Auszug aus dem Handbuch der Chipkarten in der 3. deutschen Auflage von Wolfgang Rankl und Wolfgang Effing

Der folgende Artikel ist eine komprimierte Version des Kapitels über Chipkarten-Betriebssysteme aus der dritten Auflage des Handbuchs der Chipkarten von Wolfgang Rankl und Wolfgang Effing. Es erscheint im Carl Hanser Verlag, München im März 1999 in einer stark erweiterten und vollständig aktualisierten dritten deutschen Auflage.
Eine englische Ausgabe im Verlag John Wiley & Sons ist in Planung.

Chipkarten-Betriebssysteme

Der Begriff Betriebssystem ist nicht automatisch auf riesige Programme und Datenmengen beschränkt, sondern er ist völlig unabhängig von Größen, da er ausschließlich die Funktionalität definiert. Es ist wichtig, daß man mit dem Begriff Betriebssystem nicht automatisch die mehrere Megabyte großen Programme für PCs oder Unix Computer assoziiert. Diese sind genauso spezifisch auf eine Mensch-Maschine-Schnittstelle über Monitor, Tastatur und Maus hin ausgelegt, wie die Chipkarten-Betriebssysteme auf die bidirektionale, serielle Schnittstelle zum Terminal hin gestaltet sind.

Letztendlich ist für ein Betriebssystem nur die Funktionalität entscheidend, die sich aus dem Zusammenwirken von zueinander passenden und aufeinander aufbauenden Bibliotheksroutinen ergibt. Wichtig ist in diesem Zusammenhang auch, daß ein Betriebssystem eine Schnittstelle zwischen der Hardware des Rechners und der eigentlichen Anwendungssoftware ist. Dies hat für die Anwendungssoftware auch den großen Vorteil, daß diese nicht direkt auf die Hardware zugreifen muß und so eine, wenn auch oft sehr begrenzte, Portabilität erhalten bleibt.

Im Laufe der Entwicklung hat sich weltweit die Bezeichnung COS (*card operating system*) für Chipkarten-Betriebssysteme eingebürgert. Diese ist häufig als Teil des Produktnamens (z.B.: STARCOS, MPCOS) zu finden. Mittlerweile gibt es weltweit über ein Dutzend Hersteller von allgemeinen und anwendungsunabhängigen Chipkarten-Betriebssystemen.

1 Grundlagen

Die Betriebssysteme für Chipkarten weisen im Gegensatz zu den allgemein bekannten Betriebssystemen keine Benutzeroberfläche und keine Zugriffsmöglichkeiten auf externe Speichermedien auf, da sie auf eine ganz andere Funktionalität hin optimiert sind. Die Sicherheit bei der Ausführung von Programmen und der geschützte Zugriff auf Daten haben dabei die oberste Priorität. Sie haben aufgrund der Einschränkungen durch den zur Verfügung stehenden Speicherplatz einen sehr kleinen Codeumfang, der im Bereich zwischen 3 und 30 kByte liegt. Die untere Grenze steht dabei für Spezialanwendungen und die obere für Multiapplication-Betriebssysteme. Der durchschnittliche Speicherbedarf liegt aber meist im Bereich um 16 kByte.

Die Programmodule sind als ROM-Code geschrieben, was dazu führt, daß die Methoden der Programmierung sehr eingeschränkt sind, da viele bei RAM-Programmcode übliche Abläufe (z.B. selbstmodifizierender Code) nicht möglich sind. Der ROM-Code ist auch der Grund dafür, daß nach der Programmierung und Herstellung des ROMs auf dem Mikrocontroller keinerlei Änderungen mehr vorgenommen werden können. Die Beseitigung eines Fehlers ist dadurch extrem teuer und wegen der Halbleiterherstellung mit einer Durchlaufzeit von 10 bis 12 Wochen verbunden. Ist die Chipkarte beim Endbenutzer angelangt, lassen sich Fehler nur mehr durch großangelegte Umtauschaktionen beseitigen, die den Ruf eines auf Chipkarten basierenden Systems ruinieren können. Eine „quick and dirty"-Programmierung verbietet sich deshalb von selbst. Der zeitliche Aufwand für Test

und Qualitätssicherung ist deshalb im Regelfall wesentlich höher als die Zeitdauer für die Programmierung.

Doch müssen diese Betriebssysteme neben der extremen Fehlerarmut auch sehr zuverlässig und robust sein. Sie dürfen durch kein von außen kommendes Kommando in ihrer Funktion und vor allem in ihrer Sicherheit beeinträchtigt werden. Systemzusammenbrüche oder unkontrollierte Reaktionen auf ein fehlerhaftes Kommando oder durch ausgefallene Seiten im EEPROM dürfen auf keinen Fall vorkommen.

Der Begriff „Sicherheitsbetriebssystem“ enthält auch noch einen anderen Aspekt. Falltüren und andere Hintereingänge für Systemprogrammierer, wie sie bei großen Systemen immer wieder vorkommen und sogar durchaus üblich sind, müssen bei Chipkarten-Betriebssystemen gänzlich ausgeschlossen sein. Es darf z.B. keine Möglichkeit geben, am Betriebssystem vorbei mit irgendeinem Mechanismus Daten unautorisiert auszulesen.

Nicht zu unterschätzen ist außerdem die erforderliche Leistungsfähigkeit. Die im Betriebssystem vorhandenen kryptografischen Funktionen müssen in sehr kurzer Zeit ablaufen. So ist es üblich, während der Entwicklung in wochenlanger Kleinarbeit die entsprechenden Algorithmen in Assembler zu optimieren. Es ist daher einleuchtend, daß Multitasking aufgrund der verwendeten Hardwareplattformen und der geforderten Zuverlässigkeit nicht verwendet werden kann. Die Beschränkung auf eine einzeln ablaufende Task verhindert aber leider auch den Einsatz von Schutzprozessen, die Teile des Betriebssystems in Ablauf und Randbedingungen überwachen.

Zusammenfassend hat ein Chipkarten-Betriebssystem folgende Hauptaufgaben:

- Datenübertragung von und zur Chipkarte
- Ablaufsteuerung der Kommandos
- Dateiverwaltung
- Verwaltung und Ausführung von kryptografischen Algorithmen

Kommandoabarbeitung

Die typische Kommandoabarbeitung innerhalb des Chipkarten-Betriebssystems, das keinen nachladbaren Programmcode unterstützt, läuft wie folgt ab: Alle Kommandos an die Chipkarte empfängt diese über die serielle I/O-Schnittstelle. Fehlererkennungs- und -korrekturmechanismen führt der I/O-Manager bei Bedarf völlig unabhängig von den übrigen, darauf aufbauenden Schichten aus. Nachdem ein Kommando vollständig und fehlerfrei empfangen wurde, muß der Secure Messaging Manager diesen gegebenenfalls entschlüsseln oder auf Integrität prüfen. Findet keine gesicherte Datenübertragung statt, ist dieser Manager sowohl für Kommando als auch Antwort völlig transparent.

Nach dieser Bearbeitung versucht die darüberliegende Schicht, der Kommandointerpreter, das Kommando zu decodieren. Ist dies nicht möglich, folgt ein Aufruf des Returncode-Managers, welcher einen entsprechenden Returncode generiert und via I/O-Manager an das Terminal zurücksendet. Es kann notwendig sein, den Returncode-Manager applikationsspezifisch zu gestalten, da die Returncodes nicht zwangsläufig für alle Anwendungen einheitlich sind. Konnte das Kommando jedoch decodiert werden, dann ermittelt der Logical Channel Manager den angewählten

Kanal, schaltet auf dessen Zustände um und ruft dann im Gutfall den Zustandsautomaten auf.

Dieser prüft nun, ob das Kommando an die Chipkarte mit den gesetzten Parametern im aktuellen Zustand überhaupt erlaubt ist. Ist das der Fall, dann wird der eigentliche Programmcode des Anwendungskommandos ausgeführt, welcher die Abarbeitung des Kommandos übernimmt. Falls das Kommando im aktuellen Zustand verboten ist oder die Parameter dazu nicht erlaubt sind, erhält das Terminal über Returncode-Manager und I/O-Manager eine entsprechende Meldung.

Ist es notwendig, während der Kommandobearbeitung auf eine Datei zuzugreifen, dann geschieht dies nur über die Dateiverwaltung, die alle logischen Adressen in physikalische Adressen des Chips umsetzt. Weiterhin überwacht sie die Adressen hinsichtlich der Bereichsgrenzen und prüft die Zugriffsbedingungen auf die jeweilige Datei.

Die Dateiverwaltung selber benutzt einen weiteren Speichermanager, der die komplette Verwaltung des physikalisch adressierten EEPROMs übernimmt. Damit ist sichergestellt, daß nur in diesem Programmodul mit echten physikalischen Adressen gearbeitet wird, was die Portabilität und Sicherheit des ganzen Betriebssystems erheblich steigert.

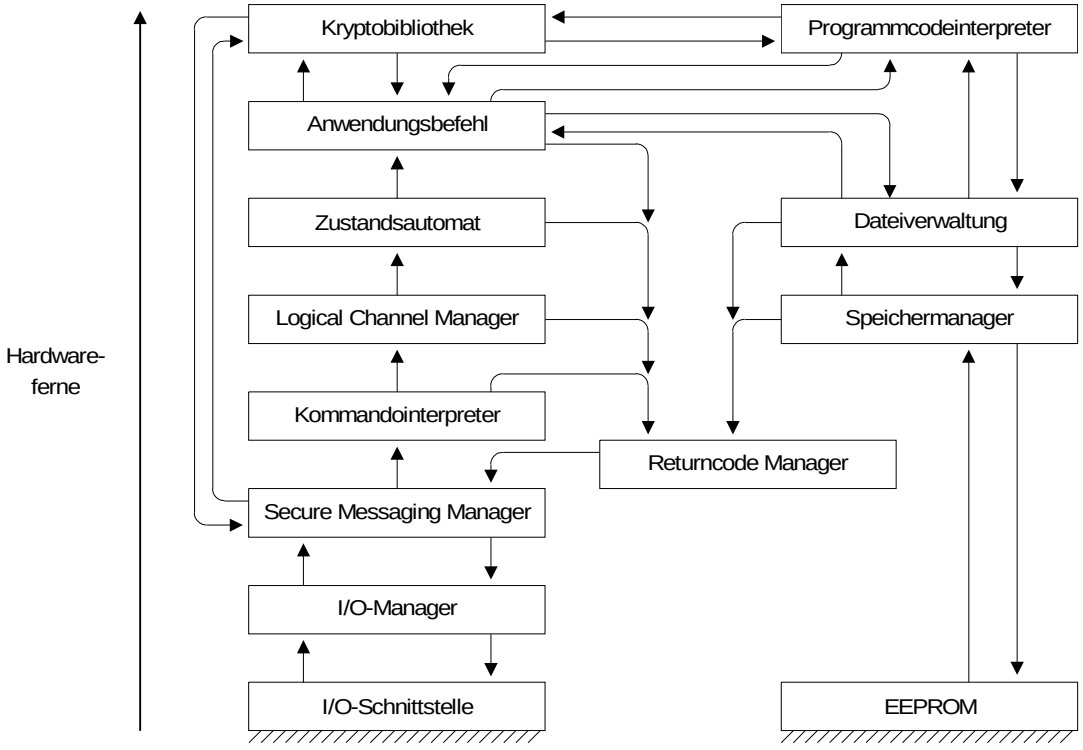

Bild 1 *Ablauf der Kommandoabarbeitung innerhalb eines Chipkarten-Betriebssystems. Der Programmcodeinterpreter ist optional und kann auch direkt vom Zustandsautomaten aus aufgerufen werden, wie dies beispielsweise bei Java geschieht.*

Die Erzeugung der Antwortcodes übernimmt ein zentraler Returncode-Manager, der für den aufrufenden Programmteil jeweils die komplette Antwort erzeugt. Diese Schicht übernimmt für alle anderen Teile des Betriebssystem die Verwaltung und Erzeugung aller verwendeten Returncodes.

Da Chipkarten-Betriebssysteme meist kryptografische Funktionen nutzen, ist in der Regel noch eine eigene und vom Rest des Betriebssystems abgetrennte Bibliothek mit kryptografischen Funktionen vorhanden. Diese Kryptobibliothek dient allen anderen Modulen als zentrale Anlaufstelle bei Benutzung von Kryptofunktionen.

Zusätzlich zu diesen Schichten kann im Bereich oberhalb der Anwendungskommandos noch ein Interpreter oder ein Prüfprogramm für ausführbare Dateien vorhanden sein. Dieser überwacht die in diesen Dateien enthaltenen Programme und führt sie aus oder interpretiert sie. Der genaue Aufbau und die Implementation hängt davon ab, ob überhaupt Dateien mit ausführbarem Code vorgesehen sind und ob Maschinencode für den Prozessor oder zu interpretierender Code abgespeichert ist.

2 Entwurfs- und Implementierungsprinzipien

Ein Chipkarten-Betriebssystem ist von seiner Aufgabenstellung her immer ein Sicherheitsbetriebssystem, das Informationen verwalten und vor allem geheimhalten muß. Ebenso sind in der Regel keinerlei Änderungen oder Updates der Software während des Betriebs mehr möglich. Damit ist das oberste Prinzip schon vorgegeben. Ein Chipkarten-Betriebssystem muß extrem zuverlässig und damit auch extrem fehlerarm sein. Eine totale Fehlerfreiheit läßt sich in der Realität aber nie erreichen, da selbst die kleinen Betriebssystemkerne der Chipkarten zu groß sind, um sie vollständig in allen Möglichkeiten des internen Programmablaufs zu prüfen.

Ein streng modularer Aufbau trägt aber entscheidend dazu bei, daß eventuelle Fehler während der Implementierungsphase entdeckt und beseitigt werden können. Diese Modularität, die die Zuverlässigkeit stark erhöht, muß nicht unbedingt mit einem großen Mehraufwand an Programmcode verbunden sein. Ein weiterer Vorteil der Modularität ist, daß sich mögliche Systemzusammenbrüche im allgemeinen nicht so stark auf die Sicherheit auswirken wie bei einem hochoptimierten und speichersparenden Programmcode. Das wiederum führt dazu, daß die Auswirkungen möglicher Fehler lokal bleiben und das Betriebssystem als ganzes robuster und stabiler wird.

Dadurch, daß die Implementation meistens in Assembler durchgeführt werden muß, erhöht sich die Fehleranfälligkeit. Der Aufbau aus einzeln vollständig austestbaren Modulen trägt durch die definierten Schnittstellen stark dazu bei, Programmierfehler rechtzeitig zu erkennen und einzugrenzen. Dies führt in der Konsequenz zu dem in Bild 1 dargestellten Schichtenaufbau des Betriebssystems. Der höhere Planungs- und Codierungsaufwand wird durch die erheblich einfacheren Tests und Prüfungen wirtschaftlich auf jeden Fall wettgemacht. Dies hat dazu geführt, daß mittlerweile fast alle Betriebssysteme die hier beschriebene oder zumindest eine dazu sehr ähnliche interne Struktur besitzen.

Die Softwareentwicklung von Betriebssystemen für Chipkarten bewegt sich von der kompletten Programmierung in Assembler weg. Viele neuere Projekte werden von Anfang an in der hardwarenahen Hochsprache C durchgeführt. Jedoch baut der eigentliche Kern der Betriebssysteme weiterhin auf maschinenabhängigen Assemblerroutinen auf, während alle übergeordneten Module, wie Dateiverwaltung, Zustandsautomat und Kommandointerpreter, in C programmiert sind. Dies senkt deutlich die Implementierungszeit, die Programme sind leichter zu portieren, wieder-

verwendbar und vor allem ist der Programmcode durch die Verwendung einer Hochsprache wesentlich besser prüfbar. Diese bessere und übersichtlichere Programmstruktur durch eine Hochsprache führt in Folge zu einer merklich geringeren Fehlerquote.

Leider benötigt der Programmcode, den selbst hochoptimierende C-Compiler generieren, für die gleiche Funktionalität, zwischen 20 % und 40 % mehr Speicherplatz im ROM als der Programmcode in Assembler geschrieben. Ferner ist die Performanz bei einer Implementierung in C geringfügig niedriger als bei einer Programmierung in Assembler. Heikel ist dies allerdings nur bei kryptografischen Algorithmen und den Übertragungsprotokollen, da alle anderen Programmteile in Chipkarten-Betriebssystemen in der Regel keine zeitkritischen Abläufe enthalten.

Um jedoch für diese Fälle eine höhere Nachvollziehbarkeit und Sicherheit zu erreichen, werden seit einiger Zeit bei Chipkarten-Betriebssystemen vermehrt Evaluierungen nach ITSEC bzw. dem Nachfolger Common Criteria angestrebt. Dies geschieht entweder freiwillig durch die Betriebssystem-Hersteller oder durch Forderung der größeren Anwendungsanbieter. Diese wollen durch eine Evaluierung des Chipkarten-Betriebssystems die Sicherheit erhöhen, daß keine signifikanten Fehler im Programmcode enthalten sind. Eine Prüfung auf absichtlich eingeschleuste trojanische Pferde würde wahrscheinlich auch durch eine Evaluierung nur bedingt entdeckt werden, da die Möglichkeiten dazu praktisch unbegrenzt sind.

Bisher übliche Evaluierungsstufen bei Chipkarten-Betriebssystemen sind E3 und E4 nach ITSEC. Vereinzelt wird sogar die Evaluierungsstufe E6 gefordert und auch angeboten. Dabei darf jedoch nicht übersehen werden, daß eine Evaluierung nach E4 für ein vollständiges Chipkarten-Betriebssystem im Bereich von einer halben Million Mark liegen kann. Hinzu kommt dann noch die Pflicht einer Neuevaluierung bei Änderungen im Programmcode, was freilich weniger aufwendig ist als die Erstevaluierung. Dies sind die wesentlichen Gründe, warum immer noch verhältnismäßig wenige Chipkarten-Betriebssysteme eine Evaluierung nach ITSEC haben. Öfter werden deshalb von Prüfinstituten Evaluierungen ohne einen Bezug zur ITSEC durchgeführt. Dabei beschränkt man sich auf eine gründliche Prüfung der Designkriterien, des Sourcecodes und der Dokumentation, was beispielsweise bei den Betriebssystemen für die deutschen ec-Karten grundsätzlich Pflicht ist.

3 Aufteilung des Programmcodes

Der Lebenszyklus eines Chipkarten-Betriebssystems ist in zwei Teile getrennt – den Teil vor und den Teil nach der Komplettierung. Im Abschnitt vor der Komplettierung, bei dem der Mikrocontroller aus der Halbleiterfertigung mit leerem EEPROM kommt, laufen alle Programmteile im ROM ab. Es werden weder Daten aus dem EEPROM gelesen, noch dort Programme ausgeführt. Stellt sich zu diesem Zeitpunkt heraus, daß im ROM-Code ein Fehler vorhanden ist, der die Komplettierung unmöglich macht, so muß die gesamte produzierte Charge der Mikrocontroller vernichtet werden, da für die Chips keine weitere Verwendung besteht.

Bei der Komplettierung werden die ROM-Teile für die eigentliche Anwendung angepaßt. Der ROM-Teil ist sozusagen eine große Bibliothek, die durch das EEPROM zu einer funktionsfähigen Anwendung verbunden und ausgebaut wird. Zusätzlich besteht bei fast allen Betriebssystemen noch die Möglichkeit, bei der

Komplettierung Programmcode für weitere Kommandos oder spezielle kryptografische Algorithmen in das EEPROM zu laden. Dies ist aber unabhängig von eventuell vorhandenen ausführbaren Dateien, da der Inhalt zu einem späteren Zeitpunkt, z.B. vom Personalisierer, geladen werden kann. Die während der Komplettierung ins EEPROM eingebrachten Programme sind vollständig in das Betriebssystem eingebunden und von diesem auch direkt zu benutzen.

Soft- und Hardmaske

Im Zusammenhang mit Feldversuchen und Chipkarten-Betriebssystemen werden oft die beiden Begriffe Softmaske (*softmask*) und Hardmaske (*hardmask*) benutzt. Genaugenommen und streng logisch gesehen, sind beide Begriffe unsinnig, da eine ROM-Maske – gemeint ist dabei der Programmcode der sich im ROM befindet – immer unveränderlich, also fest ist. Im Sprachgebrauch der Chipkartenwelt ist aber mit Softmaske nur so etwas ähnliches wie eine ROM-Maske gemeint. Man spricht von einer Softmaske, wenn sich Teile oder der gesamte Programmcode für ein Chipkarten-Betriebssystem oder die Kommandos einer Anwendung im EEPROM befinden. Der Code läßt sich auf Grund dieser Sache einfach ändern, ohne daß zeit- und kostenintensiv eine neue ROM-Maske erstellt werden muß. Diese Art von „Maske“ ist also „weich“ und veränderbar und insofern eine Softmaske. Vor allem bei Tests und Feldversuchen wird diese Technik benutzt, da sich kurzfristig und mit geringem Aufwand Fehlerbeseitigungen und Programmadaptionen durchführen lassen. Der Nachteil dabei ist jedoch, daß dazu Chips verwendet werden müssen, die ein großes EEPROM besitzen und deshalb teurer sind als äquivalente Chips mit dem Programmcode im ROM. Da bei Feldversuchen jedoch üblicherweise keine Millionenstückzahlen an Karten ausgegeben werden, sind die erhöhten Kosten für die Chips mit größerem EEPROM-Speicher durchaus vertretbar.

APIs von Betriebssystemen

Ursprünglich boten Chipkarten-Betriebssysteme keine Möglichkeit, selbsterstellten Programmcode zu laden und bei Bedarf auf der Chipkarte auszuführen. Deshalb besaßen die herkömmlichen Betriebssysteme keine veröffentlichte Programmierschnittstelle, die für Aufrufe von Betriebssystemfunktionen von Dritten genutzt werden konnte. Durch die neueren Entwicklungen der Chipkarten-Betriebssysteme wie beispielsweise Java unterstützende Betriebssysteme oder MULTOS wurde die Möglichkeit geschaffen, eigenen Programmcode auf die Chipkarte zu laden. Um nicht bereits im Betriebssystem vorhandene Programmroutinen nochmals nachprogrammieren zu müssen, besitzen diese Betriebssysteme ein durchdachtes API (*application programming interface*), das Zugriffe auf die wichtigsten Funktionen des Betriebssystems gestattet.

4 Dateien in der Chipkarte

Neben den in ihnen enthaltenen Mechanismen zur Identifizierung und Authentisierung sind Chipkarten vor allem auch Datenspeicher. Dabei haben sie gegenüber anderen Speichermedien, wie beispielsweise Disketten, den entscheidenden Vorteil, daß der Zugriff auf die Daten an Bedingungen geknüpft sein kann.

Die ersten Chipkarten wiesen nur mehr oder minder direkt adressierbare Speicherbereiche auf, in die Daten geschrieben oder aus denen Daten gelesen werden konnten. Der Zugriff erfolgte unter Angabe von physikalischen Speicheradressen. Mittlerweile verfügen alle neueren Chipkarten über ein vollständiges und hierarchisch organisiertes Dateiverwaltungssystem mit symbolischer und hardwareunabhängiger Adressierung.

Allerdings weisen diese Dateiverwaltungen einige chipkartenspezifische Eigenheiten auf: Das auffälligste dabei ist, daß keine Mensch-Maschine-Schnittstelle vorhanden ist. Alle Dateien werden mit hexadezimalen Codes adressiert, und auch die weiteren Kommandos bauen strikt darauf auf, daß hier die Kommunikation nur zwischen zwei Computern abläuft. Ebenfalls typisch für diese Dateiverwaltungen ist, daß sie auf wenig Speicherverbrauch hin ausgelegt sind. Wenn möglich, ist auch noch das letzte redundante Byte vermieden. Da der „Benutzer“ im Terminal ein Computer ist, entstehen dadurch eigentlich keine Nachteile.

Um den Speicherverbrauch so gering wie möglich zu halten, ist üblicherweise auch keine aufwendige Speicherverwaltung vorhanden. Wird eine Datei gelöscht, und selbst dies können erst wenige Betriebssysteme, dann ist es eben nicht selbstverständlich, daß der freiwerdende Platz von einer neu zu erzeugenden Datei eingenommen werden kann. Üblicherweise werden bei der Initialisierung bzw. Personalisierung alle Dateien erzeugt und in die Chipkarte geladen. Danach sind Änderungen auf den Dateiinhalt beschränkt.

Die Eigenheiten des benutzten Speichers beeinflussen natürlich ebenfalls die Art und Weise der Dateiverwaltung. Die Speicherseiten im EEPROM können eben nicht unbegrenzt geschrieben bzw. gelöscht werden, wie dies bei Festplatten an PCs der Fall ist. Dies führt dazu, daß es spezielle Dateiattribute gibt, um Informationen redundant und gegebenenfalls sogar korrigierbar abzulegen.

Interner Aufbau von Dateien

Die neueren Dateiverwaltungssysteme für Chipkarten sind objektorientiert aufgebaut. Dies bedeutet, daß alle Informationen über eine Datei in dieser Datei selber gespeichert sind. Eine weitere Auswirkung dieses Prinzips ist auch, daß eine Datei vor einer Aktion immer zuerst selektiert werden muß. Dateien sind deswegen in diesen objektorientierten Systemen immer in zwei getrennte Teile aufgespalten. Der Dateiheader genannte Teil enthält Informationen über die Struktur, Aufbau der Datei und Zugriffsbedingungen, und in dem mit einem Zeiger verbundenen Dateibody sind die veränderbaren Nutzdaten gespeichert.

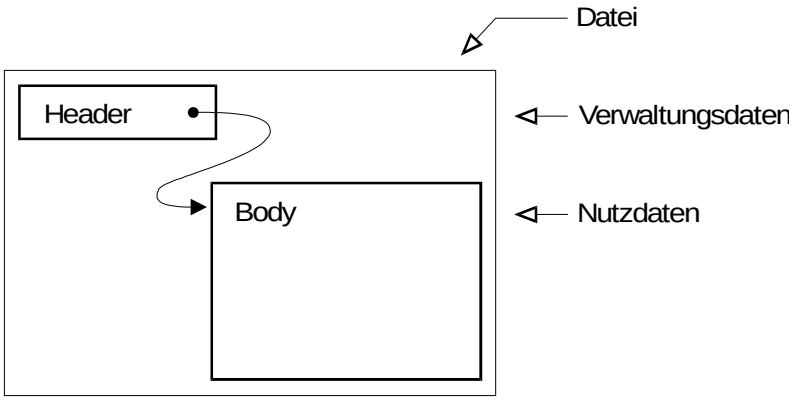

Bild 2 *Der interne Aufbau einer Datei bei Dateiverwaltungen von Chipkarten.*

4.1 Dateitypen

Der Aufbau von Chipkarten-Dateisystemen ist in der ISO/IEC 7816-4 festgelegt und ähnlich dem von DOS oder Unix. Der größte Unterschied besteht darin, daß bei Chipkarten keine anwendungsspezifischen Dateien existieren, wie zum Beispiel spezielle Dateien für eine bestimmte Textverarbeitung. Bei Chipkarten können nur die genormten Dateistrukturen benutzt werden.

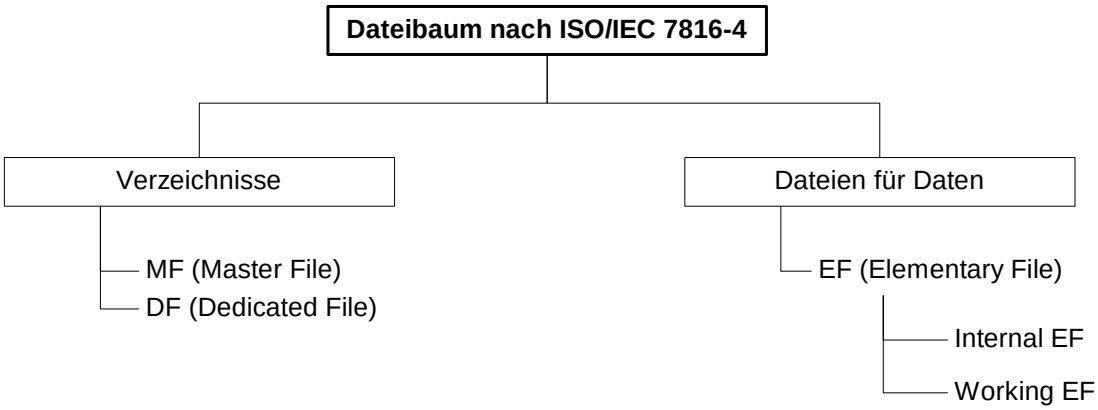

Bild 3 *Klassifizierungsbaum der Dateitypen bei Chipkarten-Betriebssystemen nach ISO/IEC 7816-4.*

Es existieren bei Chipkarten grundsätzlich zwei Kategorien von Dateien. Die Verzeichnisdateien, „Dedicated Files“ (DFs) genannt, und die „Elementary Files“ (EFs), welche die eigentlichen Nutzdaten beinhalten. Die DFs fungieren als eine Art Ordner und beinhalten entweder weitere, untergeordnete DFs oder EFs, die logisch zusammengehören. Die EFs lassen sich noch in Dateien für die äußere Welt (*working EF*) und solche für das Betriebssystem (*internal EF*) typisieren.

MF

Das Root-Verzeichnis, das implizit nach einem Reset der Chipkarte selektiert wird, hat die Bezeichnung „Master File“, abgekürzt MF. In ihm befinden sich alle anderen Verzeichnisse und alle Dateien. Das Master File ist ein Sonderfall eines Dedicated File und stellt den gesamten in der Chipkarte für den Dateibereich verfügbaren Speicher dar. Es muß in jeder Chipkarte vorhanden sein.

DF

Unter dem MF können bei Bedarf noch „Dedicated Files“, abgekürzt DFs, existieren. Diese werden vielfach auch „Directory Files“ genannt, obwohl dies im Widerspruch zur offiziellen ISO/IEC 7816-4-Abkürzung steht. DFs sind Verzeichnisse, in denen weitere Dateien (EFs und DFs) zusammengefaßt sein können. Unter den DFs können auch noch weitere DFs existieren. Im Prinzip ist die Schachtelungstiefe der DFs nicht limitiert. Der in Chipkarten sehr beschränkte Speicherplatz führt jedoch dazu, daß selten mehr als zwei Ebenen von DFs unter dem MF aufgebaut werden.

EF

Die Nutzdaten, die für eine Anwendung notwendig sind, befinden sich in den EFs. EF ist die Abkürzung von Elementary File. Sie können direkt unter dem MF oder auch unter einem DF angeordnet sein. Um Daten sowohl logisch optimal strukturiert als auch speicherplatzminimiert ablegen zu können, besitzen EFs grundsätzlich eine interne Dateistruktur. Dies ist der Hauptunterschied zu Dateien auf PCs, deren interne Struktur durch eine Anwendung, beispielsweise eine Textverarbeitung, vorgegeben ist und nicht direkt durch das Betriebssystem. EFs sind in Working EFs und Internal EFs aufgeteilt.

Working EF

Alle Daten einer Anwendung, welche vom Terminal gelesen oder geschrieben werden müssen, also für die äußere Welt (vom Standpunkt der Chipkarte aus betrachtet) bestimmt sind, befinden sich in den sogenannten Working EFs. Daten, die sich in einem solchen Dateityp befinden, werden nicht vom Betriebssystem benutzt.

Internal EF

Zusätzlich zu den EFs existieren für Anwendungen noch interne Systemdateien, in denen Daten für das Betriebssystem selber, den Ablauf einer Anwendung, geheime Schlüssel oder Programmcode abgelegt werden. Der Zugriff auf diese Dateien ist vom Betriebssystem besonders geschützt. Hier gibt es nun aber zwei Varianten, wie diese internen Systemdateien in der Dateiverwaltung integriert sind. Im Verfahren nach ISO 7816 sind diese Dateien im jeweiligen Anwendungs-DF versteckt angeordnet und können auch nicht selektiert werden. Die Verwaltung dafür übernimmt völlig transparent das Betriebssystem der Chipkarte ähnlich den Resource-Dateien im MacOS. Im Modell nach ETSI EN 726 erhalten diese Systemdateien einen regulären Filenamen (d.h. eine FID) und sind mit diesem auch selektierbar. Das entspricht in seinen wesentlichen Ansätzen der Dateiverwaltung unter DOS.

Dateien für eine Anwendung

Nach einer Konvention faßt man alle Dateien mit Nutzdaten, d.h. EFs, die zu einer Anwendung gehören, immer in einem DF zusammen. Damit erhält man eine klare und übersichtliche Struktur, und durch die Erzeugung des betreffenden DFs kann auf einfache Art eine neue Anwendung in eine Chipkarte eingebracht werden.

Da das MF nur der Sonderfall eines DFs ist, können selbstverständlich bei Chipkarten mit nur einer Anwendung alle dazugehörigen EFs direkt unter dem MF kumuliert werden. In einer typischen Chipkarte für eine einzige Anwendung können sich deshalb alle EFs entweder direkt unter dem MF oder unter dem einzigen DF befinden. Chipkarten mit mehreren Anwendungen besitzen entsprechend viele DFs, in denen die zugehörigen EFs eingerichtet sind.

Unter einem solchen Anwendungs-DF können noch weitere DFs angeordnet sein. Zum Beispiel kann ein DF direkt unter dem MF der Anwendung „Verkehrsleitsystem“ gewidmet sein. Eine zusätzlich Schachtelung könnte in eigenen DFs im Anwendungs-DF die Dateien für die möglichen Sprachen, z.B. „Englisch“ oder „Deutsch“, enthalten.

4.2 Dateinamen

Die Dateien in modernen Chipkarten-Betriebssystemen werden ausnahmslos logisch adressiert und nicht über direkte physikalische Adressen angesprochen.

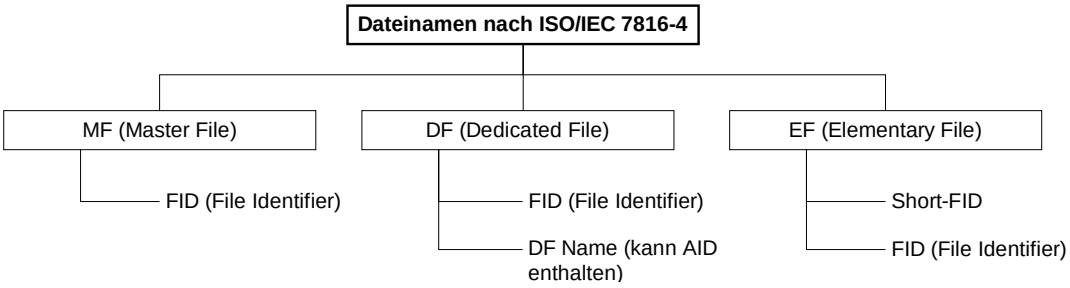

Bild 4 *Klassifizierungsbaum der Dateinamen bei Chipkarten-Betriebssystemen nach ISO/IEC 7816-4.*

File Identifier (FID)

Alle Dateien, einschließlich der Verzeichnisse, besitzen einen 2 Byte langen File Identifier (FID), unter dessen Verwendung sie ausgewählt werden können.

Das MF hat aus historischen Gründen den FID '3F00', der im ganzen logischen Adreßraum ausschließlich dem MF vorbehalten ist. Der logische Dateiname 'FFFF' ist für zukünftige Anwendungen reserviert und darf nicht verwendet werden. Daneben gibt es sowohl von ISO als auch einigen anderen Normen noch weitere reservierte FIDs.

Short File Identifier (Short-FID)

Zur impliziten Selektion von Dateien unmittelbar in einem Kommando wird die Short-FID verwendet. Die Short-FID ist für ein EF optional, muß also nicht zwangsläufig vergeben werden. Sie wird bei der impliziten Dateiselektion im Kommando mit übergeben und hat deshalb nur eine Größe von 5 Bit. Die Short-FID kann somit Werte zwischen 1 und 30 annehmen, da eine Short-FID von '0' das aktuell selektierte EF adressiert.

DF-Name

Die DFs sind die Zusammenfassungen von Dateien für die einzelnen Anwendungen. Sie sind in ihrer Art wie Verzeichnisse oder Ordner und können weitere Verzeichnisse oder EFs aufnehmen. In Zukunft wird der dafür zur Verfügung stehende Adreßraum mit dem 2 Byte langen FID zu klein werden. Deshalb besitzen DFs zusätzlich zu ihrem FID einen sogenannten „DF-Name“. Er hat nach ISO/IEC 7816-4 eine Länge zwischen 1 und 16 Byte. Der DF-Name bietet einen genügend großen Adreßraum, damit eine Chipkarten-Anwendung weltweit eindeutig ist. Da es bei freier Auswahl trotzdem irgendwann zwei DFs mit gleichem DF-Name geben würde, wird der DF-Name in der Regel nur in Verbindung mit dem in der ISO/IEC 7816-5 definierten AID (*application identifier*) benutzt. Der AID kann eine Länge zwischen 5 Byte und 16 Byte haben und setzt sich aus zwei von der ISO definierten Datenelementen zusammen. Der AID ist somit eine Teilmenge des DF-Name.

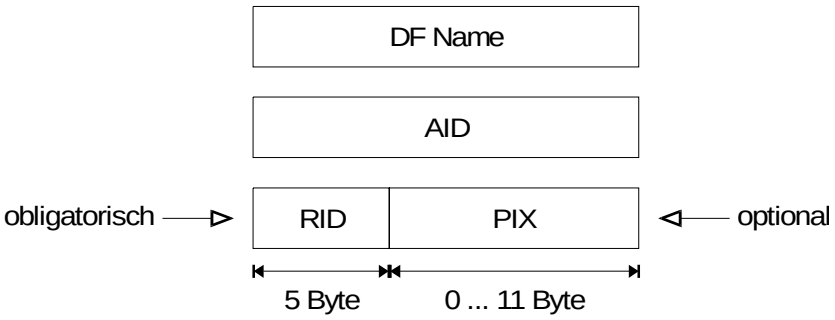

Bild 5 *Der DF-Name im Zusammenhang mit dem Aufbau des AID (*application identifier*) aus RID (*registered identifier*) und PIX (*proprietary application identifier extension*).*

Aufbau und Codierung des Application Identifier (AID)

Der Application Identifier (AID) setzt sich selber wiederum aus zwei Datenelementen zusammen. Das erste Datenelement ist der Registered Identifier (RID) mit einer festen Länge von 5 Byte. Er wird entweder von einer nationalen oder internationalen Registrierungsstelle vergeben und beinhaltet einen Ländercode, eine Anwendungskategorie und eine Nummer für den Anwendungsanbieter. Dieser Zahlencode führt zu einer nur ein einziges Mal vergebenen RID, die weltweit zur Identifizierung einer bestimmten Anwendung benutzt werden kann.

Falls es notwendig ist, kann der Anwendungsanbieter der RID eine Proprietary Application Identifier Extension (PIX) nachstellen, die der optionale zweite Teil des AID ist. Die bis zu 11 Byte lange PIX kann zum Beispiel eine Serien- und Versionsnummer sein und damit zur Verwaltung der Anwendung benutzt werden.

4.3 Dateistrukturen von EFs

Die EFs bei Chipkarten haben im Gegensatz zu Dateien in DOS Systemen eine interne Struktur. Dieses Struktur kann je nach Verwendungszweck für jedes EF individuell gewählt werden. Dies hat große Vorteile für die äußere Welt, denn durch diese internen Strukturen ist es möglich, Datenbestände so aufzubauen, daß auf sie sehr schnell und zielgerichtet zugegriffen werden kann.

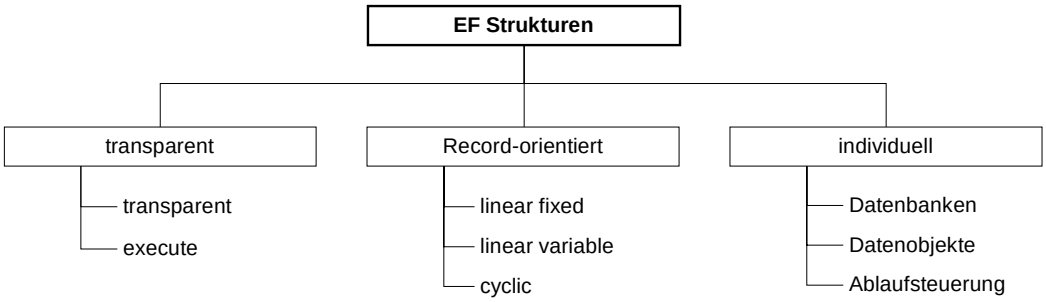

Bild 6 *Klassifizierungsbaum der Dateistrukturen von EFs bei Chipkarten-Betriebssystemen.*

Dateistruktur „transparent“

Die Dateistruktur „transparent“ wird oft auch noch als binäre oder amorphe Struktur bezeichnet. Dies bedeutet in anderen Worten, daß eine transparente Datei keine innere Struktur aufweist. Auf die in der Datei enthaltenen Daten kann byteweise oder blockweise schreibend oder lesend mit einem Offset zugegriffen werden. Dazu werden die Kommandos READ BINARY, WRITE BINARY und UPDATE BINARY verwendet.

Bild 7 *Der Aufbau der Dateistruktur „transparent“.*

Die Anwendung dieser Datenstruktur liegt hauptsächlich im Bereich von Daten, die nicht strukturiert aufgebaut oder sehr kurz sind. Ein typisches Einsatzgebiet wäre eine Datei mit einem digitalisierten Paßfoto, das von einem Terminal aus der Chipkarte ausgelesen werden kann.

Dateistruktur „linear fixed“

Die linear fixed-Datenstruktur basiert auf der Verkettung von gleich langen Datensätzen, d.h. Records. Ein Record wiederum ist eine Aneinanderreihung von einzelnen Bytes. Auf die einzelnen Records dieser Dateistruktur kann wahlfrei zugegriffen werden. Die kleinste Zugriffsgröße ist ein einzelner Record, es ist also nicht möglich, nur auf Teile eines Records zuzugreifen. Gelesen bzw. geschrieben werden kann in diese Struktur mit READ RECORD, WRITE RECORD und UPDATE RECORD.

Die typische Anwendung dieser Dateistruktur ist ein Telefonverzeichnis, in dem als erstes der Name steht und dann, ab einer festen Stelle, die dazugehörige Telefonnummer.

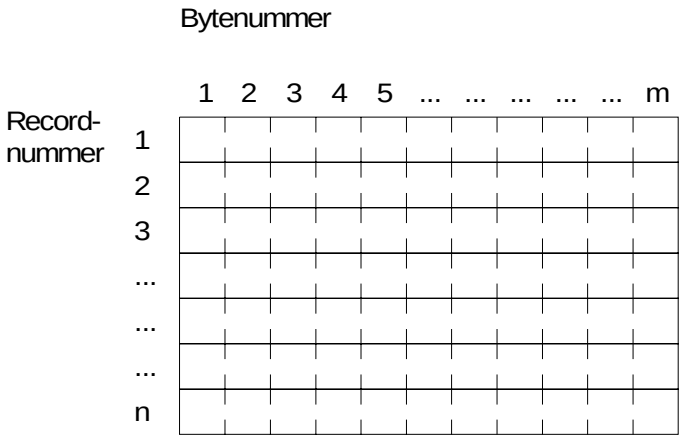

Bild 8 *Der Aufbau der Dateistruktur linear fixed.*

Dateistruktur „linear variable“

In der Dateistruktur „linear fixed“ besitzen alle Records die gleiche Länge. Dadurch wird oft Speicherplatz verschwendet, da viele Record-orientierte Daten von variabler Länge sind. Man denke dabei nur an die Namen in einem Telefonverzeichnis. Dieser Forderung nach Speicherplatzminimierung kommt die Struktur linear variable nach. Hier kann jeder einzelne Record individuell eine definierte Länge annehmen. Dies führt zwangsläufig dazu, daß jeder Record ein zusätzliches Informationsfeld über seine Länge besitzt. Sonst ist der Aufbau analog dem in der Struktur „linear fixed“.

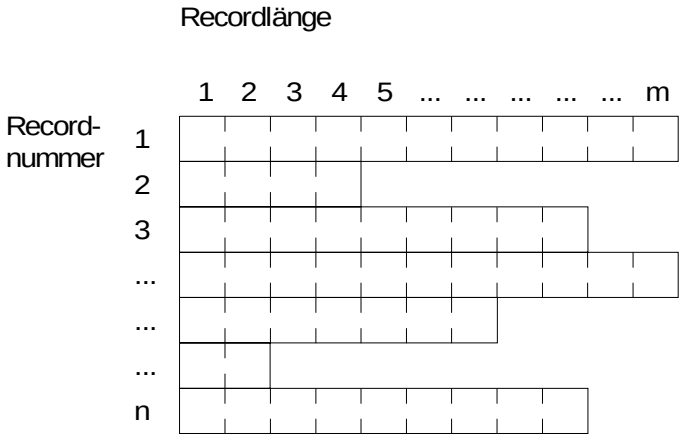

Bild 9 *Der Aufbau der Dateistruktur linear variable.*

Vorzugsweise findet diese Struktur dann Verwendung, wenn Datensätze mit sehr unterschiedlicher Länge gespeichert und Speicherplatz in der Chipkarte gespart werden soll. So könnte beispielsweise das obige Telefonverzeichnis dahingehend optimiert werden, daß die Records genauso lang sind wie die eigentlichen Einträge, und damit nicht alle Records die gleiche Länge haben. Allerdings benötigt die Ver-

waltung dieser Dateistruktur im Chipkarten-Betriebssystem sowohl Programmcode als auch zusätzlichen Speicher für die individuellen Recordlängen. Deshalb bieten oftmals Betriebssysteme für Mikrocontroller mit wenig Speicherplatz diese Dateistruktur nicht an. Die ISO/IEC 7816-4 läßt diese Einschränkung konsequenterweise auch in einigen Profilen ausdrücklich zu.

Dateistruktur „cyclic“

Diese Struktur basiert auf der linear fixed-Datenstruktur. Sie besteht also aus einer bestimmten Anzahl von Records mit gleicher Länge. Zusätzlich beinhaltet das EF noch einen Zeiger, der immer auf den letzten geschriebenen Datensatz zeigt, welcher grundsätzlich die Nummer 1 hat. Erreicht der Zeiger den letzten Datensatz im EF, dann wird er beim nächsten Schreibzugriff vom Betriebssystem automatisch wieder auf den ersten Datensatz gesetzt. Er verhält sich also wie bei einer analogen Uhr mit einem Stundenzeiger.

Die typische Anwendung für diese Struktur sind Dateien innerhalb der Chipkarte für die Aufzeichnung von Protokollen, da der jeweils älteste Eintrag immer durch den neuesten überschrieben wird.

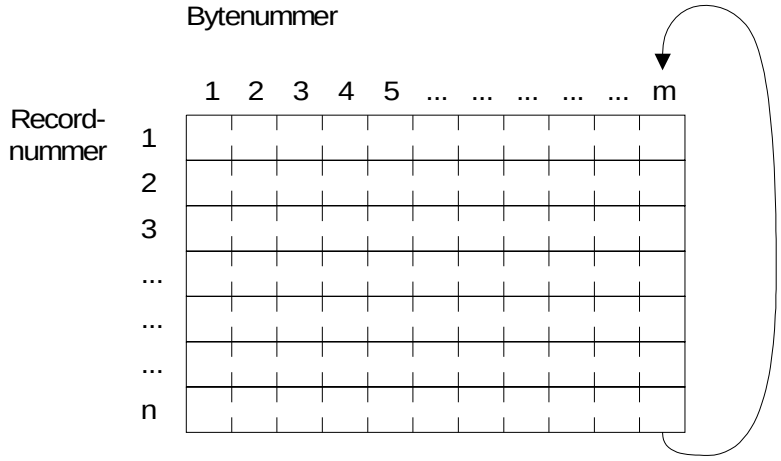

Bild 10 *Der Aufbau der Dateistruktur „cyclic“.*

Dateistruktur „execute“

Die folgende Dateistruktur ist im Prinzip nicht eigenständig, da sie auf der transparenten Datenstruktur beruht. Sie ist in der europäischen Norm EN 726-3 beschrieben und bietet innerhalb des Betriebssystems große Erweiterungsmöglichkeiten. Die Datenstruktur „execute“ ist nicht zum Speichern von Daten, sondern für die Ablage von ausführbarem Programmcode gedacht. Zugreifen kann man auf Dateien mit der Struktur „execute“ mit den entsprechenden Kommandos für transparente Dateistrukturen.

Dateistruktur für Datenbanken, Database File

Die ISO/IEC 7816-7 definiert mit SCQL eine Untermenge von SQL für Chipkarten. Um die Daten in einem für die SCQL-Kommandos auslesbaren Form im Dateisystem der Chipkarte ablegen zu können, ist es notwendig, eine eigene Dateistruktur dafür vorzusehen. Diese ist in ihrem Aufbau nicht normiert, sondern dem jeweiligen Betriebssystem-Hersteller überlassen. In dem Database File sind die eigentlichen Nutzdaten der Datenbank, unterschiedliche Sichten auf die Datenbank, die Zugriffsrechte auf die Datenbank und die Benutzer-Profile abgelegt.

Dateistruktur für Datenobjekte

Die beiden Kommandos GET DATA und PUT DATA der ISO/IEC 7816-4 werden benutzt, um TLV-codierte Datenobjekte in der Chipkarte abzulegen und von dort wieder auszulesen. Dies kann entweder völlig unabhängig von der Dateiverwaltung realisiert werden oder innerhalb einer Dateiverwaltung durch eine besondere Dateistruktur für die Ablage von Datenobjekten. Findet eine Realisierung über die Dateiverwaltung statt, dann kann man beispielsweise eine geringfügig angepaßte transparente oder linear variable Dateistruktur als Aufbewahrungsplatz für die Datenobjekte benutzen. Die Kommandos GET DATA und PUT DATA greifen dann über die Dateiverwaltung auf diese modifizierten Dateistrukturen zu.

Dateistruktur für Ablaufsteuerung

Weist ein Chipkarten-Betriebssystem eine Ablaufsteuerung für Kommandosequenzen auf, dann müssen die Informationen, wann welche Kommandos akzeptiert werden, ebenfalls im Speicher abgelegt werden. Dies geschieht üblicherweise in einer Datei mit besonders angepaßter Struktur für diese Aufgabe. Diese ist jedoch nicht genormt, und jedes Betriebssystem mit Ablaufsteuerung besitzt ausnahmslos ein eigenes und nicht zu anderen kompatibles Format.

4.4 Zugriffsbedingungen auf Dateien

Alle Dateien besitzen im Rahmen ihres objektorientierten Aufbaus Informationen, die den Zugriff im Rahmen des Dateimanagements regeln. Physikalisch ist die dazugehörige Codierung immer im Header einer Datei untergebracht. Die gesamte Sicherheit der Dateiverwaltung einer Chipkarte basiert auf der Rechteverwaltung für den Dateizugriff, da bei ihr der Zugriff auf die Dateiinhalte geregelt wird.

Die Zugriffsbedingungen werden bei der Erzeugung einer Datei festgelegt und sind dann im Regelfall nicht mehr zu verändern. Die für eine Datei möglichen Zugriffsbedingungen differieren je nach den im Betriebssystem vorhandenen Kommandos sehr stark. Es hat beispielsweise keinen Sinn, Zugriffsbedingungen für READ RECORD zu definieren, wenn dieses Kommando in einem Chipkarten-Betriebssystem nicht vorhanden ist.

Beim MF und den DFs werden im Gegensatz zu den EFs keine Informationen über den Datenzugriff (Schreib- oder Leserechte) gespeichert, sondern unter anderem die Zugriffsbedingungen für die Erzeugung von neuen Dateien. Je nach Dateityp sind also verschiedene Zugriffsbedingungen gespeichert: In den EFs für den Zugriff auf die Dateiinhalte und im MF und den DFs Bedingungen, die innerhalb dieser Organisationsstruktur gelten.

Es kann zwischen zustands- und kommandoorientierten Zugriffsbedingungen unterschieden werden. Bei zustandsorientierten Zugriffsbedingungen ist für einen Zugriff der dazu benötigte Zustand festgelegt. Es kann aber nicht nur ein bestimmter Zustand für den Zugriff erlaubt sein, sondern auch eine Vergleichsbedingung angegeben sein. So kann zum Beispiel ein Lesezugriff ab dem Zustand 5 erlaubt sein. Für jeden Zustand, der kleiner ist als 5, wäre dann der Lesezugriff verboten. Natürlich ist es auch möglich, für einen Zugriff mehrere unterschiedliche Zustände festzulegen. So könnte ein Lesezugriff im Zustand 5, 8 und 9 erlaubt sein.

Im Gegensatz dazu werden in kommandoorientierten Betriebssystemen die vor dem Zugriff korrekt auszuführenden Kommandos definiert. Dies betrifft vor allem Authentisierungs- und Identifizierungskommandos. Beispielsweise kann man nur dann auf eine Datei schreibend zugreifen, wenn vorher die PIN mit dem Kommando VERIFY erfolgreich überprüft wurde.

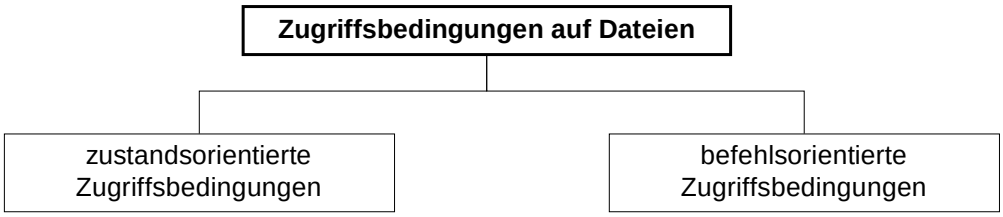

Bild 11 *Klassifizierungsbaum der beiden möglichen Arten von Zugriffsbedingungen auf Dateien.*

5 Atomare Abläufe

Oftmals wird für die Software im Mikrocontroller der Chipkarte gefordert, daß bestimmte Teile von ihr entweder vollständig oder gar nicht durchlaufen werden. Abläufe, die nicht aufteilbar sind und diese Forderung erfüllen, nennt man deshalb auch atomare Abläufe. Sie treten immer in Verbindung mit EEPROM-Schreibroutinen auf.

Atomaren Abläufen liegt der Gedanke zugrunde, daß man beim Schreiben ins EEPROM sicherstellen will, daß die betreffenden Daten in keinem Fall nur teilweise geschrieben worden sind. Dies würde beispielsweise dann der Fall sein, wenn der Benutzer die Chipkarte im falschen Augenblick aus dem Terminal zieht oder es zu einem plötzlichen Stromausfall kommt. Da die Chipkarte über keinerlei Pufferspeicher für elektrische Energie verfügt, verliert die Software in der Karte in diesen Fällen sofort ihre Aktionsmöglichkeiten.

Vor allem bei elektronischen Geldbörsen auf Chipkarten muß unter allen Umständen sichergestellt sein, daß die Einträge in den Dateien vollständig und korrekt sind. Beispielsweise wäre es äußerst fatal, wenn der Saldo einer Börse durch das Ziehen der Karte aus dem Terminal nur unvollständig auf den neuesten Stand gebracht würde. Auch müssen die entsprechenden Einträge in den Protokolldateien immer vollständig sein. Da die Hardware von Chipkarten atomare Abläufe nicht unterstützt, müssen diese deshalb mit Software realisiert werden. Die dazu angewandten Methoden sind im Prinzip nicht neu, sondern werden im Bereich von Datenbanken und Festplattenlaufwerken schon seit langer Zeit eingesetzt. Eine Variante, die bei Chipkarten-Betriebssystemen Verwendung findet, ist nachfolgend in ih-

ren grundlegenden Abläufen beschrieben. Dieses Fehlerbehebungsverfahren (*error recovery*) ist transparent zur äußeren Welt und erfordert dadurch keinerlei Änderungen in eventuell bestehenden Anwendungen.

6 Chipkarten-Betriebssysteme mit nachladbarem Programmcode

Man kann wohl ohne zu übertreiben behaupten, daß sich hier innerhalb eines Jahres (1997/1998) ein Paradigmenwechsel vollzogen hat. Nachladbarer Programmcode in Chipkarten wird mittlerweile als Regelfall und nicht mehr als Ausnahme angesehen, obwohl er noch von den wenigsten Chipkarten-Betriebssystemen unterstützt wird.

Im Gegensatz zu allen anderen Betriebssystemen für Computer ist es aber nicht generell üblich, Programme in Chipkarten nach Ausgabe einzubringen und dort bei Bedarf auszuführen. Dies ist aber eigentlich neben der Datenspeicherung eine der Hauptfunktionen aller Betriebssysteme. Es gibt natürlich Gründe, warum bisher gerade diese Funktionalität in der Chipkartenwelt weitgehend gefehlt hat.

Technisch und funktionell gesehen stellt ausführbarer Programmcode, beispielsweise in Dateien (d.h. EFs) gespeichert, keinerlei Problem dar. Neuere Betriebssysteme bieten deshalb auch die Möglichkeit, Dateien mit ausführbarem Code zu verwalten und auch zu einem Zeitpunkt nach der Personalisierung in die Chipkarte zu laden. Damit ist es möglich, daß beispielsweise ein Anwendungsanbieter Programmcode in der Chipkarte ausführen kann, den der Betriebssystem-Hersteller nicht kennt. So kann ein Anwendungsanbieter einen nur ihm selbst bekannten Verschlüsselungsalgorithmus in die Chipkarte einbringen und dort ausführen. Hierdurch ist es möglich, das Wissen um die Sicherheitsfunktionen des Systems auf verschiedene Parteien zu verteilen, was eine Grundforderung in Sicherheitssystemen ist.

Ein weiterer gewichtiger Grund für den Mechanismus des nachladbaren Programmcodes ist die sich damit eröffnende Möglichkeit der Beseitigung von Programmfehlern (*bug-fixing*) in vollständig personalisierten Karten. Erkannte Fehler im Betriebssystem können damit bei bereits ausgegebenen Karten behoben oder zumindest entschärft werden.

Es gibt grundsätzlich zwei Wege, Programmcode in einer Chipkarte auszuführen. Der erste und technisch einfachste Weg ist, compilierten Code in der Maschinensprache des Zielprozessors (*native code*) in Dateien der Chipkarte zu laden. Dieser Programmcode muß natürlich relokierbar sein, da die Speicheradressen nach außen nicht bekannt sind. Neben der technischen Unkompliziertheit dieser Lösung kann der Programmcode noch mit voller Ausführungsgeschwindigkeit des Prozessors abgearbeitet werden, was diese Lösung gerade für nachladbare Algorithmen sehr interessant macht. Weiterhin ist auch kein zusätzlicher Programmcode für einen Interpreter in der Chipkarte notwendig. Das große Problem dieser Lösung ist, daß der nachgeladene Programmcode bei Mikrocontrollern ohne MMU (*memory management unit*) auch auf Speicherbereiche von Fremdanwendungen zugreifen kann.

Der zweite Weg, ausführbaren Programmcode in Chipkarten auszuführen, besteht darin, ihn zu interpretieren. Der Interpreter prüft dann während der Programmausführung, welche Speicherbereiche angesprochen werden. Die Interpretation muß aber schnell ablaufen, da ein langsam ausgeführter Programmcode keine

Vorteile mehr bringt. Ebenso soll die Implementation eines Interpreters selber sowenig Speicher wie möglich in Anspruch nehmen, da dieser bekanntermaßen stark limitiert ist. Die derzeitig bekanntesten Lösungsvarianten dazu sind die Java Card Spezifikation und der C-Interpreter MEL (*Multos executable language*) von Multos. Mittlerweile gibt es für Chipkarten sogar einen BASIC-Interpreter. Interpreter, die dem Programm einer Anwendung einen eigenen geschützten Speicher zur Verfügung stellen, sind im übrigen nicht für Fehlerbeseitigungen im Betriebssystem von Chipkarten geeignet, da sie auf diese Programm- und Datenteile konsequenterweise keinen Zugriff haben.

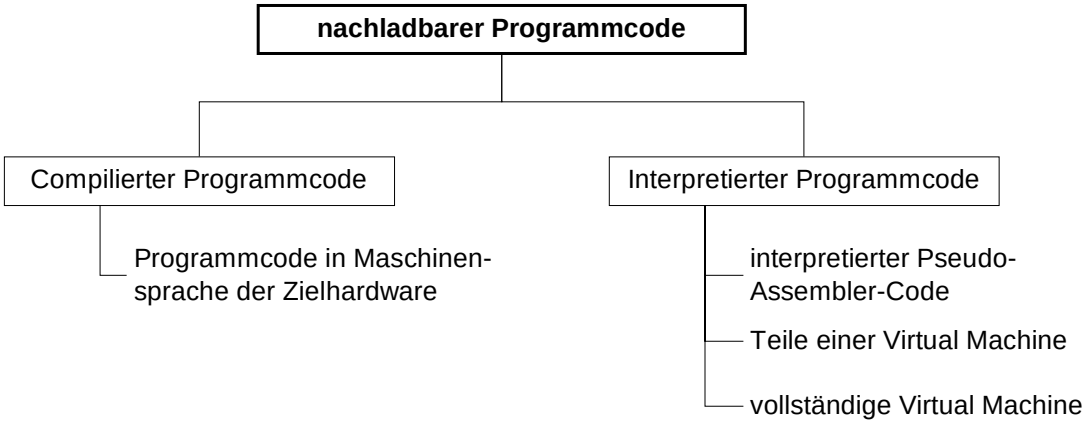

Bild 12 *Klassifizierungsbaum der Varianten, um ausführbaren Programmcode in Chipkarten-Betriebssysteme nachzuladen und dort auszuführen.*

Das Urproblem aller Interpreter ist jedoch die langsame Abarbeitungsgeschwindigkeit, welche für dieses Prinzip immanent ist. Um dieses Minus auszugleichen und um den Programmcode für den eigentlichen Interpreter so klein wie möglich zu halten, gibt es mehrere Lösungsansätze. Die einfachste Methode ist es, einen Pseudocode zu interpretieren, welcher idealerweise den Maschinenbefehlen der Zielhardware möglichst ähnlich sein sollte. Die Abarbeitungsgeschwindigkeit ist durch die maschinennahe Pseudosprache verhältnismäßig hoch, und es kann maschinenunabhängiger Programmcode benutzt werden. Speicherzugriffe während der Interpretation können überwacht werden, was aber nicht zwangsläufig der Fall sein muß. Eine langsamere und programmiertechnisch etwas aufwendigere Lösung ist die Aufspaltung des Interpreters in einen offcard-Teil (*offcard virtual machine*) und einen oncard-Teil (*oncard virtual machine*). Dieser Lösungsweg wird bei vielen heutigen Java Card Implementationen beschritten. Er hat den großen Vorteil eines verläßlichen Speicherschutzes und vollständiger Hardwareunabhängigkeit. Nachteilig ist die Aufteilung des Interpreters in off- und oncard-Teil. Dies bedingt zwangsläufig einen kryptografischen Schutz beim Übertragen von Programmen zwischen offcard-Teil und oncard-Teil des Interpreters, da mit manipuliertem Programmcode der oncard-Teil des Interpreters bewußt zu einem Fehlverhalten gebracht werden kann.

Die technisch optimale Lösung ist ein vollständiger Interpreter auf der Chipkarte. Damit ist es möglich, in die Chipkarte beliebigen Programmcode zu laden und dort ohne Risiko für andere auf der Chipkarte befindliche Anwendungen auszuführen.

Allerdings ist der Programmumfang dazu auch entsprechend groß, weshalb es sicherlich noch einige Jahre und mehrere Chipgenerationen dauern wird, bis diese Variante breiten Einzug in die Chipkartenwelt hält.

6.1 Executable Native-Code

Mikrocontroller für Chipkarten haben zur Zeit meistens Prozessoren, die über keinerlei Speicherschutzmechanismen oder Überwachungsmöglichkeiten verfügen. Sobald sich der Programmzähler innerhalb eines fremden Maschinencodes für den Prozessor befindet, liegt die gesamte Kontrolle aller Speicher und Funktionen bei diesem ausführbaren Code. Es gibt dann keinerlei Möglichkeiten mehr, dieses ausführbare Programm in seinen Funktionen zu beschränken. Jede adressierbare Speicheradresse kann unter Umgehung aller Speichermanager oder Handler gelesen und – sofern im RAM oder EEPROM – auch geschrieben werden. Alle Speicherinhalte können dann natürlich auch über die Schnittstelle zum Terminal gesendet werden.

Genau dies ist die Schwachstelle bei ausführbaren und nachladbaren Programmen. Würde man jedermann ein Nachladen von Programmen erlauben, oder wäre es durch Umgehung der Schutzmechanismen möglich, dann ist keine Sicherheit mehr für geheime Schlüssel oder Informationen innerhalb des gesamten Speicherbereichs gegeben. Dies wäre der ideale Angriff auf eine Chipkarte. Diese Karte würde sich nach außen hin wie eine nicht manipulierte Karte verhalten, und mit einem speziellen Kommando könnten der gesamte Speicher ausgelesen oder Teile davon geschrieben werden.

Wäre das Laden nur einigen wenigen Anwendungsanbietern erlaubt – was mit einer gegenseitigen Authentisierung vor dem eigentlichen Laden des Programmcodes ohne weiteres durchführbar ist – so ist das Problem auch nicht aus der Welt geschafft. Der Anwendungsanbieter kann ohne Einschränkungen über die Grenzen seines ihm zugeteilten DFs auf geheime Informationen von anderen vorhandenen Anwendungen zugreifen. Das System wäre wiederum gebrochen.

Doch gibt es noch ein weiteres stichhaltiges Argument gegen von Dritten nachladbaren Programmcode. Um wichtige Funktionen im Betriebssystem nutzen zu können, muß der Ersteller der nachladbaren Datei alle Einsprungadressen und Aufrufparameter kennen. Die Betriebssystem-Hersteller erachten es aber für die Sicherheit als relevant, daß sowenig wie möglich über interne Abläufe oder Adressen von Programmcode bekannt ist. Zusätzlich müßte auch noch sichergestellt werden, daß der eingebrachte Code genau das fehlerfrei ausführt, was beabsichtigt ist, und nicht etwa ein trojanisches Pferd enthält. Dies kann dann wiederum nur eine unabhängige Instanz prüfen.

Die eleganteste und auch die wohl zukunftsträchtigste Lösung ist der Einsatz einer hardwareunterstützten Speicherverwaltung (*memory management unit – MMU*) zusätzlich zum eigentlichen Prozessor in der Chipkarte. Diese prüft den ablaufenden Programmcode mit einer Hardwareschaltung auf die Einhaltung seiner ihm zugewiesenen Grenzen. Erst dann wäre es ohne den Verlust an Sicherheit möglich, jeden Anwendungsbetreiber Programmcode ohne vorherige Prüfung durch den Kartenherausgeber in die Chipkarte laden zu lassen. Diesem Anwender würde man den physikalisch zusammenhängenden Speicherbereich eines DFs zuweisen. Die MMU prüft die zugeordneten Speichergrenzen während des Aufrufs eines im DF nachgeladenen Programms. Werden die Grenzen überschritten, so kann man über einen Interrupt

den Programmablauf unmittelbar stoppen und die Anwendung bis auf weiteres sperren.

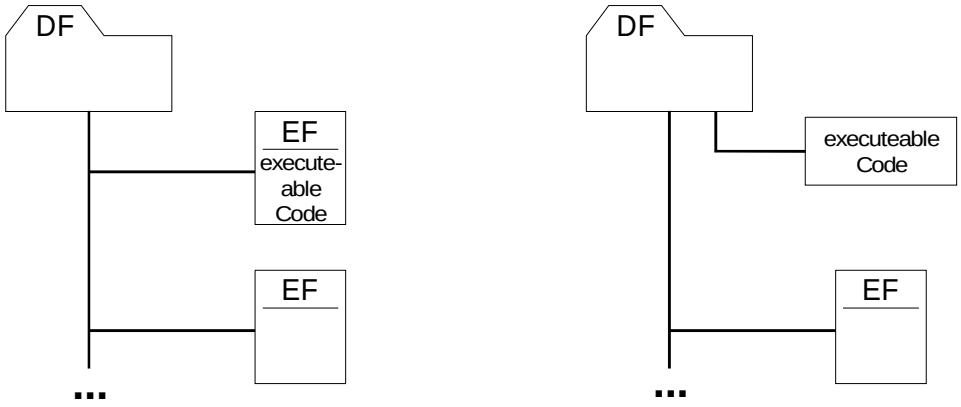

Bild 13 *Die beiden unterschiedlichen Varianten zur Einbringung von ausführbarem Programmcode in ein übliches Chipkarten-Betriebssystem. Links als ausführbare Datei und rechts als ASC (*application specific commands*).*

Für das Nachladen von nativem Programmcode in Chipkarten existieren zwei Varianten der Realisierung. In der ersten befindet sich der Programmcode in einem EF mit der Struktur „executable“. Nach einer vorherigen Selektion ist das EF mit einem Kommando EXECUTE ausführbar. Je nach Anwendung ist dazu vorher noch eine Authentisierung notwendig. Die Parameter für den Programmaufruf sind im Kommando EXECUTE an die Chipkarte enthalten. Die vom Programm im EF erzeugte Antwort kommt als Teil der Antwort auf das Kommando zum Terminal zurück.

Die zweite Variante gestaltet sich vom Prinzip her etwas anders. Man benutzt dabei einen objektorientierten Ansatz. Dieser ist unter anderem auch in der EN 726-3 als *Application Specific Commands* (ASC) beschrieben. Nach dieser Norm enthält ein DF die vollständige Anwendung mit allen ihren Dateien und anwendungsspezifischen Kommandos. In einem in diesem DF intern vom Betriebssystem verwalteten Speicherbereich kann Programmcode nachgeladen werden. Dies geschieht mit einem speziellen Kommando, das alle dafür notwendigen Informationen an die Chipkarte sendet. Ist nun das betreffende DF selektiert und wird ein Kommando an die Karte gesendet, so prüft das Betriebssystem, ob es zu den Nachgeladenen gehört, und ruft gegebenenfalls ohne weiteres Zutun den im DF befindlichen Programmcode auf. Ist hingegen ein anderes DF selektiert, so existiert das nachgeladene Kommando in diesem Kontext nicht.

6.2 Java Card

Im Jahr 1996 wurde von Europay ein Papier über OTA (*open terminal architecture*) veröffentlicht, in welchem ein Forth Interpreter für Terminals beschrieben und in weiten Teilen spezifiziert war. Zweck war, eine einheitliche Softwarearchitektur für Terminals zu schaffen, um die Grundlage für eine hardwareunabhängige Terminalprogrammierung zu schaffen. Dann müßte man eine bestimmte Anwendung (z.B. Bezahlen mit Kreditkarte) nur noch ein einziges Mal programmieren, und diese Software würde auf allen Terminals der verschiedenen Hersteller unverändert laufen. Dieses Modell wurde aber bisher nie vollständig realisiert, es sorgte allerdings in der Chipkartenwelt für ausführliche Diskussionen.

Als dann im Herbst 1996 bekannt wurde, daß die Firma Schlumberger eine Chipkarte entwickelt, welche Programme abarbeiten kann, die in der Programmiersprache Java erstellt sind, hielt sich das Erstaunen in Grenzen. Das Prinzip eines Interpreters auf speicherplatzarmen Mikrocontrollern war durch die Open Terminal Architecture (OTA) schon reichlich bekannt. Die veröffentlichte Spezifikation Java Card 1.0 sah zur Einbindung von Java in das ISO/IEC 7816-4 Betriebssystem ein dazugehöriges API (*application programming interface*) vor, so daß von Java aus auf das bei Chipkarten übliche Dateisystem mit MF, DFs und EFs zugegriffen werden konnte.

Nach kurzzeitiger Verwunderung vieler Hersteller von Chipkarten-Betriebssystemen, warum eine Sprache wie Java, die einen üblichen Speicherbedarf weit jenseits von einem Megabyte hat, auf Chipkarten verwendet werden soll, kam es aber bereits im Frühjahr 1997 zu einem ersten Treffen von nahezu allen großen Chipkartenherstellern und der Firma Sun, die Java entwickelt und bekanntgemacht hat.

Dies war die erste Konferenz des inzwischen sogenannten Java Card Forums (JCF), welches das international tätige Standardisierungsgremium für Java auf Chipkarten ist. Die Aufgabe der technischen Gruppe des Java Card Forums ist es, eine Untermenge von Java für Chipkarten festzulegen, den Rahmen für den Java Interpreter (d.h. die *Java Virtual Machine – JVM*) zu spezifizieren und sowohl ein allgemeines, als auch anwendungsspezifische (z.B.: GSM, Zahlungsverkehr) APIs als Schnittstelle zwischen Chipkarten-Betriebssystem und Java festzulegen.

Die zur Zeit aktuellen Spezifikationen haben die Bezeichnung Java Card Version 2.1 Application Programming Interface, Language Subset and Virtual Machine Specification und Programming Concepts. Sie sind auch in der jeweils aktuellen Version auf dem WWW-Server des Java Card Forums verfügbar.

Java für Chipkarten

Die wirklichen großen Vorteile einer modernen Programmiersprache wie Java im Einsatz bei Chipkarten sind nicht nur in der Richtung zu sehen, daß nun jeder Programme für Chipkarten schreiben kann. Das wäre mit Assembler oder C, offengelegten Schnittstellen und einigen Anpassungen mit den meisten Chipkarten-Betriebssystemen denkbar. Interessant ist ein Konzept wie die Java Card vor allem für große Systembetreiber. Diese haben das Problem, daß sie unterschiedliche Masken auf unterschiedlichen Chips von unterschiedlichen Kartenherstellern einkaufen müssen. Dieses Multiple-Sourcing, das aus taktischen Gründen (reduzierte Abhängigkeit von einem Hersteller, Preisdruck auf die Hersteller) durchaus Sinn macht,

verursacht aber andererseits laufend Probleme mit Kompatibilität und Testing. Es ist auf absehbare Zeit nicht erreichbar, daß sich zwei Betriebssysteme unterschiedlicher Hersteller in allen Variationsmöglichkeiten auf der Schnittstelle gleich verhalten. Dies stellt jedoch für Systembetreiber eine ernsthafte Schwierigkeit dar.

Die ideale Lösung aus Sicht der Systembetreiber wäre somit hardwareunabhängiger Programmcode, der auf einem evaluierten Interpreter in Chipkarten einheitlich ausgeführt wird. Man müßte dann dieses Anwendungsprogramm nur noch einmal erstellen, testen, evaluieren, und es könnte dann von den verschiedensten Chipkarten-Betriebssystemen abgearbeitet werden. Nach außen, d.h. auf der Schnittstelle, wären keine Unterschiede sichtbar. Damit blieben die Vorteile des Multiple-Sourcing erhalten, während die Nachteile verschwinden.

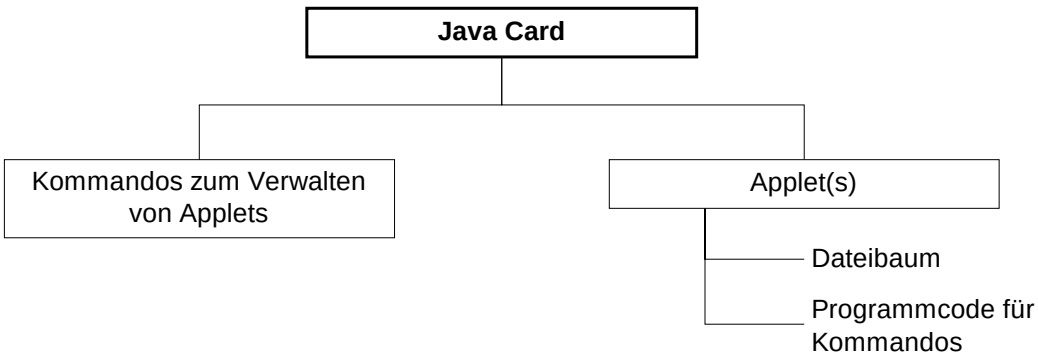

Bild 14 *Die beiden grundlegenden informationstechnischen Bestandteile einer Chipkarte mit Java.*

Diese Stichpunkte sollte man im Hinterkopf behalten, wenn man betrachtet, wie Java in Chipkarten eingebunden ist. Dies ersten Versionen sahen noch vor, daß der Java Bytecode in einem EF unter dem MF oder einem DF abgelegt wird. Mit einem EXECUTE Kommando wäre die Virtual Machine mit dem Programm in dem EF gestartet worden. Zum Dateisystem gab es ein dazugehöriges API (*application programming interface*), so daß aus Dateien gelesen und in Dateien geschrieben werden konnte.

Diese Lösung hat sich jedoch nicht durchgesetzt. Eine Chipkarte mit Java besitzt nach der Java-Card-Spezifikation eine Java Virtual Machine, welche in der Kartenfertigung aktiviert und am Ende des Kartenlebenszyklus deaktiviert wird. Ein Dateisystem nach ISO/IEC 7816-4 ist nicht mehr vorgesehen, da sich dieses auch mit Objekten innerhalb eines Java Applets aufbauen läßt. Dazu existieren einige Klassen, welche den Aufbau eines ISO/IEC 7816-4 konformen Dateibaums relativ einfach ermöglichen. Der Programmcode sowie der dazugehörige Dateibaum sind dann Teil eines Applets, welches in die Chipkarte geladen wird. Es kann dort mit einer eindeutigen AID und dem SELECT Kommando ausgewählt werden. Nach der Selektion des Applets erhält es automatisch alle Kommandos zur Abarbeitung. Der Programmcode des Applets kann dann die Kommandos und deren Daten auswerten und bearbeiten sowie die entsprechenden Zugriffe auf das Dateisystem durchführen.

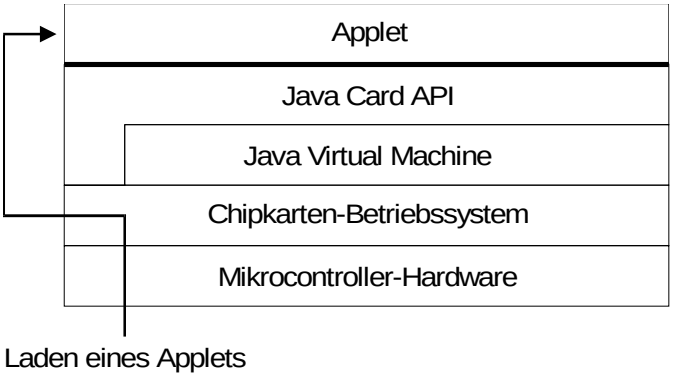

Bild 15 *Der grundsätzliche Ablauf beim Laden eines Applets in ein Chipkarten-Betriebssystem mit Java.*

Dieses Prinzip schafft maximale Flexibilität und Kompatibilität, da sich Anwendungen inklusive Dateibaum innerhalb eines Applets befinden. Die eigentlichen Kartenkommandos, wie READ BINARY oder MUTUAL AUTHENTICATE, sind als Programmcode innerhalb des Applets enthalten. So ist es dadurch beispielsweise möglich, gleiche Kommandos mit unterschiedlicher Codierung und differierendem Programmablauf innerhalb einer Karte unabhängig voneinander in zwei getrennten Applets zu unterstützen.

Man erkauft sich diesen Vorteil allerdings mit einem erheblichen Speicherplatzbedarf für die jeweiligen Applets, da diese dann zwangsläufig einige redundanten Daten und Programmcodeteile enthalten. Um diesen eklatanten Speichermehrverbrauch in manchen Fällen etwas abzumildern, ist es möglich, Objekte eines Applets mit anderen Applets zu teilen (*object sharing*). Aus Sicherheitsgründen läßt sich dies nur von dem Applet aus durchführen, das das Objekt auch erzeugt hat. Der Vorgang selber kann nicht mehr rückgängig gemacht werden. Das heißt, wenn der Zugriff auf ein Objekt eines Applets für ein anderes Applet freigegeben worden ist, dann ist dies bis zum Ende des Kartenlebenszyklus der Fall.

Die einzigen appletunabhängigen Kommandos, die noch existieren, dienen zum sicheren Laden von Applets in die Chipkarte. Diese werden dann im EEPROM abgelegt und von der Java Virtual Machine ausgeführt. Lediglich die Übertragungsprotokolle sind einheitlich für die gesamte Chipkarte.

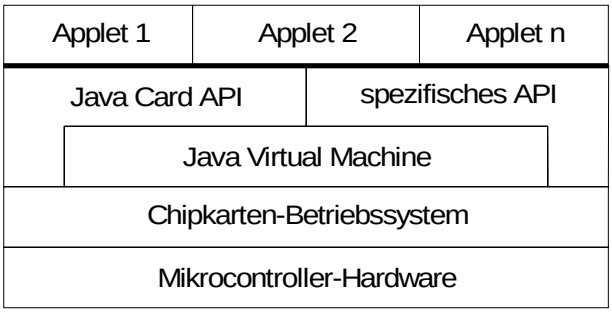

Bild 16 *Der softwaretechnische Aufbau einer Chipkarte mit Java Virtual Machine, zwei verschiedenen APIs und mehreren Applets.*

Softwareentwicklung für Java auf Chipkarten

Wie ist nun die Vorgehensweise, um ein Java-Programm für eine Chipkarte zu erstellen und zur Ausführung zu bringen? Als erstes erstellt der Programmierer mit einem Texteditor den eigentlichen Java Source Code. Anschließend compiliert er diesen mit einem beliebigen Java Compiler und erhält den maschinenunabhängigen Bytecode. Bis zu diesem Punkt ist der Ablauf identisch dem der Java-Programmierung für PCs.

Der Bytecode wird nun dem Offcard-Teil der Java Virtual Machine (d.h. Offcard-VM) als Class-Datei übergeben, welche Prüfungen des Formats, der Syntax, von Feldreferenzen und ähnlichem durchführt. Sind alle Prüfungen mit positivem Ergebnis abgeschlossen, dann erstellt die Offcard-VM eine sogenannte CAP-Datei (*card appplication file – CAP-file*). Diese wird je nach Anwendung und Bedarf noch mit einer digitalen Signatur versehen, um damit sicherzustellen, daß es von der Offcard-VM geprüft wurde und authentisch ist. Wäre keine überprüfbare Signatur vorhanden, dann könnte man mit einem manipulierten Applet die Sicherheit der Oncard-VM umgehen, denn diese kann aus Speicherplatzgründen nicht alle Prüfungen durchführen. Anschließend wird das Applet im Format der CAP-Datei in die Chipkarte geladen. Diese überprüft als erstes die, in der Regel vorhandene, digitale Signatur und übergibt das kontrollierte Applet der Oncard-VM. Was nun folgt, verhält sich wieder weitgehend analog der Programmausführung auf einer Virtual Machine auf einem PC. Die Oncard-VM prüft und interpretiert Zeile für Zeile des Bytecodes und erzeugt daraus Maschinenbefehle für den Prozessor der Chipkarte.

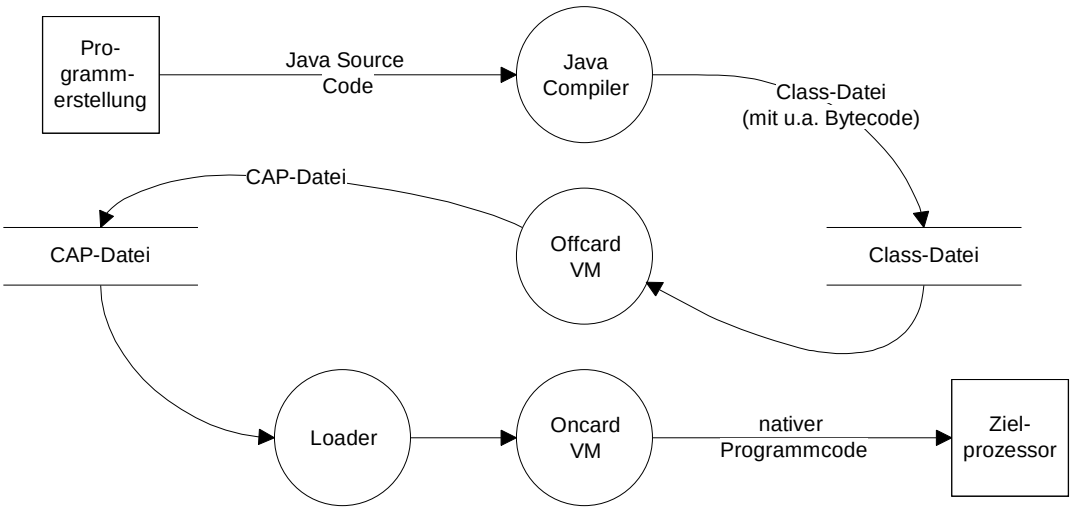

Bild 17 *Der übliche Ablauf von der Programmentwicklung bis zur Ausführung des Programms auf der Java Virtual Machine (JVM) auf dem Chipkarten-Mikrocontroller.*

Resümee und Zukunft

Trotz des immer noch anhaltenden Hype um Java darf man nicht vergessen, daß sie sicherlich nicht Abhilfe für alle IT-Probleme der letzten und nächsten Jahre bieten wird und die in sie gesteckten Erwartungen unter Umständen nicht alle erfüllt werden. Man denke dabei nur an die vergangenen, mittlerweile aus der Mode gekommenen Sprachen wie Pascal („modular“), Lisp („KI für alle“), C („portabel“) und C++ („wiederverwendbarer Programmcode“). Diese Sprachen haben zwar die Informationstechnik um Größenordnungen weitergebracht, doch viele der prognostizierten positiven Effekte traten nie ein. Es kann jedoch als sicher angenommen werden, daß Java der Beginn einer neuen Chipkarten-Ära ist, da es erstmals für jedermann möglich ist, ausführbaren Programmcode in Chipkarten ablaufen zu lassen. Ob es langfristig Java sein wird, das die Standard-Programmiersprache für Chipkarten ist, muß abgewartet werden. Dies wird der Markt anhand der Leistungsdaten und die Geschäftspolitik einiger Firmen entscheiden.