Dalvik VM

Reinhard Penn, Sebastian Ratzenbck

Embedded Systems Design, FH Obersterreich
Campus Hagenberg
[1] S1410567017@students.fh-hagenberg.at
[3] S1419002063@students.fh-hagenberg.at

Fig. 1. Android Logo[1]

***Abstract*—This paper is about the Dalvik Virtual Machine. The goals are to give a short overview on the Dalvik VM and its function. Furthermore the *.dex* file format will be explained. In the end a comparison between the Dalvik VM and the newer Android Runtime will take place.**

I. Allgemeines[2]

Bei Dalvik handelt es sich um eine Open Source Software die von Dan Bornstein entwickelt wurde. Benannt ist sie nach einem Fischerdorf in Eyjafjrur, Island. Verffentlicht wurde die Software unter Apache License 2.0. Die Ausfhrung findet auf einem Linux Kernel statt.

Verwendung findet Dalvik in Googles Betriebssystem Android. Einsatz findet Dalvik hierbei als virtuelle Maschine. Der Hauptverwendungsbereich liegt im Mobilbereich, wie zum Beispiel bei Smartphones, Tablets und seit neuestem auch bei Smart Tvs und Wearables, wie Smartwatches.

II. Architektur[3]

In diesem Kapitel wird die Architektur der Dalvik Virtual Machine erklrt. Besonderes Augenmerk wird hierbei auf den Aufbau der *.dex* Datei gelegt.

A. Funktion

In Abbildung 2 ist die allgemeine Architektur von Android zu sehen. In ihr sieht man, dass die Dalvik Virtual Machine Teil der Android Runtime ist. In Android bekommt jeder Prozess seine eigene virtuelle Maschine, bei dieser handelt es sich um eine Dalvik VM. Sie hnelt teilweise einer Java VM. Ein bedeutender Unterschied ist allerdings, dass eine Java VM stapelbasiert und eine Dalvik VM registerbasiert arbeitet. Diese registerbasierte Arbeitsweise lehnt sich an moderne Prozessorarchitekturen an. Sie verarbeitet Registermaschinencode, dadurch wird die Dalivk VM schneller als die Java VM und ist ressourcenschonender.

Ein weiterer bedeutender Unterschied liegt darin, dass eine Dalvik VM klassische Java Bibliotheken nicht untersttzt, zum Beispiel *AWT* und *Swing*. Es werden eigene Bibliotheken verwendet, die Apache Harmony als Grundlage verwenden.

Ein wichtiger Bestandteil der *SDK* ist das Tool *dx*. Es sorgt dafr, dass Java Binrdaten in Dalvik Executables umgewandelt werden. Das heit es wandelt *.class* Dateien in *.dex* Dateien um. Bei dieser Umwandlung knnen auch mehrere *.class* Dateien zu einer *.dex* Datei zusammengefasst werden, beziehungsweise kann der Speicherbedarf, mithilfe von *.odex* Dateien optimiert werden.

In Abbildung 3 ist ein solcher Vorgang beispielhaft dargestellt. Zuerst werden die *.java* Dateien in *.class* Dateien kompiliert. Nach dem Kompiliervorgang werden die Binrdateien und die *.class* Dateien mithilfe des *dx* Tools zu eienr *.dex* Datei konvertiert. Mithilfe dieser Datei kann nun eine *.apk* Datei erstellt werden. Bei dieser handelt es sich um die Applikationsdatei mit der, der Endbenutzer die Anwendung auf seinem Mobiltelefon installiert.

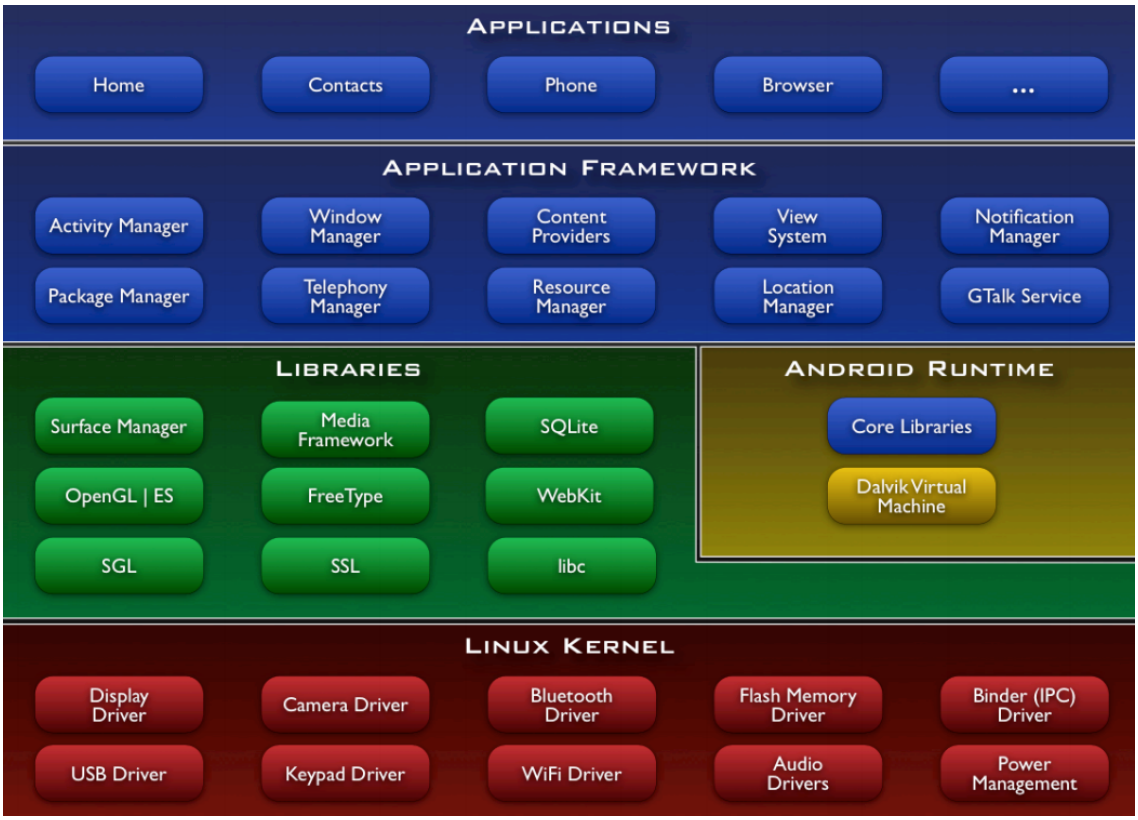

Fig. 2. Android Architektur[4]

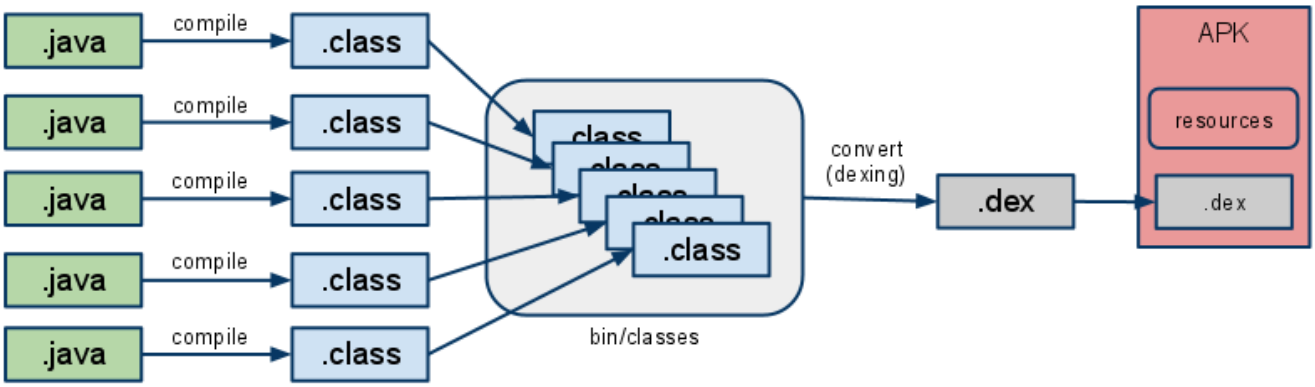

Fig. 3. Android Kompiliervorgang[5]

B. .dex Format

In .dex Dateien werden die Klassen Definitionen und die dazugehrigen Daten gespeichert. Es handelt sich dabei um die ausfhrbaren Dateien der Dalvik VM. Android Programme werden zuerst in Java Bytecode kompiliert. Dieser wird im Anschluss in Dalvik Bytecode bersetzt.

In Abbildung 4 ist der Aufbau einer .dex Datei im Vergleich zu einer .class Datei von Java zu sehen. Fr alle folgenden Listen der Datei drfen keine doppelten Eintrge vorhanden sein.

1) Header: In Ausschnitt 1 ist der Header der *.dex* Datei zu sehen. Dabei handelt es sich um eine bestimmte Bytefolge die den Start der Datei bestimmt.

2) StringIds: Hierbei handelt es sich um eine Liste aller Strings die in der Datei verwendet werden. Die Liste muss sortiert sein und darf keine doppelten Eintrge

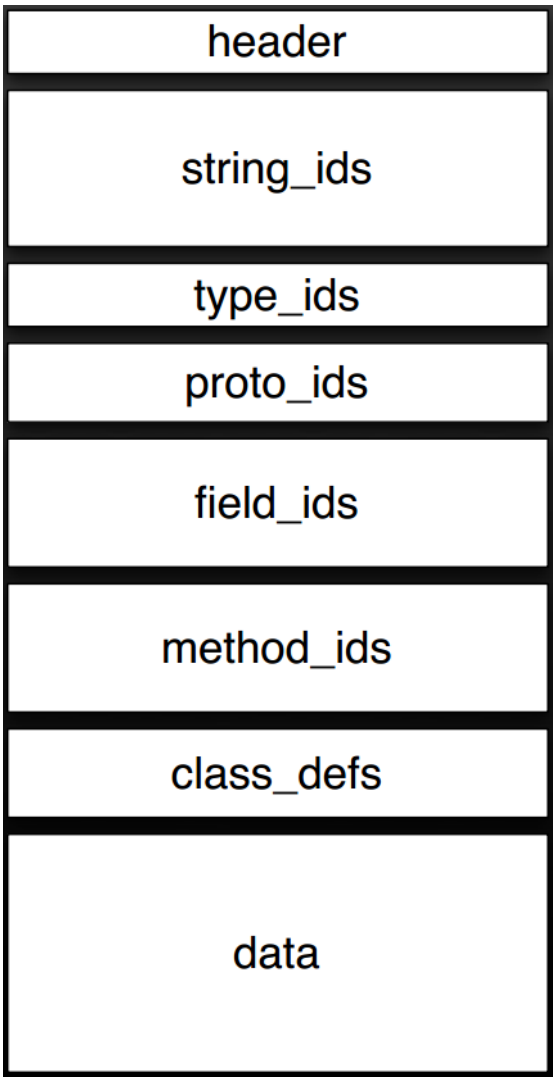

Fig. 4. Dex Dateiformat[4]

```
ubyte[8] DEX_FILE_MAGIC =
{ 0x64 0x65 0x78 0x0a 0x30 0x33 0x35 0x00 }
= "dex\n035\0"
```

Ausschnitt 1. Dex Header[3]

enthalten. Mgliche Inhalte sind zum Beispiel konstante Strings und Funktionsnamen.

3) TypeIds: Dies ist eine Liste aller verwendeten Typen dieser Datei. Dazu gehren Klassen, Arrays, und Primitive Datentypen. Diese Liste ist nach den *StringIds* sortiert.

4) ProtoIds: In dieser Liste befinden sich alle Methoden Prototypen, die in *.dex* Datei verwendet werden. Die Prototypen sind primr anhand ihrer Rckgabetypen sortiert und danach anhand ihrer Argumente.

5) FieldIds: Diese Liste enthlt alle Felder die von der *.dex* Datei referenziert werden. Diese Liste wird anhand des Feldtypen und Feldnamen sortiert.

6) MethodIds: In der Methoden Liste werden alle von der *.dex* Datei referenzierten Methoden aufgelistet. Diese Liste wird nach dem Methodennamen und dem Methodenprototypen sortiert.

7) ClassDefs: In dieser Liste werden die Klassendefinitionen der *.dex* Datei angegeben. Die Liste muss so geordnet werden, dass bergeordnete Klassen und Interfaces vor den davon ableitenden Klassen angefhrt werden.

8) Data: In dem Data werden zustzliche Daten fr die oben angefhrten Listen gespeichert. Die verschiedenen Elemente haben verschiedene Alignements und padding bytes werden, wenn notwendig, eingefgt. Des weiteren befinden sich in diesem Bereich die Daten von statisch verlinkten Dateien. Bei nicht verlinkten Dateien ist dieser Teilbereich leer.

C. Bytecode & Instruction Format

Das Modell der Dalvik VM ist an eine echte Architektur angelegt und soll Calling Conventions C hnlich realisieren. Die Dalvik VM ist registerbasiert und ihre Frames bekommen bei der Erstellung eine fixe Gre zugewiesen. Jedes Frame besteht aus einer bestimmten Anzahl an Registern, die von einer Methode aus dem Bytecode Set vorgegeben wird. Zustzlich beinhaltet das Frame noch alle zustzlich bentigten Daten, so wie zum Beispiel Program Counter und die *.dex* Datei die die Methode beinhaltet. Fr Bitwerte, wie zum Beispiel *integer* und Gleitkommazahlen sind Register 32-Bit lang. Fr 64-Bit Werte wird mit einem anliegenden Register ein Paar gebildet. Fr Registerpaare wird kein Alignement bentigt. Wenn die Register fr Objekt Referenzen benutzt werden, dann sind sie gro genug um exakt eine Referenz zu speichern. Bei einem Methodenaufruf werden N Argumente der Methode in den N letzten Registern des Frames der Methode platziert. Grere Argumente bentigen hierbei 2 Register. Methoden einer Instanz bekommen als ersten Parameter zustzlich eine *this* Referenz bergeben.

Instruktionen sind nicht auf Datentypen limitiert, solange sie den Inhalt der Register nicht interpretieren. Ein Beispiel dafr ist der *move* Befehl, ihm ist es egal ob er *integer* oder *floats* verschiebt. Die meisten Instruktionen sind bei ihrer Ausfhrung auf die ersten 16 Register limitiert und knnen auf hhere Register nicht zugreifen. Wenn es mglich ist erlauben einige

Instruktionen, aber auch Zugriff auf die ersten 256 Register. In einigen Ausnahmefllen wie zum Beispiel einzelnen *move* Instruktionen ist es mglich auf Register zwischen 0 und 65535 zuzugreifen. Untersttzt eine Instruktion das gewnschte Register nicht muss das hohe Register vor der Operation in ein niederes Register kopiert und nach der Operation wieder zurck kopiert werden.

Die Dalvik VM besitzt einige Pseudoinstruktionen, die dazu verwendet werden um Daten mit variabler Lnge zu verarbeiten zum Beispiel *fill-array-data*. Diese Instruktionen mssen auf 4 Byte aligned sein. Um dieses Ziel zu erreichen werden bei der Generierung der *.dex* Datei wenn ntig *nop* Instruktionen eingefgt. Wenn die Applikation auf einem System installiert wird knnen manche Instruktionen, aufgrund von Optimierungen, verndert werden. Damit kann eine schnellere Ausfhrung am System erreicht werden.

In Abbildung 5 sind einige Beispielinstruktionen der Dalvik VM zu sehen. Die Syntax ist so aufgebaut das zuerst das Ziel- und danach das Quellregister angegeben wird. Manche Opcodes haben Namenserweiterungen um gewisse Eigenschaften anzuzeigen. Zum Beispiel wird bei 64 Bit Opcodes *-wide* hinzugefgt und bei typenspezifischen Opcodes wird der Typ hinzugefgt, *-char*. Anhand des *move-wide/from16 vAA, vBBBB* Opcodes lassen sich diese Eigenschaften gut erkennen.

Op & Format	Mnemonic / Syntax	Arguments	Description
00 10x	nop		Waste cycles. **Note:** Data-bearing pseudo-instructions are tagged with this opcode, in which case the high-order byte of the opcode unit indicates the nature of the data. See "`packed-switch-payload` Format", "`sparse-switch-payload` Format", and "`fill-array-data-payload` Format" below.
01 12x	move vA, vB	`A:` destination register (4 bits) `B:` source register (4 bits)	Move the contents of one non-object register to another.
02 22x	move/from16 vAA, vBBBB	`A:` destination register (8 bits) `B:` source register (16 bits)	Move the contents of one non-object register to another.
03 32x	move/16 vAAAA, vBBBB	`A:` destination register (16 bits) `B:` source register (16 bits)	Move the contents of one non-object register to another.
04 12x	move-wide vA, vB	`A:` destination register pair (4 bits) `B:` source register pair (4 bits)	Move the contents of one register-pair to another. **Note:** It is legal to move from `vN` to either `vN-1` or `vN+1`, so implementations must arrange for both halves of a register pair to be read before anything is written.
05 22x	move-wide/from16 vAA, vBBBB	`A:` destination register pair (8 bits) `B:` source register pair (16 bits)	Move the contents of one register-pair to another. **Note:** Implementation considerations are the same as `move-wide`, above.
06 32x	move-wide/16 vAAAA, vBBBB	`A:` destination register pair (16 bits) `B:` source register pair (16 bits)	Move the contents of one register-pair to another. **Note:** Implementation considerations are the same as `move-wide`, above.
07 12x	move-object vA, vB	`A:` destination register (4 bits) `B:` source register (4 bits)	Move the contents of one object-bearing register to another.
08 22x	move-object/from16 vAA, vBBBB	`A:` destination register (8 bits) `B:` source register (16 bits)	Move the contents of one object-bearing register to another.
09 32x	move-object/16 vAAAA, vBBBB	`A:` destination register (16 bits) `B:` source register (16 bits)	Move the contents of one object-bearing register to another.

Fig. 5. Dalvik VM Instruktionen[3]

III. Android Runtime

A. Allgemein

B. Vergleich zu Dalvik VM

References

[1] Android logo. [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/thumb/d/d7/Android_robot.svg/872px-Android_robot.svg.png

[2] Dalvik (software). [Online]. Available: http://en.wikipedia.org/wiki/Dalvik_(software)

[3] Art and dalvik. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[4] D. Bornstein. Dalvik vm internals. [Online]. Available: https://14b1424d-a-62cb3a1a-s-sites.googlegroups.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf

[5] Android kompiliervorgang. [Online]. Available: https://devmaze.files.wordpress.com/2011/05/image10.png