The Google Android Stack

Content

- **Open Handset Alliance [05.11.07]**
- **Android Mobile Phone Software Stack [12.11.07]**
- **Android Application Model**
- **Dalvik**

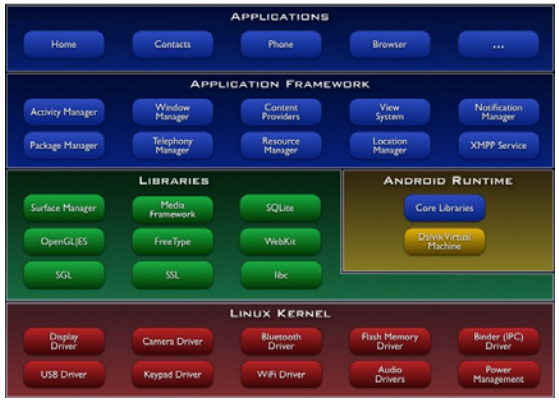
Dominik Gruntz & Carlo Nicola, IMVS
dominik.gruntz@fhnw.ch
carlo.nicola@fhnw.ch

Open Handset Alliance

- **Operators**
  - T-Mobile, Telecom Italia, Telefonica, NTT DoCoMo, KDDI, Sprint, China Mobile
- **Handset Manufacturers**
  - HTC, LG, Motorola, Samsung
- **Semiconductor Companies**
  - Audience, Broadcom, Intel, Marvell, NVIDIA, Qualcomm, SiRF, Synaptics, Texas Instruments
- **Software Companies [Applications]**
  - Ascender, eBay, **Esmertec**, Google, LivingImage, NMS Communications, Nuance Communications, PacketVideo, SkyPop, SONiVOX,
- **Commercialization Companies**
  - Aplix, TAT, Wind River, **Noser**

Android Software Stack (1/5)

- **Java**
- **C/C++**
- **Kernel**



Android Software Stack (2/5)

- **Kernel**
  - Linux 2.6 Kernel
  - Hardware Abstraction Layer
    - Proven driver model
    - Provides Memory Management / Process Management / Networking
    - Security Model
  - Unix C APIs are available

Android Software Stack (3/5)

- **Open-Source Libraries [C/C++]**
  - Surface Manager — composing windows on the screen
  - SGL — 2D Graphics
  - OpenGL|ES — 3D Library (HW acceleration possible)
  - Media FW (PacketVideo) — Audio/Video Codecs (mp3, AAC, mpg4, …)
  - FreeType — Font rendering
  - WebKit — Browser Engine
  - libc (System C libraries)
  - SQLite
  - OpenSSL

Android Software Stack (4/5)

- **Runtime**
  - Dalvik VM [Dan Bornstein]
    - Register-based VM
    - Optimized for low memory requirements
    - Designed to allow multiple VM instances to run at once
      - Every program runs in its own process / own VM instance
    - Relies on the underlying OS for process isolation, memory management and threading support
    - Operates on DEX files
  - Core Libraries

Android Software Stack (5/5)

- **Application Layer**
  - Browser, Contacts, Maps, …
- **Application Framework**
  - All Applications are equal
  - Applications are written in Java
  - Android Application Model: Activities & Intents

Android SDK

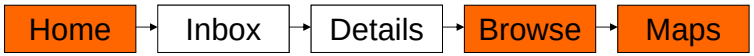
- **SDK [Windows / Mac OS X intel / Linux i386]**
  - http://code.google.com/android/
- **Content**
  - android.jar — android application framworkd (classes only)
  - docs — 100MB documentation
  - samples — 5 sample apps (+ skeleton app)
  - tools — various tools like adb (android debugger), dx, …
- **Demo**

Android Applications

- **Activity** **[Midlet]**
  - UI component typically corresponding to one screen
    - E.g. Contacts: 3 activities: View contacts, Send message, Edit contact
- **Intent Receiver** **[Push Registry]**
  - Responds to external events, can wake up your process
    - Phone rings, network activity established, time controlled
- **Service**
  - Task that runs in the background (e.g. mp3 player) without UI
  - Messages can be sent to a service
- **Content Provider**
  - Enables applications to share data
    - E.g. Contacts are provided to all applications

Android Applications: Life Cycle

- **Android Applications (Activities)**
  - Runs in its own process
  - Processes are started & stopped as needed
  - Processes may be killed to reclaim resources
  - Upon Invocation of another activity, the view state is saved (icicle)



  - Comparable with EJBs stateful session beans (SFSB)

Android Applications: Intents as Components

- **Intents**
  - Allows to invoke other activities from own activity

```
startActivity(new Intent(Intent.VIEW_ACTION,
   new ContentURI("http://www.fhnw.ch"));

startActivity(new Intent(Intent.VIEW_ACTION,
   new ContentURI("geo:47.480843,8.211293"));
```

  - Own Intent Handlers can be registered

=> Component Model on Activity Level

Android Applications: Content Providers

- **Android Application**
  - Every Android Application has access to its own database
  - Data is provided to other applications via content providers

```
abstract class ContentProvider {
   public Cursor query(ContentURI uri,
        String[] projection, String selection,
        String[] selectionArgs, String groupBy,
        String having, String sortOrder);
   public ContentURI insert(ContentURI uri,
        ContentValues values)
   public int delete(ContentURI uri, String selection,
        String[] selectionArgs);
   public int update(ContentURI uri,
        ContentValues values, String selection,
        String[] selectionArgs)
}
```

Android Applications: Content Providers

- **Access to Data Providers**
  - Comparable to a DB Access (Cursor)
  - Identification via URI
    - content://contacts/phones

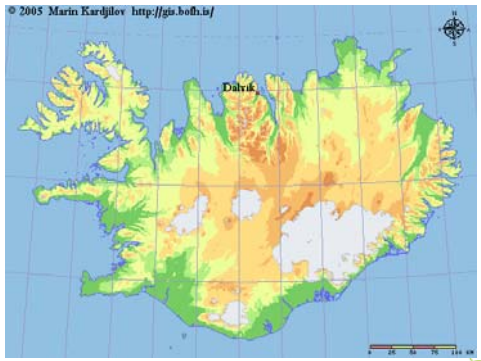
```
String[] projections = new String[]{
  "number", "name", "_id", "photo"};

Cursor cur = managedQuery(
   new ContentURI("content://contacts/phones"),
   projections, null, "name ASC");
```


Android Applications: Other Aspects

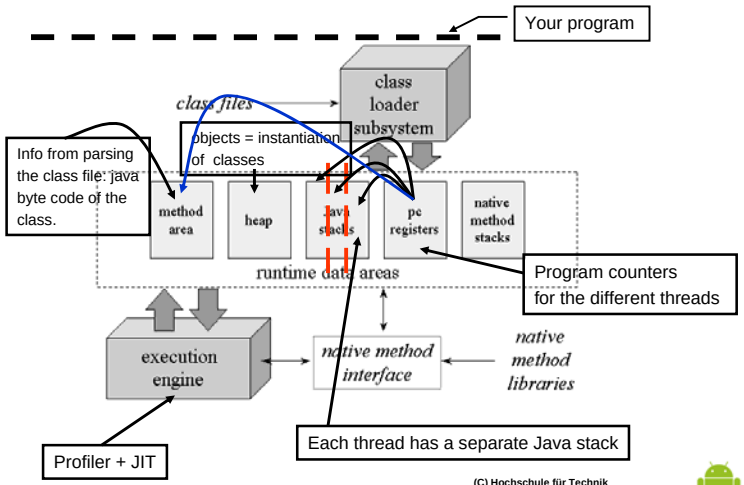
- **Security Model**
  - Each application runs in its own process and has a unique Linux User ID
    - Created files have same User ID
  - Permissions for specific operations
    - Declared in Manifest and checked at installation time
- **User Interface**
  - Declarative Layouts (XUL)
- **APIs**
  - Location API
  - Notification Manager
  - XMPP Service allows to send device-to-device data to other android users (multiplyer game, geo information, …)

Dalvik: Android Under the Hood

- **Dalvik**
  - Small fishing village in Eyjafjörður in Iceland
  - Home of one of the ancestor of Dan Bornstein



The complexity of the Java VM "down under"



Google's 10 Design Principles

1. Focus on the user and all else will follow.
2. It's best to do one thing really, really well.
3. Fast is better than slow.
4. Democracy on the web works.
5. You don't need to be at your desk to need an answer.
6. You can make money without doing evil.
7. There's always more information out there.
8. The need for information crosses all borders.
9. You can be serious without a suit.
10. Great just isn't good enough.

Amdahl's law: Make the common case fast

How the Dalvik VM manages this complexity

1. Run transparently normal Java `.class/.jar` files.
2. Use runtime memory very efficiently.
3. Run code with an highly cpu-optimized interpreter.
4. Support multiple virtual machine processes per device.
5. Provide class/namespace hiding mechanism for multiple JSR/spec compliance.
6. Seamless integration with an underlying system that includes a large amount of native code.

Some other less essential characteristics

- Include low-effort/lightweight JIT.
- Host other dynamic languages as efficiently as Java:
  - without altering the semantic of original language and without adding a runtime "impedance-matching" layer
  - The likeliest candidates are: JavaScript, Python
- Do exact garbage collection of thread stacks.

=> These goals are as yet not fully implemented.

Interpreter's Efficiency and Performance

1. New file format (`.dex`) is designed for runtime efficiency:
   a) shared constant pool across classes (reduces memory)
   b) maximize read-only structures (improves the sharing in cross-address-space)
   c) structures made as concise and fixed-width as possible (reduces parse time)
2. Byte codes are engineered for speed of interpretation.
   a) register-based instructions instead of stack-based (higher semantic density, reduced memory I/O overhead, reduced instruction dispatch overhead)
   b) only fixed-width instructions (simplifies parsing)
   c) 16 bit code units (minimizes instruction stream reads)

How Android runs Java

- **Working hypothesis about the most likely environment on which Android runs:**
  - Normally all preinstalled apps come with a .dex file.
  - Subsequently-encountered code gets translated on-the-fly.
  - Really new Java code is a relative rarity.
  - Not so fast flash memory is abundant.

  so it make sense to:
  - cache the translated .dex files liberally for both user-installed apps and transiently-used applets (or similar objects).

How to play efficiently with native code

1. **Provide efficient and lightweight native linkage**
   a) Goal: zero-overhead native method calling
   b) Implication: Java and native code in the same thread use a shared stack
2. **Use underlying OS facilities wherever possible**
   a) native threads and mutexes
   b) isolation through process separation
3. **Share heap with native non-Java-oriented code**
   - native graphics, UI, etc.
4. **Self-host to a reasonable extent**
   - helps minimize susceptibility to security flaws
5. **JIT is simplified because of the register-based instructions**

Verification

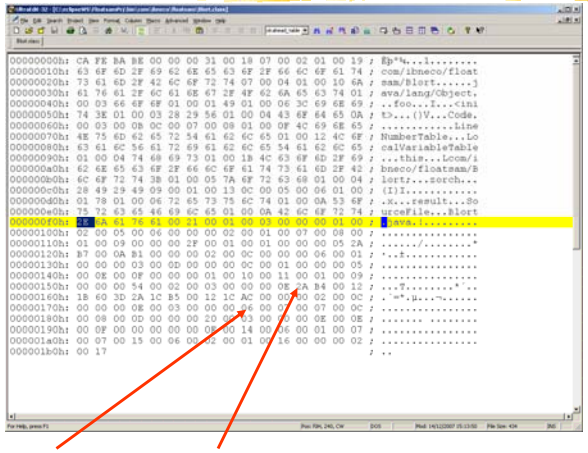
- **There are two possible places for verification:**
  - When loading Java **`.class`** files,
  - When loading **`.dex`** files.
- **There are good reasons to verify in both places:**
  - Verification of **`.class`** files is part of the Java security spec;
  - Verification of **`.dex`** file checks the soundness of the **`.class!.dex`** file's translation.

`.dex` verification is relatively easier. On security grounds it is important that both verifiers deliver consistently the same results.

Example: Blort.java

```
public class Blort {
   private int foo;
   public int zorch(int x) {
      int result = foo + x;
      foo = result;
      return result;
   }
}
```


Example: Blort.class



End zorch(int x): 14 bytes

Start zorch(int x)

Java Bytecode after javac

```
max_stack:    0002
max_locals:   0003
0000: 2a              aload_0
0001: b4 00 02        getfield Blort.foo:I
0004: 1b              iload_1
0005: 60              iadd
0006: 3d              istore_2
0007: 2a              aload_0
0008: 1c              iload_2
0009: b5 00 02        putfield Blort.foo:I
000c: 1c              iload_2
000d: ac              ireturn
```

Load a reference from local variable index 0 (= this)

Add 2 integer on the operand stack

Byte

private int foo;

All the operands are on the stack

Java Bytecode after .dex conversion

args in virtual CPU registers to be mapped into real ones

```
max_locals: 3
arg_count:  2
0000: 1070 0002   iget v0, v1, Blort.foo:I
0004: 20e0        add_int/2addr v0, v2
0006: 1077 0002   iput v0, v1, Blort.foo:I
000a: 0018        return v0
```

16 bit alignement

16 bit format for all instruction

Android Hardware

- **Chipset**: MSM7200A with RTR6285 and PM7540, based on ARM11 core
- **Memory**: at least 128MB RAM, 256MB Flash
- **External Storage**: Mini, Micro SD
- **Primary Display**: QVGA TFT LCD or larger, 16-bit color or better
- **Navigation Keys**: 5-way navigation with 4 application keys, power and volume control
- **Camera**: 2 Mega Pixel CMOS
- **USB**: Standard mini-B USB interface
- **Bluetooth** 1.2 or 2.0

Some characteristics of the ARM11 core

- **Characteristics**
  - Load/store architecture
  - No support for misaligned memory accesses
  - 16 × 32-bit register file
  - Fixed instruction width of 32 bits to ease decoding and pipelining
  - Mostly single-cycle execution
- **4-bit Condition Code is very interesting for Dalvik**
  - 4-bit *condition code* on the front of every instruction, meaning that an instruction is executed only if the correspondent flag is set:

```
loop CMP Ri, Rj     ; set condition "NE" if (i != j)
                    ; "GT" if (i > j),
                    ; or "LT" if (i < j)
  SUBGT Ri, Ri, Rj  ; if "GT", i = i-j;
  SUBLE Rj, Rj, Ri  ; if "LT", j = j-i;
  BNE loop          ; if "NE", then loop
```


How Dalvik uses the underlying HW: Thumb

- **Alternate instruction set supported by many ARM processors:**
  - 16-bit fixed size instructions
  - Only 8 registers easily available ! Saves 2 bits in instr. format
  - Registers are still 32 bits wide
  - Drops 3rd operand from data operations ! saves 5 bits
  - Only branches are conditional ! Saves 4 bits
  - Drops barrel shifter ! Saves 7 bits
- **Why use Thumb?**
  - 30% higher code density: less memory for the same application
  - Potentially higher performance on systems with 16-bit memory bus

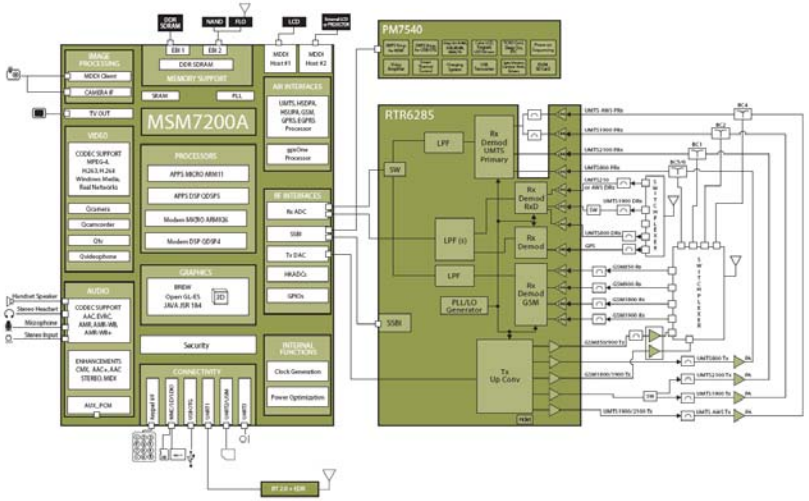
How Dalvik uses the underlying HW: Registers

ARM registers r0-r3 hold the first 4 arguments to a method
r9 is given special treatment in some situations, but not for Android
r10 (sl) can be freely used
r11 (fp) is used by gcc
r12 (ip = instruction pointer) is scratch -- not preserved across method calls
r13 (sp) should be managed carefully in case a signal arrives
r14 (lr) must be preserved
r15 (pc) can be tinkered with directly
r0 holds returns · 4 bytes
r0-r1 hold returns of 8 bytes, low word in r0

How Dalvik uses the underlying HW: Stack and FP

- **Stack is "full descending“**
  - Only the arguments that don't fit in the first 4 registers are placed on the stack. "sp" points at the first stacked argument (i.e. the 5th argument).
- **VFP**
  - single-precision results in s0, double-precision results in d0
- **In the EABI ARM version, "sp = stack pointer" must be 64-bit aligned on entry to a function, and any 64-bit quantities (long long, double) must be 64-bit aligned.**
- **This means we have to scan the Java method signature, identify arguments that must be padded, and fix them up appropriately.**

Not for softies (girls or boys)



Summary

- **Android**
  - Interesting new Programming Model
  - Full Control of the device allows to think at new applications
    - Do you have ideas for research projects? => contact IMVS
    - Android Developer Challenge: 10 Mio $