgaMdpi/
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/spacebar.png
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/keyboard.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/arrow_left.png
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/key.png
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/background_port.png
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/controls.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/button.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/hardware.ini
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/select.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/arrow_right.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/arrow_down.png
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/skins/MrknHvgaMdpi/layout
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/skins/MrknHvgaMdpi/background_land.png
 extracting: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
     linux-x86/skins/MrknHvgaMdpi/arrow_up.png
   creating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
      linux-x86/libs/
  inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
```

```
    linux-x86/libs/com.marakana.android.service.log.jar
inflating: /home/student/android/sdk/add-ons/marakana_alpha_addon-eng.student- ↩
    linux-x86/libs/com.marakana.android.lib.log.jar
```

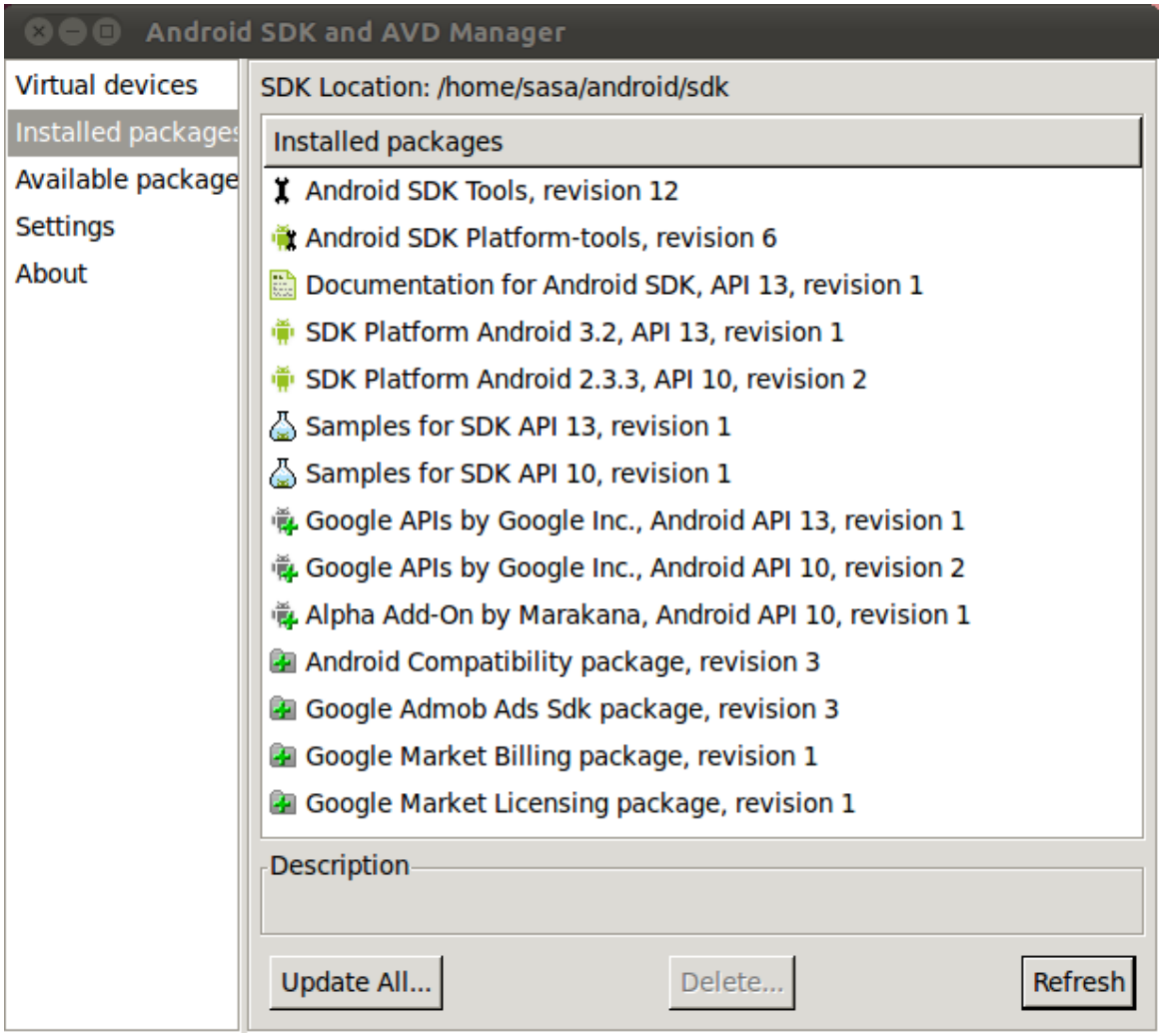
Tip
Adjust the path to Android SDK directory as necessary.

12. Let's test that it shows up in our *Android SDK and ADV Manager*:
    a. Run

```
$ /home/student/android/sdk/tools/android $
```

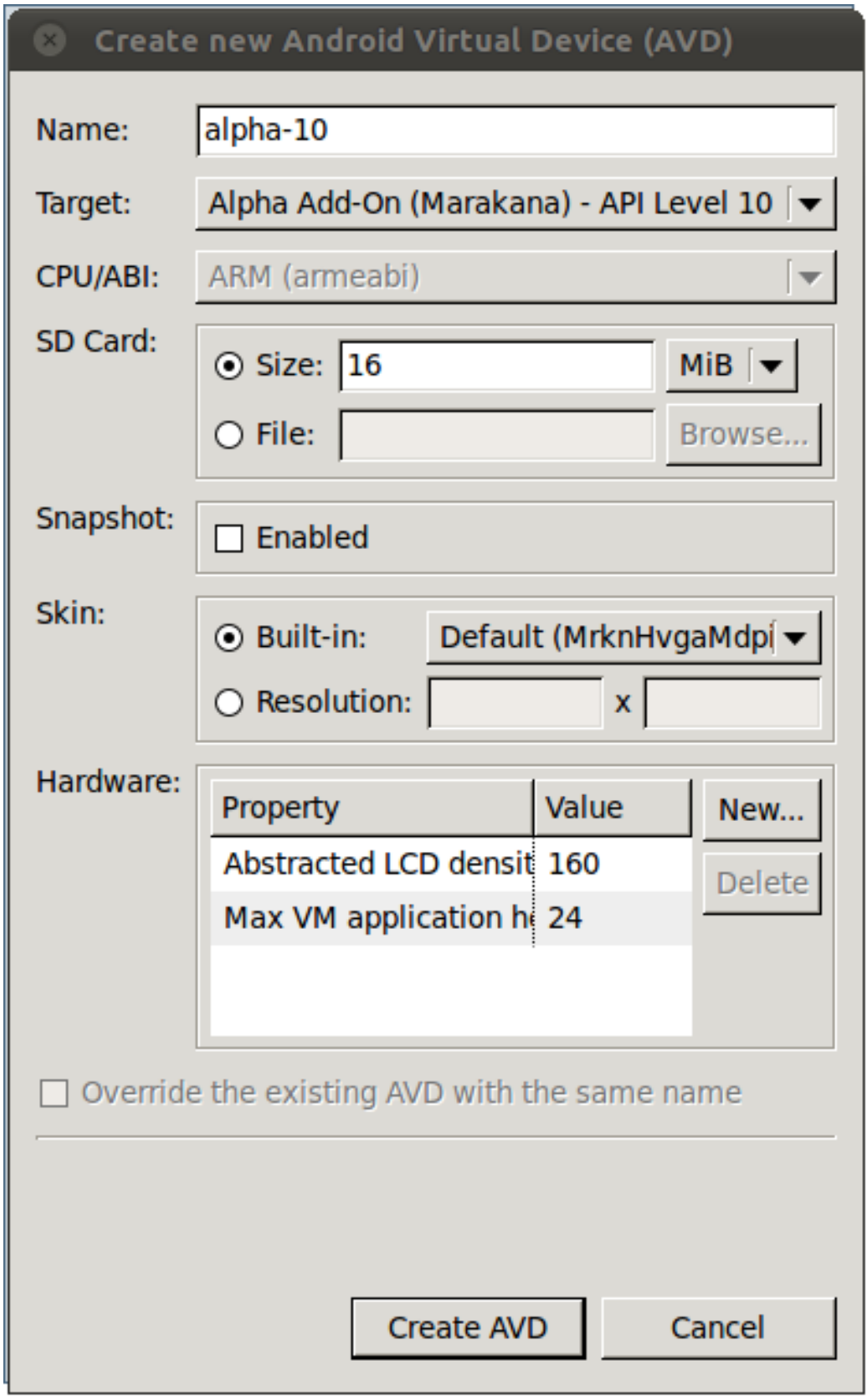
Note
Adjust the path to Android SDK directory as necessary.

    b. And we should now see "Alpha Add-On by Marakana, Android API 10, revision 1" show up under *Installed packages*:



Note
For this to work, we need to have our addon's `api` setting (in `manifest.ini` file) match an available *SDK Platform Android* of the same API version.

13. Now we can create a new emulator image (AVD) based on our add-on (from the *Android SDK and AVD Manager*):


Create new Android Virtual Device (AVD)
Name:
alpha-10
Target:
Alpha Add-On (Marakana) - API Level 10
CPU/ABI:
ARM (armeabi)
SD Card:
Size:
16
MiB
File:
Browse...
Snapshot:
Enabled
Skin:
Built-in:
Default (MrknHvgaMdpi
Resolution:
x
Hardware:
Property
Value
New...
Abstracted LCD densit
160
Delete
Max VM application h
24
Override the existing AVD with the same name
Create AVD
Cancel

14. To see that it works, we can now launch our newly created "alpha-10" AVD:

15. Our `alpha-10` AVD will not have any of the *Mrkn*ClientApp* apps we previously created for the "Marakana Alpha" device , but since our APIs are available to us via the libraries, we can re-create these applications in Eclipse and deploy them to our AVD - this is left as an exercise for the reader :-)