Universität Rostock

Fakultät für Informatik und Elektrotechnik
Institut für Informatik
Lehrstuhl für Rechnerarchitektur
Prof. Dr.-Ing. habil. Djamshid Tavangarian

Studienarbeit

Erweiterung der Kommunikationsmöglichkeiten des ARM-Internet-Servers um den USB auf der Basis einer SmartCard-Schnittstelle

von Stephan Hüls

Betreuer : Dipl.-Inf. Ulrike Lucke, Dipl.-Inf. Daniel Versick
Datum : 29.09.2004

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit soll eine Erweiterung der Kommunikationsmöglichkeiten des ARM Internet Server um eine SmartCard Schnittstelle realisiert werden.

1.1 Motivation

Die Nutzung von Krypto-Tokens auf Basis des Universal Serial Busses bekommt durch die starke Verbreitung des USB eine immer größere Bedeutung. Vorhandene USB-Kryptotokens haben vielfach den Nachteil, dass sie keine offenen Systeme darstellen. Eine Veränderung der Software auf den Systemen und damit eine bessere Anpassung an spezifische Anwendungen ist nicht möglich. Da dies in einigen Fällen aber durchaus wünschenswert ist, soll es Ziel dieser Arbeit sein, erste Grundlagen für die Entwicklung eines frei in Java programmierbaren USB-Kryptotokens zu erarbeiten.

1.2 Aufgabenstellung

Inhalt der Studienarbeit ist es, eine Kommunikation des am Lehrstuhl für Rechnerarchitektur entwickelten ARM-Internet-Servers mit anderen USB-tauglichen Geräten zu ermöglichen. Diese Hardwareplattform basiert auf einem ARM7TDMI-Kern der Firma Samsung und kann später als Hardwaregrundlage für die Entwicklung einer leistungsfähigen Smartcard dienen. Der ARM-Internet-Server soll mittels seiner General-Purpose-I/O-Ports mit einem SmartCard-Reader in Form eines USB-Tokens verbunden werden und so eine Kommunikation über den USB ermöglichen.
Die am Lehrstuhl für Rechnerarchitektur für den Einsatz in eingebetteten Systemen entwickelte Kertaserie VM soll als Java-Laufzeitumgebung dienen. Diese Arbeit unterteilt sich in die folgenden Teilaufgaben.

- Verfügbare USB-Tokens sind für die Nutzung im Hinblick auf diese Arbeit zu untersuchen. Das exemplarisch ausgewählte USB-Token ist auf geeignete Weise mit den GPIO-Ports des ARM-Servers zu verbinden.
- Die Kertasarie VM soll um geeignete Schnittstellen erweitert werden, die eine Kommunikation des ARM-Internet-Servers mit dem USB-Token über ein smartcardspezifisches Protokoll ermöglichen. Es können dabei wahlweise das T=0 oder das T=1 Protokoll verwendet werden.
- Es sind geeignete Testprogramme für Host und Target zu implementieren, die die Funktionsfähigkeit des implementierten Protokollstacks

demonstrieren. Anschließend ist die Leistungsfähigkeit der erstellten Verbindung unter denkbaren Anwendungsszenarien zu messen.

Die in der Arbeit erzielten Ergebnisse sind zu analysieren, zu diskutieren sowie schriftlich zu dokumentieren und im Rahmen des Forschungsseminars des Lehrstuhls für Rechnerarchitektur vorzustellen.

2 Grundlagen

In Absatz 2 wird auf die Themen SmartCards, ARM Internet Server und Universal Serial Bus eingegangen.

2.1 SmartCards

Schon 1968 hatten die deutschen Erfinder Jürgen Dethloff und Helmut Grötrupp die Idee, einen VLSI-Chip in einer Plastikkarte zu integrieren [6]. Die kommerzielle Nutzung begann aber erst in den achtziger Jahren durch die Einführung der Telephonkarten in Frankreich. Die Plastikkarten mit integriertem Chip werden als SmartCards bezeichnet. SmartCards werden in der Regel als sicherer Aufbewahrungsort für geheime Daten und als sichere Ausführungsplattform für kryptographische Algorithmen verwendet. Die Sicherheit ergibt sich durch den Aufbau der Hardware und die definierte Kommunikationsschnittstelle. Die Karte ist so aufgebaut, dass ein Verändern oder Auslesen der Daten durch Unbefugte nicht möglich ist. Typische Einsatzbereiche sind das Speichern von Kundendaten oder die Authentifikation an Bankautomaten.
Im SmartCard-Bereich wird zwischen Speicherkarten und Prozessorkarten unterschieden. Speicherkarten sind nur zum Ablegen von Daten gedacht. Auf Prozessorkarten können zusätzlich Programme ausgeführt werden. Abbildung 1 zeigt den grundsätzlichen Aufbau einer Prozessorkarte.

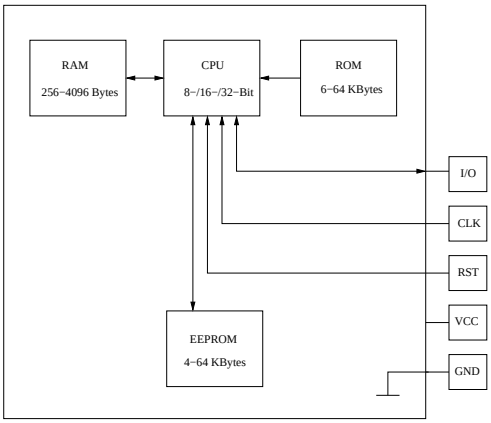

Abbildung 1: Aufbau einer Prozessorkarte

Der RAM dient als Hauptspeicher für das System. Im EEPROM werden die Daten und Programme für die Applikationen abgelegt. Das ROM enthält das Betriebssystem. Die CPU wird verwendet, um Programme auszuführen. Die Karte stellt 5 Pins zur Verfügung. Der I/O-Pin ist für die serielle Datenübertragung verantwortlich. An den CLK-Pin ist von außen der Takt der Karte anzulegen. Über RST ist ein Reset ausführbar. An VCC muss die Betriebsspannung gegenüber GND angelegt werden. Abbildung 2 zeigt zwei übliche SmartCards im Standard und SIM Format.

Abbildung 2: SmartCards

2.1.1 Protokolle

Im SmartCard Bereich sind eine Vielzahl von Übertragungsprotokollen anzutreffen. Die Protokolle ermöglichen eine Kommunikation zwischen der SmartCard und einem Kartenlesegerät. Die am meisten verbreiteten Protokolle sind das T=0 und T=1 Protokoll. Das T=0 Protokoll ist byteorientiert, während das T=1 Protokoll blockorientiert ist.
In der Aufgabenstellung ist die Auswahl des SmartCard-Kommunikationsprotokolls freigestellt. Für die Implementierung wurde das T=0 Protokoll ausgewählt. Das Protokoll ist einfacher zu implementieren, der Standard ist frei verfügbar und es reicht für die Übertragung zu einem USB Token in Bezug auf die Datenübertragungsrate aus. Das in Frankreich entwickelte Protokoll ist sehr verbreitet. In GSM Mobiltelephonen wird es ausschließlich für die Kommunikation zwischen Handy und SIM Card verwendet. Es ermöglicht eine standardisierte Datenübertragung zwischen SmartCards und SmardCard-Terminals. Das T=0 Protokoll ist in ISO 7816 standardisiert. Der Standard für das T=1 Protokoll ist nicht zugänglich.

2.1.2 Schichtenmodell

Die Datenübertragung im SmartCard Bereich kann in einem Schichtenmodell abstrahiert werden. Das Schichtenmodell lehnt sich an das ISO-OSI Standardmodell für Kommunikationsprotokolle an. Für die relativ einfache Übertragung wurden aber nur 3 Schichten übernommen. Eine exakte Trennung der Schichten, wie es das OSI-Modell fordert, ist hier nicht umgesetzt. Außerdem werden nur drei Schichten des sieben schichtigen OSI-Modells verwendet. Abbildung 3 zeigt das dreischichtige Modell. Die Bitübertragungsschicht ist unter anderem für das Timing und die Abtastung der Leitung bei kontaktbehafteten Karten zuständig. Bei kontaktlosen Karten muß hier die Übertragung von Bits über einen leitungslosen Kanal zur Verfügung gestellt werden. Die Betrachtungen in dieser Arbeit beziehen sich alle auf eine kontaktbehaftete Verbindung
Auf Leitungsschicht stehen verschiedene Protokolle zum Datenaustausch zur Verfügung.
Auf Anwendungsebene werden APDU's (Application Protocol Data Unit) ausgetauscht. Diese sind unabhängig vom verwendeten Protokoll der Schicht darunter.

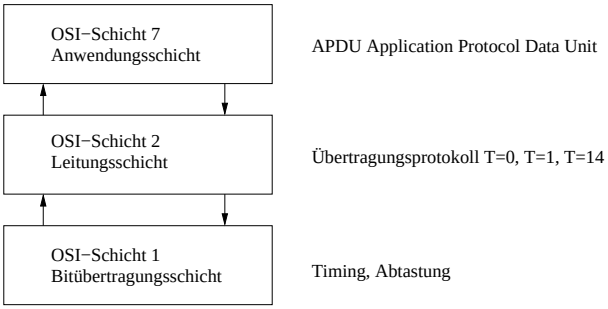

Abbildung 3: Schichtenmodell für SmartCards

Physikalische Übertragung:
Die Bitübertragungsschicht beschreibt die Übertragung der Bits auf einem physikalischen Kanal. Der Austausch der Daten geschieht bitseriell. Die Bits werden der Reihe nach übertragen. Die logischen Zustände eines Bits sind durch die Potentiale 0 Volt und 5 Volt auf der Leitung repräsentiert. Direct convention bedeutet, dass logisch 0 dem Potential 0 Volt und logisch 1 dem Potential 5 Volt entspricht. Inverse convention legt fest, dass für logisch 0 ein Potential von 5 Volt und für logisch 1 ein Potential von 0 Volt auf der Leitung liegen muss. Welche Convention verwendet wird, kann die SmartCard festlegen (siehe Absatz 2.1.3). Im Ruhezustand liegt die Datenleitung auf 5 Volt. Die Reihenfolge der Bits ist auch durch die Convention festgelegt. Bei direct convention ist das erste übertragene Bit das niederwertigste Bit des Bytes. Bei inverse convention wird das höchstwertige Bit als erstes gesendet. Ein Kommunikationspartner darf um logisch 1 zu senden kein 5 Volt Potential auf die Leitung legen. Bei Kommunikationsstörungen könnte dadurch ein Kurzschluss entstehen, wenn ein Kommunikationspartner eine logische 0 und der andere eine logische 1 sendet. Um das zu vermeiden dürfen SmartCard und Kartenleser je nur das 0 Volt Potential auf die Leitung legen oder sich in Bezug auf die Leitung hochohmig schalten. Abbildung 4 verdeutlicht den Sachverhalt.

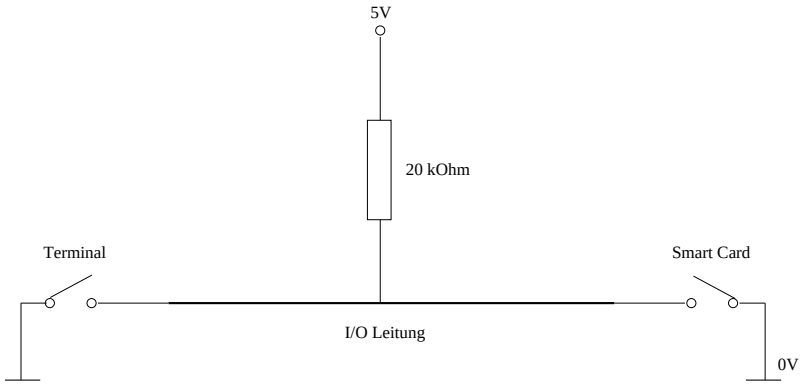

Abbildung 4: Elektrische Übertragung bei SmartCards

Der 20 kOhm Widerstand, der sich im Lesegerät befindet, zieht die Leitung auf 5 Volt, falls sich die SmartCard und der Karten-Leser hochohmig schalten. Eine logische 0 auf die Leitung zu legen, bedeutet also, das Massepotential durchzuschalten. Eine logische 1 kann gesendet werden, indem der Kommunikationspartner seinen Ausgang hochohmig schaltet. Dabei zieht der Widerstand das Leitungspotential auf 5 Volt. Das ist durch die Schalter auf beiden Seiten angedeutet.
Das T=0 Protokoll ist asynchron und byteorientiert. Byteorientiert bedeutet, dass nacheinander Bytes übertragen werden. Bei einer asynchronen Übertragung ist die Übertragungsgeschwindigkeit bei Sender und Empfänger bekannt. Eine weitere Leitung zur Synchronisation wird nicht benötigt. Die Synchronisation zwischen Sender und Empfänger geschieht über das Verwenden von Start- und Stopbits. Asynchrone Übertragungen sind in der Regel langsamer als synchrone, haben aber den Vorteil, dass nur eine Leitung für die Datenübertragung gebraucht wird. Abbildung 5 zeigt die Übertragung eines Bytes mit Start- und Stopbits in direct convention.

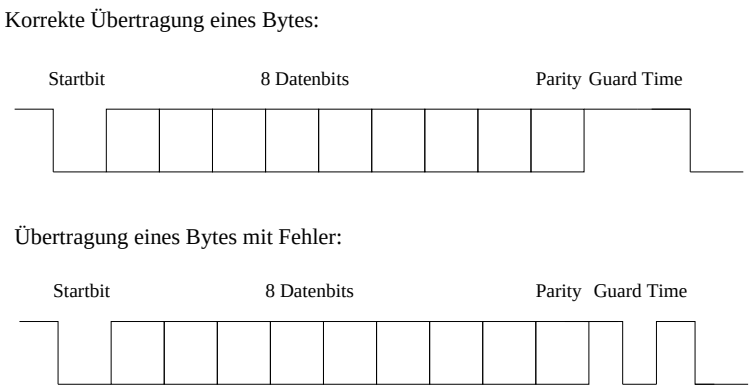

Abbildung 5: Übertragung eines Bytes beim T=0 Protokoll

Im Ruhezustand befindet sich die Leitung auf logisch 1. Die Übertragung eines Bytes wird durch ein Startbit eingeleitet, das logisch 0 entspricht. Danach werden die 8 Datenbits übertragen. Anschließend folgt ein Paritätsbit. Im SmartCard Bereich wird gerade Parität verwendet. Das bedeutet, dass die Anzahl der Einsen der Datenbits auf einen geraden Wert erweitert werden. Ist die Anzahl der Einsen gerade, so ist die Parität gleich 0. Bei ungerader Anzahl der Einsen der Datenbits ist die Parität 1. Das macht es möglich Einzelbitfehler zu erkennen. Zweifachfehler werden damit nicht erkannt. Nach der Parität werden Stopbits übertragen. Diese sind logisch 1. Die Stopbits werden auch als guard time bezeichnet. In der guard time haben Sender und Empfänger Zeit um sich auf die Übertragung bzw. das Empfangen des nächsten Bytes einzurichten.
Wird eine falsche Parität erkannt, so signalisiert der Empfänger das durch Herunterziehen der Datenleitung auf logisch 0 während der guard time. Falls das passiert ist, wird das gleiche Byte erneut gesendet. Es ist zulässig, die Signalisierung ein halbes Bit nach dem Stopbit durchzuführen. Da dies von einem RS232 UART nicht erkannt werden kann, ist das T=0 Protokoll mit einem Standardbaustein für serielle Übertragungen nicht abtastbar.
Die Dauer eines einzelnen Bits kann für eine SmartCard nicht direkt angegeben werden. Die Bitdauer hängt vom angelegten Takt und von einem Teiler ab. 1 Bit wird als 1 etu (elementary time unit) definiert. Der Teiler gibt die Anzahl der Takte pro Bit an. Im wesentlichen werden die Teiler 372 und 512 verwendet. Der Takt dividiert durch den Teiler ergibt die Übertragungsgeschwindigkeit. Bei den Standardfrequenzen 3,5712 und 4,9152 MHz ergibt sich die übliche Datenrate von 9600 Bit/s. Laut Standard ist eine Maximalfrequenz von 10 MHz und ein minimaler Teiler von 32 erlaubt. Das ergibt eine maximale Datenrate von 312.500 Bit/s. Ein Bit darf bei der Übertra-

gung eine maximale Abweichung von +-0,2 etu haben.
In [6] wird eine Dreifachabtastung vorgeschlagen. Ein Abtastpunkt liegt genau in der Mitte des Bits. Mit Berücksichtigung der zulässigen Abweichung liegen die beiden anderen Punkte 0,15 etu vor und nach der Bitmitte. Nach dem Byzantinischen Mehrheitsentscheid wird aus den drei Abtastungen der Bitwert abgeleitet. Dadurch ist eine stabile Kommunikation zu realisieren. Über die Abtastung wird allerdings in der Norm keine Aussage gemacht. Abbildung 6 zeigt die Dreifachabtastung.

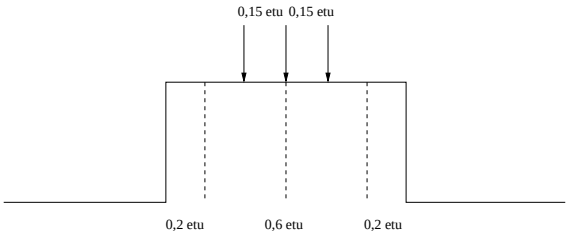

Abbildung 6: Dreifachabtastung eines Bits

Leitungsschicht:
Die Leitungsschicht wird über das Kommunikationsprotokoll beschrieben. Hier kommt das T=0 Protokoll zum Einsatz. Das T=0 Protokoll ist asynchron und byteorientiert. Die Bytes werden so wie im Abschnitt zuvor übertragen. Die Kommunikation läuft nach dem Master-Slave Verfahren ab. Ein Datenaustausch kann immer nur vom Kartenlesegerät initiiert werden. Abbildung 7 verdeutlicht das Verfahren.

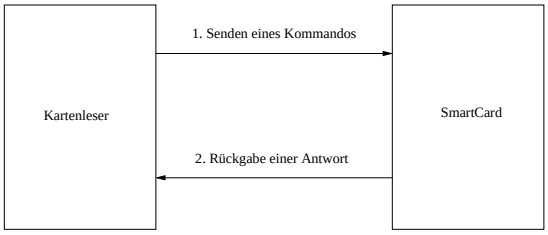

Abbildung 7: Master-Slave Verfahren bei SmartCards

Außerdem kann zu einem Zeitpunkt nur in einer Richtung gesendet werden. Es handelt sich also um eine Halbduplex-Verbindung. Liegt ein Paritätsfehler vor, so wird das aktuelle Byte erneut übertragen.
Die zu übertragenden Daten sind auf der Protokollebene in TPDUs gekapselt. TPDU steht für Transport Protocol Data Unit. Abbildung 8 zeigt den

Aufbau von Command TPDU und Response TPDU.

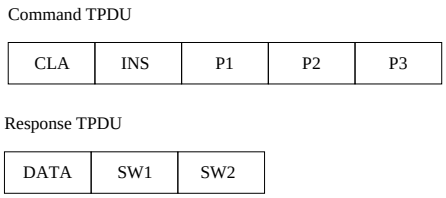

Abbildung 8: Aufbau einer Kommando TPDU des T=0 Protokolls

Jedes Kästchen stellt ein Byte dar. Mit dem Class Byte (CLA) wird beschrieben, um welche Art von Anwendung es sich handelt. Class Bytes mit dem Inhalt 0x00 stehen für die normierten Befehle nach ISO 7816, 0x80 für die elektronische Geldbörse und 0xA0 für die Befehle im GSM Bereich. Das Instruction Byte (INS) steht für einen Befehl in der Anwendung. Vergleichbar ist das mit dem Aufruf einer Funktion innerhalb eines Programms. Mit dem Unterschied, dass hier die Funktionen mit einem Byte und nicht mit einem Funktionsnamen codiert sind. Mit den Parameter Bytes P1 und P2 können zu den Befehlen Parameter ähnlich wie bei einem Funktionsaufruf übertragen werden. P3 steht für die Länge der Daten, die mit dem Kommando zu übertragen sind. Falls ein Kommando ohne Daten aufgerufen werden soll, so ist für P3 0x00 zu übertragen.
Die Response TPDU enthält immer die zwei Antwort-Bytes SW1 und SW2. Damit kann signalisiert werden, ob das Kommando fehlerfrei oder nicht ausgeführt wurde. Bei korrektem Ausführen des Befehls wird standardisiert SW1=0x90 SW2=0x00 übertragen. Bei Fehlern werden definierte Antworten erzeugt. SW1=0x6D SW2=0x00 steht beispielsweise für "Befehl wird nicht unterstützt". Es ist also ein Kommando mit einem unbekannten Instruction Byte übertragen worden.
Es ist nicht möglich, dass eine SmartCard unmittelbar nach einer Command TPDU mit Daten antwortet. Für die Datenübertragung zum Terminal ist ein spezieller Befehl Get Response definiert. Eine Karte verarbeitet also eine Anfrage und stellt die eventuell entstandenen Rückgabedaten zur Verfügung, die dann mit einem Get Response Befehl abgeholt werden können.
Das ist der Grund, warum zwei Antwort Bytes definiert sind. Mit SW2 kann die SmartCard signalisieren, dass Daten erzeugt wurden. Nach ISO 7816 ist die Antwort mit SW1=0x61 und SW2=0xXX festgelegt. 0xXX steht für die Länge der Daten. Der Get Response Befehl hat das Instruction Byte 0xC0. Bei Aufruf dieses Befehls können die Daten der SmartCard vor SW1 und

SW2 übertragen werden. Abbildung 9 zeigt den Ablauf des T=0 Protokolls.

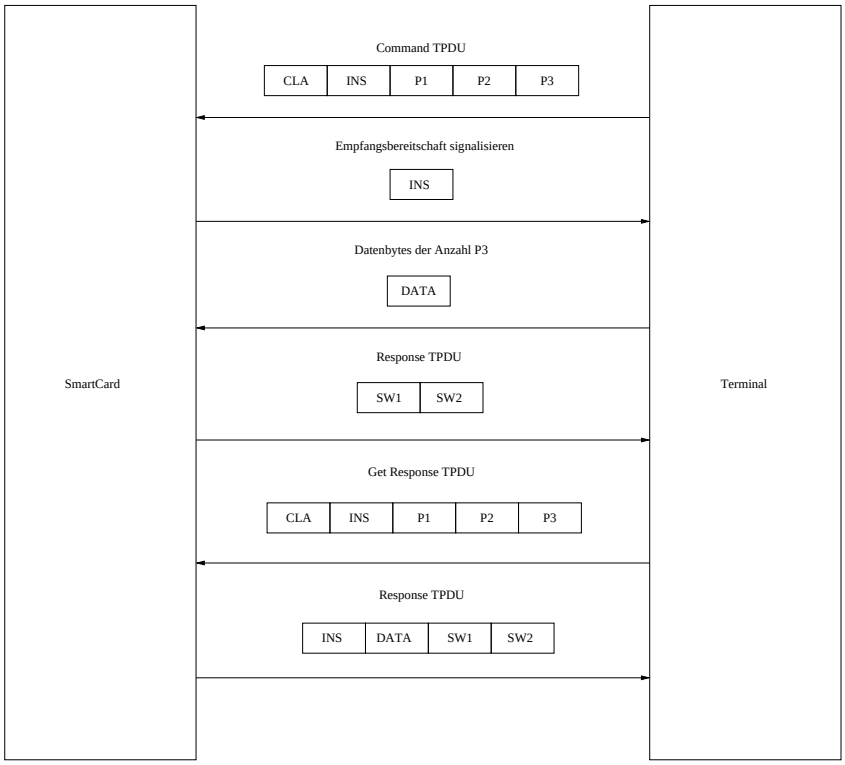

Abbildung 9: Ablauf des T=0 Protokolls

Auf Leitungsebene werden die Bytes einer Command TPDU nicht unmittelbar als Bytes zu einer SmartCard gesendet. Zuerst werden lediglich die 5 Bytes der TPDU übertragen. Dann muss die SmartCard die Empfangsbereitschaft signalisieren. Das geschieht durch das Senden des INS Bytes als Acknowledge. Dann kann das Terminal die Daten byteweise auf die Leitung legen. Nach dem Einlesen der Bytes muß die SmartCard einen 2 Byte Return Code erzeugen.
Falls die SmartCard nach Abarbeitung des Kommandos Ergebnisdaten erzeugt hat, so signalisiert sie das mit dem Antworten eines speziellen Return Codes. Mit dem Get Response Befehl können dann die Daten vom Terminal geholt werden. Das Terminal schickt das Get Response Kommando. Danach antwortet die Karte mit den Daten und den 2 Antwort Bytes als Bytefolge. Es ist darauf zu achten, dass das erste Byte der Daten dem INS Byte der Get Response TPDU entsprechen muss. Auf Anwendungsebene erscheinen

die INS Bytes zur Synchronisation der Kommunikation nicht mehr.
Um eine vollständige Kommunikation auf der Leitungsebene umzusetzen, sind lediglich zwei Befehle und die Antwort TPDU nötig. Ein Command Befehl um Anfragen an die SmartCard mit und ohne Daten zu schicken, ein Get Response Befehl um Daten von der SmartCard abzuholen und die Antwort TPDU mit und ohne Daten um die Befehle zu bestätigen. Reale Anwendungen definieren natürlich mehrere Befehle um die unterschiedlichen Funktionen auf der SmartCard direkt ansprechen zu können. Im Anhang sind die standardisierten Befehle und Antwortkodierungen aufgeführt. Bei der Entwicklung eigener Anwendungen ist es auch möglich, eigene Kommandos und Antwort Codes zu definieren. Es ist aber sinnvoll, falls ausreichend, die standardisierten Codierungen zu verwenden.
Für die INS Bytes sollen nur gerade Werte verwendet werden. Mit einem Acknowledge INS+1 kann die Programmierspannung für das EEPROM eingeschaltet werden. Das war aber nur in früheren Anwendungen nötig. Die heutigen SmartCards verwenden diesen Mechanismus nicht mehr.
Das T=0 Protokoll ermöglicht auch den Einzelempfang der Bytes vom Terminal. Dazu muss das invertierte INS Byte als Acknowledge von der SmartCard gesendet werden. Das Terminal sendet dann nur ein Byte und wartet auf ein weiteres Acknowledge der Karte. Das geht so lange bis alle Daten übertragen sind, oder die SmartCard das nicht invertierte INS Byte sendet Dann schickt das Terminal die verbleibenden Bytes als Folge. Dieser Modus ermöglicht eine sehr stabile Übertragung, die in der Regel aber nicht benötigt wird.
Es ist zu erkennen, dass das T=0 Protokoll die klare Trennung der Schichten, wie im OSI Modell gefordert, nicht einhält. Das INS Byte, das auf Anwendungsebene definiert ist, wird für die Synchronisation auf der Leitungsebene verwendet. Es darf also nicht verändert werden. Das macht Probleme bei der Verschlüsselung eines T=0 Kanals. Eine direkte Verschlüsselung ist nicht möglich.
Anwendungsschicht:
Hier kommt die zweistufige Übertragung von Daten mit dem Get Response Kommando nicht mehr vor. Auf eine Anfrage kann direkt eine Antwort mit Daten folgen. Die Applikation, die den Kartenleser anspricht, hat die Aufgabe, die Befehle, die für das T=0 Protokoll nötig sind, zu generieren.
Auf Anwendungsebene werden die Daten in APDUs ausgetauscht. APDU steht für Application Protocol Data Unit. Eine APDU hat den allgemeinen Aufbau wie in Abbildung 10.

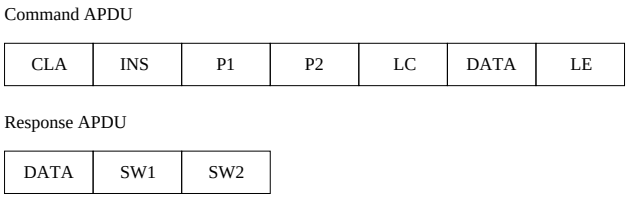

Abbildung 10: APDU auf Anwendungsebene

CLA, INS, P1 und P2 haben die gleiche Bedeutung wie bei dem T=0 Protokoll beschrieben. Statt P3 wird hier von einem Length Count Byte (LC) Byte gesprochen. Das LC Byte entspricht der Länge der zu sendenden Daten. Das LE Byte steht für Length Expectet. Es definiert die Anzahl der Datenbytes, die bei der Response APDU erwartet werden. Es ergeben sich insgesamt 4 verschiedene Kommunikationssituationen. Abbildung 11 zeigt die Varianten.

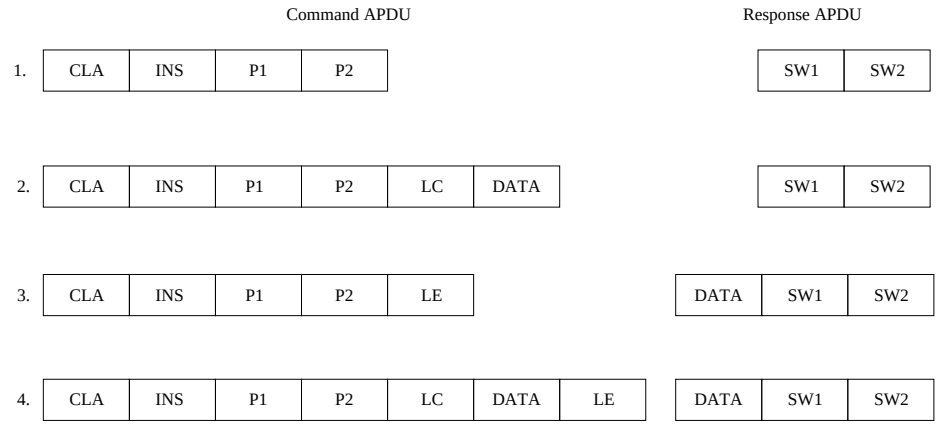

Abbildung 11: Varianten der Kommunikation auf Anwendungsebene

Zu jeder Command APDU ist die Response APDU angegeben. Im ersten Fall wird ein Kommando ohne Daten geschickt und eine Antwort ohne Daten erwartet (LC=0, LE=0). Mit dem zweiten Befehl werden Daten an die Karte geschickt und eine Antwort ohne Daten erzeugt (LC!=0, LE=0). Der dritte Befehl ist ohne Daten, es werden aber Daten in der Antwort erwartet (LC=0, LE!=0). Mit dem letzten Befehl werden Daten an die Karte übertragen und Daten als Antwort gesendet (LC!=0, LE!=0).

2.1.3 Answer To Reset

Nachdem der Kartenleser einen Reset ausgeführt hat, sendet die SmartCard einen Answer To Reset (ATR). In der ATR werden Informationen über die

Chipkarte und Übertragungsparameter übermittelt.
Die ATR darf aus maximal 33 Byte aufgebaut sein. Die Übertragung geschieht immer mit dem Teiler 372. Dadurch ist für beide Kommunikationspartner die Übertragungsgeschwindigkeit, für diesen ersten Austausch von Daten, bekannt.
Die Karte muß mit dem Senden der ATR frühstens nach 400 Takten und spätestens vor 40000 Takten nach dem Reset beginnen. Das ergibt bei einem Standardtakt von 3,5712 MHz ein Intervall von 112 usec bis 11,20 msec.
Zwischen den Bytes ist es der Karte erlaubt maximal 9600 etu zu warten. Bei der Standardfrequenz wie oben ergibt sich dafür genau 1 sec. Die SmartCard kann diese Zeit nutzen um z.B. das Betriebsystem zu booten. Bei einer maximalen Verwendung der Bytelänge ist also eine Wartezeit von 32 sec erzeugbar (für 3,5712 MHz), was für eine Anwendung aber wohl aus Performance-Gründen nicht gebraucht wird.
Abbildung 12 zeigt den allgemeinen Aufbau einer ATR.

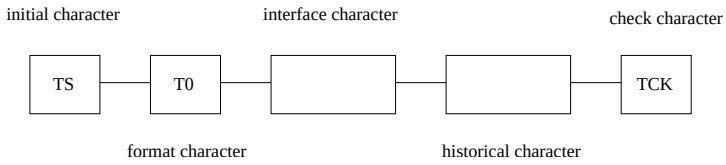

Abbildung 12: Allgemeiner Aufbau einer ATR

Das Byte initial character dient dem Anzeigen der convention für die Bytes. 0x3B steht für direct convention, 0x3F bedeudet, dass inverse convention zu verwenden ist.
Das Byte format character legt fest, ob interface characters folgen und wieviele historical characters übertragen werden. Die vier Bits des oberen Halbbyte stehen für die interface character TA1, TB1, TC1 und TD1. 1 bedeutet, dass Byte wird übertragen, 0 bedeutet, dass Byte ist nicht Teil der ATR. Die vier Bits des unteren Halbbytes definieren die Anzahl der zu übertragenden historical characters.
Die interface character beschreiben die Übertragungsparameter für die Kommunikation. Werden die interface character in der ATR weggelassen, so werden Standardparameter zum Datenaustausch verwendet. Die Standardparameter sind Terminal und SmartCard bekannt.
Die Menge der interface character wird weiter in Bytes vom Typ global interface character und specific interface character verfeinert. Global interface character beschreiben grundlegende Protokollparameter und gelten für alle möglichen Protokolle. specific interface character regeln zu verwendende Parameter für ein bestimmtes Protokoll.

Abbildung 13 beschreibt den Aufbau der interface character innerhalb einer ATR.

Abbildung 13: Aufbau der interface characters

TDi beschreibt die Verkettung der jeweils folgenden interface character. Das höherwertige Halbbyte steht für die interface character, das niederwertige beschreibt die Protokollnummer. Ist TDi nicht vorhanden folgen keine weiteres interface Bytes.
TA1 legt im höheren Halbbyte den Teiler den Übertragung fest. Die vier niederwertigen Bit beschreiben den Übertragungsanpassungsfaktor. Damit können über Tabellen die Übertragungsgeschwindigkeiten normgerecht festgelegt werden.
Mit TB1 ist anzeigbar, dass eine Programmierspannung benötigt wird. Zusätzlich ist ein Faktor festlegbar. Dieser Mechanismus wird nicht mehr gebraucht. Die heutigen SmartCards verfügen über Ladungspumpen, die eine Programmierspannung für EEPROMs erzeugen.
Mit TC1 können weitere Stopbits festgelegt werden (extra guard time). Der Inhalt des Bytes legt die Anzahl der Stopbits fest. Mit 0xFF wird die übliche Anzahl von 2 Stopbits auf 1 reduziert.
Die weiteren möglichen interface character sollen hier nicht angegeben werden, da sie das T=0 Protokoll nicht beschreiben. Einzige Ausnahme ist TC2. Damit kann die work wait time erhöht werden. Die work wait time ist die maximale Zeit zwischen den Startflanken aufeinander folgender Bytes.

2.1.4 Protocol Type Selection

PTS steht für Protocol Type Selection. Nach dem Übertragen der ATR, ist es, soweit erlaubt, dem Kartenleser möglich der Karte andere Parameter für die Übertragung vorzuschlagen.
Eingeleitet wird der PTS Datenaustausch durch das Byte 0xFF. Da ein PTS in dieser Arbeit nicht zugelassen wird, findet keine weitere Beschreibung statt. In [6] sind die Regeln für das PTS nachzulesen.

2.1.5 Automat des T=0 Protokolls

Die Punkte 2.1.2, 2.1.3 und 2.1.4 lassen sich als Automat zusammenfassen, der das T=0 Protokoll beschreibt. Abbildung 14 zeigt den Automat.

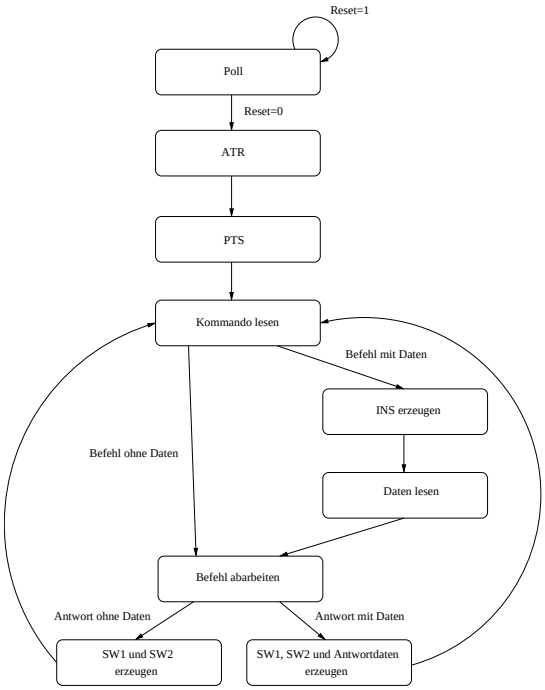

Abbildung 14: Automat des T=0 Protokolls

2.2 ARM Internet Server

Der ARM Internet Server ist eine am Lehrstuhl für Rechnerarchitektur der Universität Rostock entwickelte Hardwareplattform [3]. Als Grundlage dient ein ARM7TDMI-Core in einer Implementierung der Firma Samsung. Dieser Samsung 4530A ist ein ARM-Core mit folgenden Eigenschaften:

- 32 Bit breiter externer Speicherbus
- 10/100 MBit Ethernet -Controller
- bis zu zwei 460 Kbps UART-Kanäle
- DRAM-Controller
- 2 HDLC-Kanäle
- 26 general purpose I/O Ports, von denen aber nicht alle verfügbar sind
- i2C-Bus

- zwei Timer
- JTAG-Schnittstelle
- DMA-Controller
- keine MMU

Neben dem Samsung 4530A besitzt der ‚ARM Internet Server' folgende Komponenten:

- 4 MB Flash-Speicher
- 8 MB SDRAM
- PHY Intel LXT971A und RJ45-Ethernet-Buchse
- 2 Transceiver für die seriellen Schnittstellen
- ein Oszillator, der die Schaltung mit einer Taktfrequenz von 50 MHz taktet

Abbildung 15 zeigt den ARM Internet Server. Die Architektur dient als Basissystem für die Entwicklung eines Smart Token. Bei der Entwicklung des Smart Token ist der Aufbau auf eine kleinere geometrische Abmessung zu integrieren. Unnötige Bauteile, wie der Ethernetstecker oder die Pins für die anderen Schnittstellen können entfallen. Außerdem muss für den ARM Core ein weiterintegrierter Baustein mit kleinerer geometrischer Abmessung verwendet werden.

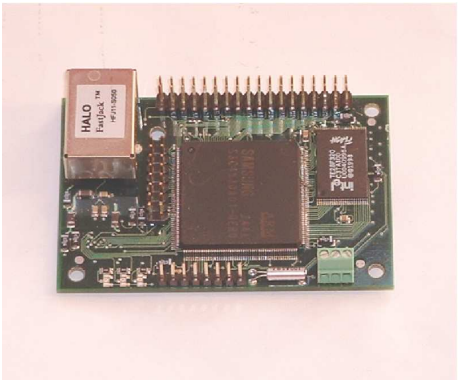

Abbildung 15: ARM Internet Server

2.3 Universal Serial Bus

Der Universal Serial Bus (USB) ist ein Bus zur einfachen Erweiterbarkeit eines Arbeitsplatzrechners mit Peripherie-Geräten [7].
Er zeichnet sich durch folgende Merkmale aus:

- einfache Verbindung unterschiedlicher Peripheriegeräte
- USB ermöglicht Plug and Play
- Anschluss über eine kostengünstige Schnittstelle mit zwei einheitlichen preiswerten Stecker
- Serielle Übertragung über ein abgeschirmtes 4-Draht-Kabel
- Standard-Übertragungsrate (Full-Speed)=12 Mbit/s mit verdrillten Signalleitungen
- Langsamere Rate (Low Speed) von 1.5 Mbit/s für langsamere Geräte (z.B. Maus)
- Spannungsversorgung von Geräten mit niedrigem Leistungsbedarf über das USB-Kabel
- Anschluß von maximal 127 Geräten

Die Struktur des Busses entspricht einem Baum. Abbildung 16 zeigt die Baumstruktur. Die angeschlossenen Geräte entsprechen den Blättern, die Hubs sind Zwischenknoten. Der Wurzelknoten (Root Hub) ist immer im

Host. Die Hubs können in den angeschlossenen Geräten, oder eigenständig realisiert sein. Ohne die Wurzelschicht sind maximal 7 Schichten zulässig. Ein Kabel darf höchstens 5 Meter lang sein. Mit den 7 Schichten ergibt sich eine maximale Ausdehnung des Busses von 35 Meter.

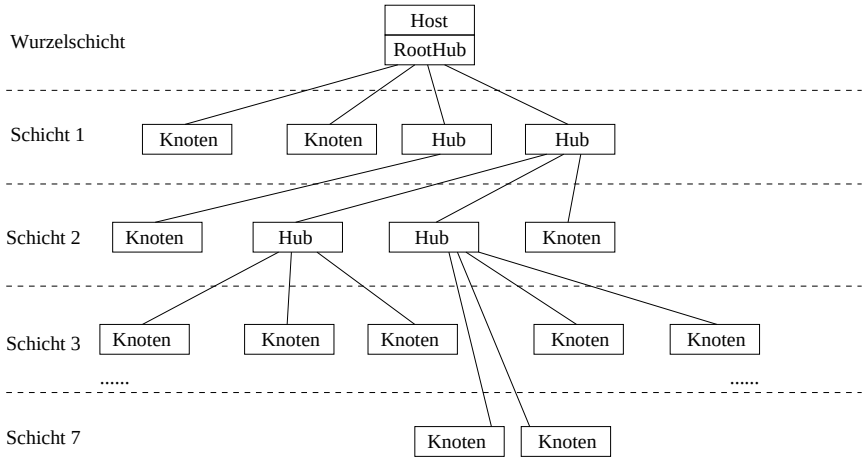

Abbildung 16: USB-Topologie

Die Host-Hardware besteht aus einem USB-Controller, der in einem Chipsatz integriert ist. Die Host-Software hat die Aufgabe, alle angeschlossenen Geräte zu initialisieren, zu nummerieren und zu adressieren. Es wird ein Polling in regelmäßigen Abständen durchgeführt, um Übertragungswünsche zu erkennen. Außerdem überwacht der Host die Betriebsspannung und erkennt das Neuanschließen von Geräten.
Die Hubs sind für die Verzweigungen in der Busstruktur zuständig. Es gibt pro Hub einen Upstream Port für die Kommunikation mit dem Host und mehrere Downstream Ports zur Kommunikation mit den Geräten. Langsame oder volle Übertragungsraten werden erkannt. Bei Änderung der Topologie wird der Host von den betreffenden Hubs informiert. Jeder Hub überwacht die Versorgungsspannung seiner angeschlossenen Geräte. Außerdem haben die Hubs die Aufgabe, die Downstream Ports zu aktivieren, zu deaktivieren oder zurückzusetzen.

Die Endgeräte sind über die Hubs mit dem Host verbunden. Sie bestehen aus mehreren logischen Endpunkten. Es können Geräte mit einer Stromaufnahme von kleiner 500 mA versorgt werden. Zu den Endpunkten sind Übertragungsraten von 12Mbit/s und 1,5 Mbit/s möglich.
Die Datenübertragung geschieht in Form von Paketen. Zur Synchronisation wird die Zeit in konstante 1ms Rahmen unterteilt. Ein Rahmen beginnt mit einem Start-Paket und endet mit einem Zeitintervall konstanter Länge ohne Busaktivität. Jedes Paket fängt mit einem eindeutigen 8-bit-Synchronisationszeichen an.
Adressiert werden die Geräte mit ihren logischen Endpunkten. Die 7 höherwertigen Bits einer 11 Bit Adresse bezeichnen das Gerät, die unteren 4 Bit codieren die logischen Endpunkte. Nach dem Reset hat jedes Gerät die Adresse 0. Danach nummeriert der Host alle Geräte aufsteigend durch. Zu den logischen Endpunkten sind logische Subkanäle (Pipes) aufbaubar. Anwendungen können über Pufferspeicher im Hauptspeicher des Hosts auf die Pipes zugreifen. Dabei können Nachrichten und Stromdaten in die Puffer geschrieben werden. Der logische Subkanal 0 ist für jedes Gerät vorhanden und dient der Übertragung von Steuerinformationen.
Es werden 10 verschiedene Paketformate unterschieden. Ein Datenpaket kann bis zu 1023 Datenbytes übertragen. Jedes Datenpaket enthält ein 16-bit-Prüfzeichen. Weitere Pakete sind das Start-of-Frame-Paket zur Kennzeichnung eines neuen Zeitrahmens, das Token-Paket zur Übertragung der USB-Adresse, die PID-Kennung zur Festlegung der Datenrichtung und dem Kommunikationstyp, ACK um einen fehlerfreien Empfang zu bestätigen, NAK zeigt an, dass kein Senden oder Empfangen möglich ist, STALL zeigt an, dass das Gerät nicht in Betrieb ist und SETUP zur Initialisierung von Geräten.
Es werden unterschiedliche Übertragungsarten mit unterschiedlichen Bandbreiten verwendet. Die isochrone Übertragung garantiert eine konstante Datenrate. Es wird die Übertragungsrate von 12 Mbit/s ohne Fehlererkennung bzw. -korrektur benutzt. Die Bulk-Übertragung ist für größere Datenmengen vorgesehen. Eine Fehlererkennung durch ein Prüfzeichen ist dabei implementiert. Die Bandbreite beträgt 9,7 Mbit/s. Die Übertragung von Interrupt-Daten ist für asynchron auftretende geringe Datenmengen vorgesehen. Eine maximale Latenz, aber kein konkretes Zeitverhalten ist geboten. Auch eine Fehlererkennung ist bei dieser Übertragungsform möglich. Wenigstens 10 Prozent eines Rahmens werden für die Steuerdaten reserviert. Deshalb ergibt sich eine Bandbreite von 6,6 Mbit/s.

3 Ähnliche Arbeiten

Produkte mit ähnlichen Eigenschaften, wie in dieser Arbeit gefordert, sind bereits auf dem Markt verfügbar.

3.1 USB Token

Als ein Vertreter der USB Token soll hier das Token iKey 3000 der Firma Rainbow vorgestellt werden. Abbildung 17 zeigt das Token [10].

Abbildung 17: USB Token Rainbow iKey 3000

Das Token wird verwendet um elektronische Transaktionen und Geschäftsprozesse zu sichern. Dabei wird das herkömmliche Verwenden von Benutzername und Passwort durch eine Public Key Infrastruktur (PKI) erweitert. Es können Zertifikate sicher abgelegt, und Kryptooperationen sicher ausgeführt werden (z.B. Erzeugen eines Schlüssels). Die Firma Rainbow spricht hier von einem Zwei-Faktoren-Authentisierungs Token. Der erste Faktor ist das, was der Benutzer besitzt, nämlich das USB Token. Der zweite Faktor ist das, was der Benutzer kennt, das ist seine PIN. Die mitgelieferte Software bietet für Windows folgende Einsatzzenarien:

- Sichere Authentisierung an ein Windows System
- Sichere Authentisierung für Windows Clients in Firmennetzen
- Sicherer VPN Client Logon für den Remote Zugriff auf das Firmennetz
- Sichere Mail Signatur
- Mail Verschlüsselung mit Microsoft Outlook

In dem 32 KByte Speicher lassen sich bis zu 3 Zertifikate nach X.509 speichern. Der Chip ist gegen einen unerlaubten Zugriff geschützt. Es wird das Betriebssystem STARCOS von Giesecke und Devrient verwendet. Im wesentlichen besteht das Token aus einem USB Controller und dem Prozessor 5032 von Philips.

Zugegriffen wird auf das Token durch eine API des Public Key Chryptographic Standard (PKCS). PKCS beschreibt einen Standard für asymetrische Verschlüsselungen nach dem Public Key Verfahren. Es werden das RSA und Diffie-Hellmann Verfahren unterstützt. Die Beschreibung umfasst die Dokumente PKCS#1 bis PKCS#15. PKCS#3 beschreibt beispielsweise ein Verfahren zur Implementierung des Diffie-Hellman Verfahrens.
Für USBTokens ist PKCS#11 wichtig. Es definiert eine API namens Cryptoki für kryptographische Informationen.
Der iKey 3000 unterstützt die 1024 und die 2048 Bit Verschlüsselung.
Die auf dem Markt verfügbaren Token sind nur durch Schnittstellen auf hohem Abstraktionsniveu programmierbar. Ziel ist es aber am Fachbereich Rechnerarchitektur ein Token zu entwickeln, das als ein offen zugängliches System verwendet werden kann.

3.2 Java SmartCards

Im SmartCard Bereich hat die Programmiersprache Java Einzug gehalten. SmartCards stellen in der Regel die Assembler Befehlssätze der Mikro-Prozessoren Intel 8051 oder Motorola 6805 zur Verfügung. Die langwierige Entwicklung und die schlechte Portabilität von Assembler haben dazu geführt, dass auch Hochsprachen in SmartCards eingesetzt werden. Das vereinfacht die Programmierung erheblich. Java ist besonders geeignet, um eine sichere Ausführungsplattform zur Verfügung zu stellen, da die Java Technologie einen kontrollierbaren Interpreter bereit stellt.
Im ROM der Karte befindet sich neben dem Betriebssystem eine Java Card Runtime Environment (JCRE) zur Ausführung von Java Card Applets [11]. Das JCRE definiert eine Java Card Virtal Machine (VM) und eine Java Card API. Dabei wurde auf die Ressourcenknappheit auf SmardCards geachtet. Das führt zu einer Einschränkung der Funktionalität der VM und zu einer Reduzierung des Sprachumfangs der Java API.

Unterstützt werden:

- Primitive Datentypen: boolean, byte, short
- Eindimensionale Arrays
- Java packages, classes, interfaces, exceptions
- Vererbung, abstrakte Methoden, Überladen, dynamische Objektorientierung, Sichtbarkeit, Binderegeln

- optional 32 Bit Integer

Nicht unterstützt sind:

- long, double, float
- Strings
- mehrdimensionale Arrays
- dynamisches Nachladen der Klassen
- Sicherheits-Manager
- Garbage-Collection (nur teilweise)
- Multithreading
- Objekt Serialisierung
- Klonen von Objekten

Abbildung 18 zeigt das Schichtenmodell einer Java Card Architektur.

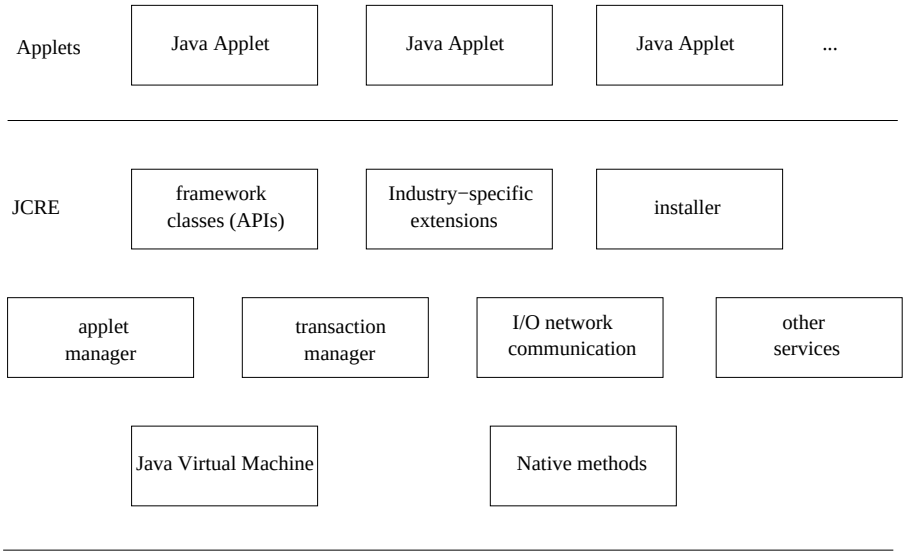

Abbildung 18: Java Card Architektur

Applet manager, transaction management, I/O network communication und other services werden als System Klassen bezeichnet. In ihr wird die Ausführung der Applets und die Kommunikation zwischen Terminal und SmartCard kontrolliert. Die framework classes stellen eine Programmierschnittstelle zur Verfügung. Die API besteht aus den Paketen java.lang, javacard.framework, javacard.security und javacard.crypto. Unter den Erweiterungen können anwendungsspezifische Klassen abgelegt werden. Mit dem Installer können Java Applets auf die Karte geladen werden.
Die Java Card API ist sehr eingeschränkt. Dagegen stellt die Kertasarie VM mehr Funktionalität zur Verfügung. Weiteres Ziel ist es ein Token zu entwickeln, das im Gegensatz zu Java SmartCards leistungsfähiger ist.

4 Realisierungsvorschläge

Das USB Protokoll kann mit den GPIO Ports des Java Servers nicht abgetastet werden. Dazu fehlt die Leistungsfähigkeit. Es ist nötig, ein weiteres Protokoll zu benutzen.

4.1 USB nach RS232 Konverter

Eine Möglichkeit ist einen USB nach RS 232 Konverter zu verwenden. Der Vorteil ist, dass das ARM Board über serielle Schnittstellen verfügt. Die Softwareimplementierung gestaltet sich dabei also als sehr einfach. Bei dieser Lösung ist es allerdings nötig auf dem PC, auf dem das USB Token zum Einsatz kommt, eine Hostsoftware bereitzustellen. Unter Linux wäre eine einfache Kommunikation über eine Konsole mit dem Befehl

```
echo "command:encrypt_text" > /dev/ttyS0
```

möglich. Das ist aber nicht zufriedenstellend. Besser ist es, die Kommunikation in einem Programm mit graphischer Benutzeroberfläche zu abstrahieren. Das muss aber zur Verfügung stehen, falls das USB Token auf einem fremden Rechner zum Einsatz kommt. Möglich wäre es auch, mit dem USB Token über ein signiertes Applet zu kommunizieren. Das Applet kann von einem Web Server geladen werden. Aus dem Browser heraus ist dann eine Kommunikation möglich.

4.2 SmartCard Reader

Eine weitere Möglichkeit ist der Einsatz eines SmartCard Protokolls. Der Java Internet Server wird dabei zur SmartCard und kommuniziert mit einem PC über ein Kartenlesegerät. Für ein USB Token ist ein USB Kartenleser

im SIM Format zu verwenden. Das setzt voraus, dass der Java Internet Server auf eine geometrische Abmessung einer SIM Karte integriert wird. Das Kartenlesegerät kann über die GPIO Pins mit dem Java Internet Server verbunden werden. Die Leistungsfähigkeit der I/O Pins reicht aus, um das T=0 Protokoll direkt auf der Leitung abzutasten. Wie in 2.1.2 erwähnt, kann ein normaler RS232 UART nicht verwendet werden. Denkbar ist auch, ein Hardwaremodul für die Kommunikation zu realisieren. Während des Datenaustausches werden auf der SmartCard keine Befehle ausgeführt. Deswegen tritt keine Leistungseinbuße auf, wenn der Prozessor für die Kommunikation eingesetzt wird. Nachteil bei dieser Lösung ist, dass ein SmartCard Protokoll in Software zu implementieren ist. Grosser Vorteil ist, dass die SmartCard Anwendungen weit verbreitet sind. Es ist wahrscheinlich, auf einem fremden Rechner eine Software anzufinden, die mit einem SmartCard Lesegerät kommunizieren kann. Denkbar ist auch, den integrierten Java Internet Server in beliebigen anderen Lesegeräten einzusetzen. Dabei ist es sinnvoll, SmartCards im SIM- und im Standard-Format herzustellen. Für diese Arbeit wurde die Lösung mit einem SmartCard Reader ausgewählt.

5 Realisierung

In der Realisierung wird auf die Softwarelösung für das PC-Parallelport und den ARM Internet Server eingegangen.

5.1 Anbindung an das PC-Parallel-Port

Die Entwicklung im eingebetteten Bereich ist aufwendiger, als im PC Bereich. Programme werden auf einem PC mit einem Cross Compiler entwickelt. Die Ausführung geschieht auf dem eingebetteten System. Dafür ist ein Upload nötig. Um die Entwicklung des T=0 Protokolls zu vereinfachen, wurde im ersten Schritt eine Verbindung des SmartCard Lesegeräts mit dem Parallel Port des PC's realisiert. Der Vorteil ist, dass nach der Compilation die Software direkt ausprobiert werden kann. Das macht aber eine Portierung auf den Java Internet Server nötig.

5.1.1 Installation des Kobil Readers

Zu Beginn der Arbeit stand ein USB SmartCard Reader der Firma Kobil im Fachbereich zur Verfügung. Alternative Kartenleser wurden nicht in Betracht gezogen, da das Gerät von Kobil alle Anforderungen für diese Arbeit erfüllt. Diese sind:

- ein Standardkartenformat wird verwendet, Abbildung 2 rechte Karte
- die Anbindung an den PC erfolgt über den USB
- das T=0 Protokoll wird unterstützt
- Treiber für alle wichtigen Betriebssysteme stehen zur Verfügung

Auf der beiliegenden CD-ROM sind die Dateien für die Installation abgelegt. Auch ist dort eine vollständige Beschreibung des Kartenlesers zu finden. Auf den Kartenleser kann über die CT-API zugegriffen werden. Diese einfache API besteht nur aus drei Funktionen, die die komplette Kommunikation mit dem Reader erlaubt. Die Funktionen sind im beigelegten Headerfile definiert:

```
char CT_init(unsigned short ctn, unsigned char pn);

char CT_close(unsigned short ctn);

char CT_data (unsigned short ctn,
              unsigned char * dad,       unsigned char * sad,
              unsigned short cmd_len,    unsigned char * cmd_str,
              unsigned short * resp_len, unsigned char * APDU_respstr);
```

Um die Funktionen verwenden zu können, muss die dynamische Library `libct.so` in `/usr/lib` abgelegt werden (oder anderweitig bekannt gemacht werden). Beim ersten Aufruf der API wird im Home-Verzeichnis die Datei `.CTdevices` angelegt. Darin muss, falls nicht automatisch erkannt, das richtige USB Device eingetragen werden. Auf dem Entwicklungsrechner wurde das Device `/dev/ttyUSB0` verwendet. Zu beachten ist, dass der USB Standard erst ab Kernel 2.4.2 unterstützt wird. Mit dem Programm `cardping` lassen sich sehr einfach auf der Konsole APDU's an den Kartenleser senden. Im Punkt 5.1.2 ist der Aufruf zu sehen.

5.1.2 Kommunikation mit einer GSM Karte

Um den Kartenleser auszuprobieren, wurde eine GSM SIM Karte eingelegt. Die GSM Kommandos können als APDU's über das Programm cardping gesendet werden. Abbildung 19 zeigt die Kommunikation.

```
studienarbeit@huels:~/reader> ./cardping -b1 20100101 -b1 A020000108303736 37FFFFFF
FF
Going to call CT_init
CT_init (Port 1): 0
CT_Reset: 0
CT_data Aufruf mit: 20 10 01 01
CT_data: 0
Antwort: 3B BA 94 00 40 14 47 47 33 52 53 37 31 36 53 30 90 01
CT_data Aufruf mit: A0 20 00 01 08 30 37 36 37 FF FF FF FF
CT_data: 0
Antwort: 90 00
```

Abbildung 19: Kommunikation mit einer GSM Karte

Die APDU's werden über den Parameter -b1 an das Programm cardping übergeben. b1 steht für den zu verwendenden Port. Die erste APDU 20100101 veranlasst den Kartenleser einen Reset durchzufüren. Die zweite APDU A02000010830373637FFFFFFFF ist das GSM Kommando für PIN Verify. Hierbei ist die Pin 0767. Die Bytes, die für die PIN übertragen werden, müssen in den höherwertigen vier Bits die Folge 0011 enthalten. Es werden 8 Datenbytes übertragen, die letzten 4 Byte haben den Wert 0xFF.
Nach dem Reset antwortet die Karte mit der ATR der GSM Karte. Das Pin Verify Kommando wird mit dem Return Code 0x90 0x00 bestätigt. Das bedeutet, dass die richtige Pin übertragen wurde.

5.1.3 Versuchsaufbau

Zur Anbindung an das Parallel Port wurden an die Kontakte des Kartenlesers Kabel angelötet. Mit den I/O Pins können für die logischen Zustände die Potentiale 0 Volt und 5 Volt erzeugt werden. Wie in Abbildung 4 zu sehen, darf kein Kommunikationspartner einen aktiven 5 Volt Pegel auf die Leitung legen. Es muss also durch eine Zusatzschaltung der hochohmige Zustand erzeugt werden. Das geht am einfachsten mit einem Transistor. Hier wurde ein Standard-Bipolartransistor (BC 548) verwendet. Abbildungen 20 und 21 zeigen den Versuchsaufbau.

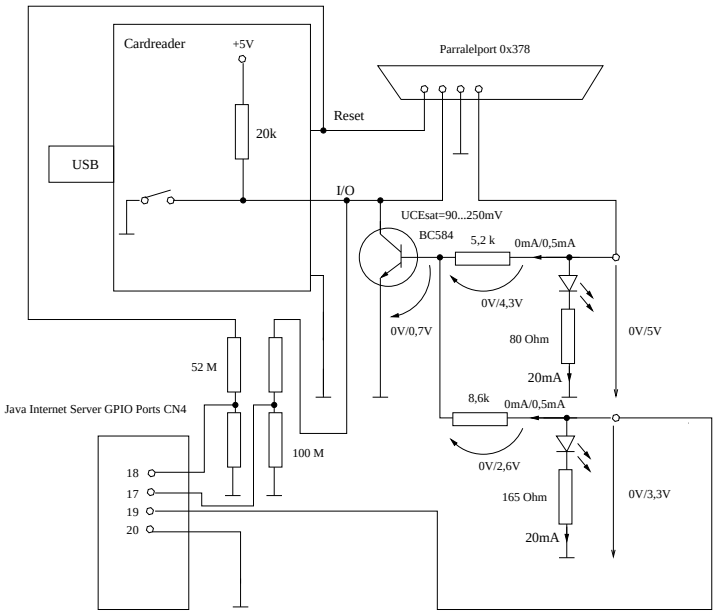

Abbildung 20: Versuchsaufbau

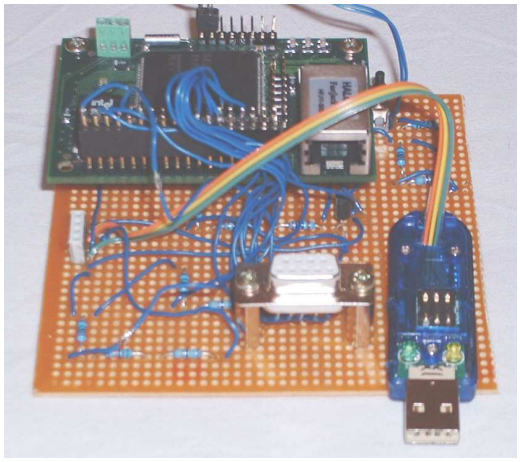

Abbildung 21: Versuchsaufbau

Der Transistor leitet, falls an dem 5,2k Ohm Widerstand die Spannung 5 Volt angelegt wird. Ist der Transistor mit 0 Volt beschaltet, befindet sich das Bauelement im Sperrbereich. Der Transistor schaltet sich zum Kollektor hochohmig. Dabei zieht der Widerstand des Kartenlesers die I/O Leitung auf 5 Volt. Leitet der Transistor, dann wird die I/O Leitung über den Kollektor

Emitterpfad auf Masse gezogen. Der Widerstand ist so dimensioniert, dass sich der Transistor beim Anlegen von 5 Volt im Sättigungsbereich befindet. Der Emitterfolger hat also eine invertierende Wirkung. Das Anlegen von 5 Volt erzeugt auf der I/O Leitung 0 Volt. 0 Volt am Eingang erzeugen auf der Datenleitung 5 Volt. Das muß in der Software durch Invertieren der Bits berücksichtigt werden. Im Schaltplan sind auch die Bauelemente eingezeichnet, die für die Anbindung an den Java Internet Server benötigt werden. Am Datenbit des Parallel Port wurde zusätzlich eine Leuchtdiode angeschlossen. Das macht eine Aktivität auf der Datenleitung sichtbar. Die Leuchtdiode muss mit einer Spannung von 1,7 Volt und mit einem Strom vom 20 mA betrieben werden. Als Datenausgang wird Pin 2 des Parallel Ports verwendet. Für Reset und für den Dateneingang kommen die Pins 12 und 13 zur Anwendung, die als Eingänge geschaltet sind.

5.1.4 Timingproblem

Das T=0 Protokoll ist asynchron. Um das Protokoll in Software zu implementieren, muss die Zeit eines Bits erzeugt werden. Das ist für die Abtastung nötig.
Abbildung 22 zeigt die Aufnahme der Kommunikation mit einem Logik Analysator. Die Auflösung wurde so gewählt, dass die Bitzeit gemessen werden kann. Es ist für ein Bit die Zeit von 120 usec abzulesen. Wie bereits erwähnt ergibt sich die Länge eines Bits aus dem Takt der Kartenlesers und dem Teiler. Der Teiler ist hier 372. Der Kartenleser von Kobil erzeugt für den Takt eine Frequenz von 3,1 MHz. Mit diesem etwas untypischen Wert ergibt sich eine Übertragungsrate von 8333 Baud. Optimal ist es aus dem Takt des Kartenlesers die Bitzeit abzuleiten. Dazu ist es nötig, den Takt abzutasten. Das ist mit dem Parallelport nicht möglich. Dazu wird in erster Näherung die Bitzeit unabhängig vom Lesegerät erzeugt.

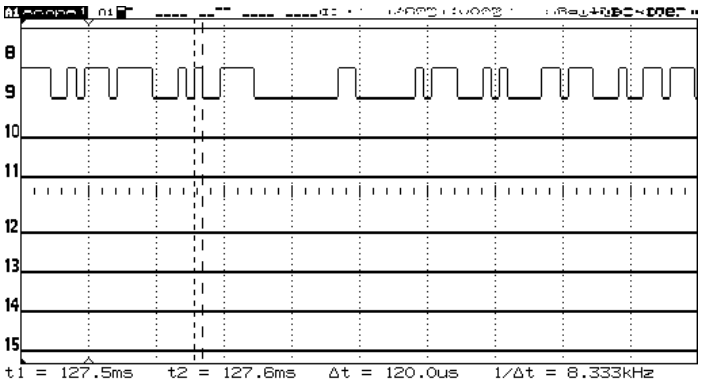

Abbildung 22: Messen der Bitzeit

In C steht die Funktion `usleep(int usec)` zur Verfügung. Damit kann ein Prozess im Mikrosekundenbereich angehalten werden. Falls die Funktion aber oft direkt hintereinander aufgerufen wird, stimmen die Zeiten nicht mehr. Das liegt daran, das durch den Kernelaufruf eine Verzögerung stattfindet. Dieser Verzögerungsfehler pflanzt sich bei wiederholtem Aufrufen fort, was die Funktion für diese Anwendung unbrauchbar macht.
Eine einfache aber sehr effektive Methode ist, die Bitzeit durch eine `for`-Schleife zu erzeugen. Der Nachteil ist allerdings, dass bei einer Portierung des Protokolls die Bitzeit neu auf die Architektur angepasst werden muss.
Das Programm messen.c wird für die Erzeugung der Bitzeit verwendet. Dabei kommt die Methode `gettimeofday(struct timeval *tv, struct timezone *tz)` zum Einsatz, mit ihr können sehr genau Zeiten im Mikrosekundenbereich vermessen werden. Über einen Toggel-Test lässt sich die Bitzeit mit dem Logik-Analysator verifizieren. Für den PC, auf dem die Entwicklung des Protokolls gemacht wurde, ergibt sich für die 120 usec eine Schleife, die von 0 bis 1970 zählt.

5.1.5 Aufnahme des Protokolls

Um das Timing zu testen, wurde das Programm `lauschen.c` geschrieben. Es schneidet die Kommunikation zwischen Kartenleser und GSM-Karte mit. Abbildung 23 zeigt das Ergebnis des Programms.

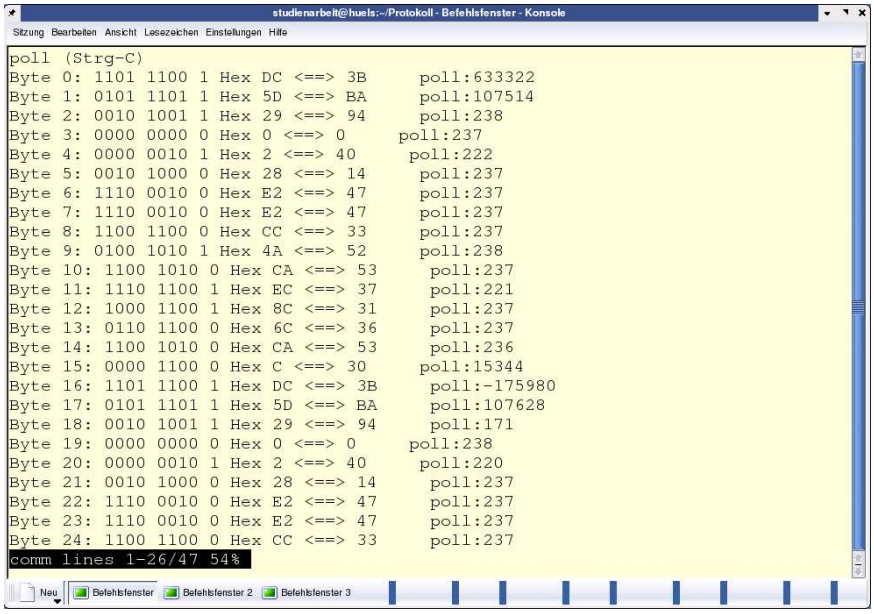

Abbildung 23: Protokollaufnahme

Alle aufgenommenen Bytes werden auf der Konsole ausgegeben. In Abbildung 22 sind nur die ersten 25 Bytes der Kommunikation zu sehen. Die Binärfolgen sind so aufgelistet, wie sie auf der Leitung abgetastet werden. Die Reihenfolge ist umgekehrt zur Reihenfolge der Anwendungsschicht. In der vorletzten Spalte sind die Bitfolgen umgekehrt in hexadezimaler Form angegeben. Der Vergleich mit der Kommunikation aus Punkt 5.1.2 zeigt, dass die Aufnahme richtig ist. Die Bytefolge entspricht der ATR. Es fällt auf, dass die ATR zweimal gesendet wird.
Das Programm besteht im wesentlichen aus einer Funktion zum Einlesen von Bytes. Abbildung 24 zeigt das Struktogramm für die Funktion. In A.1 ist der Code zu finden.

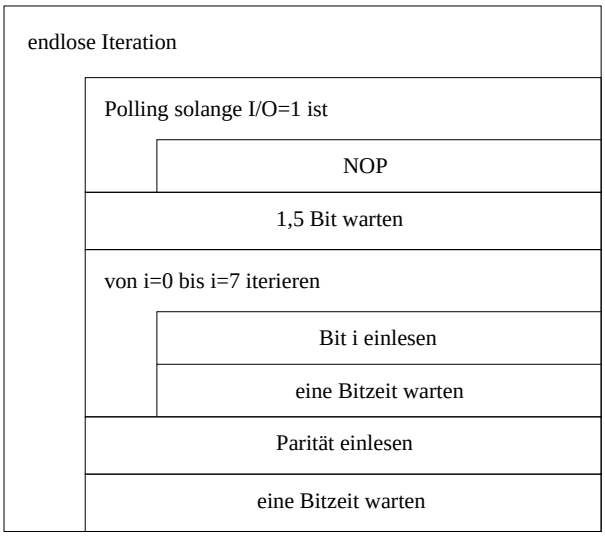

Abbildung 24: Funktion zum Einlesen von Bytes

Es wird ein Polling auf der Datenleitung durchgeführt. Wechselt der Zustand von 1 auf 0, so wird die Abtastung eines Bytes eingeleitet. Alle Bits werden in einem zweidimensionalen Array abgelegt. Beendet wird das Programm durch Strg-C. Das Signal wird von einem Handler abgefangen. Vor dem Beenden wird noch die Ausgabe durchgeführt. Das kann nicht während der Kommunikation erfolgen. Das Schreiben auf die Konsole hat eine zu große Verzögerungszeit. Deshalb wird die Übertragung erst in ein Array geschrieben, und dann komplett ausgegeben.
Der Zugriff auf das Parallelport geschieht durch Lesen der Register mit dem Befehl `inb()`. Unter Linux kann mit den Befehlen `ioperm()`, `inb()`, `outb()` direkt auf den Gerätespeicher aus dem Usermode zugegriffen werden. Der Vorteil ist, dass der Kernel die Zugriffe nicht verzögert. Das wird sehr oft im eingebetteten Bereich verwendet, spricht allerdings gegen das ‚normale' Vorgehen, der Abstraktion von Geräten durch Gerätetreiber.

5.1.6 Softwareumsetzung des T=0 Protokolls

Eine funktionsfähige Softwareimplementierung des T=0 Protokolls für das Parallelport ist im Programm t0comm.c zu finden. Für das Timing werden for-Schleifen wie in 5.1.5 verwendet. Das Programm soll hier nicht erklärt werden. Der Aufbau der C-Lösung für das ARM Board ist der gleiche. Lediglich der Zugriff auf die I/O-Ports und das Timing ist dort anders gelöst. Da die Zielplattform dieser Arbeit der Java Internet Server ist, wird lediglich die Lösung dafür näher erläutert.

5.2 Portierung auf das ARM Board

Die Implementierung des T=0 Protokolls für den PC ist auf dem Java Internet Server nicht lauffähig. Es ist nötig, den Code zu portieren.

5.2.1 Uploadproblem

Der Java Internet Server verfügt über eine Ethernet-Schnittstelle. Die Anbindung über einen Treiber ist aber noch nicht fertig gestellt. Da das komplette Flashen des ROM-Image über den Hardware-Debugger sehr zeitaufwendig ist, wurde nach einer Alternative gesucht.
Eine gute Möglichkeit für den Upload ist die Verwendung des ZMODEM-Protokolls. Die Sourcen befinden sich im Unterverzeichnis /zmodem auf der beiliegenden CD. Das Makefile ist für den Cross-Compiler des ARM Boards angepasst. Die Makros POSIX und MD=2 müssen verwendet werden. Mit dem Makefile entsteht das Programm rz. Dieses ist im ROM-Image abgelegt. Für einen Upload ist das Programm rz auf dem Java Internet Server zu starten. Anschließend kann in Minicom über die Tastenkombination Strg-A Z S ein Upload über das ZMODEM Protokoll ausgewählt werden.
Abbildung 25 zeigt den Upload der JVM in Minicom.

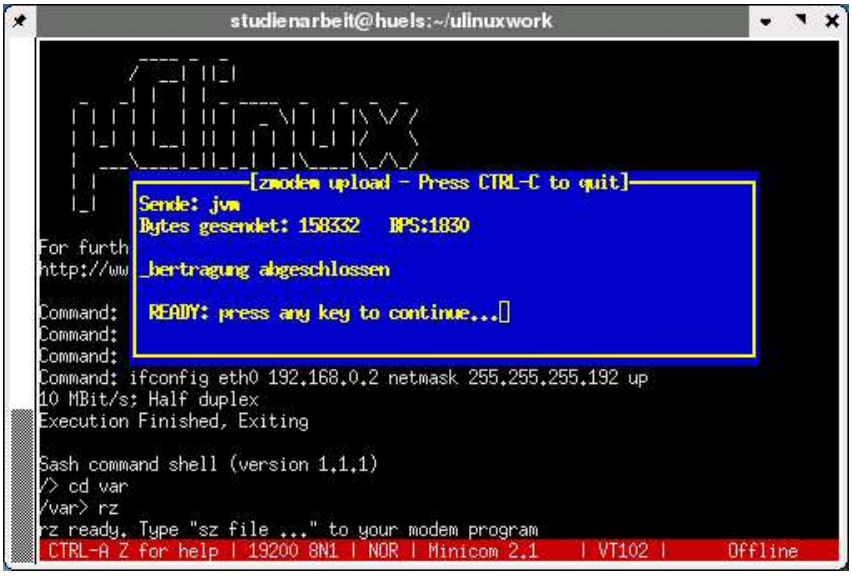

Abbildung 25: Upload über das ZMODEM Protokoll

5.2.2 Bittiming und Zugriff auf die GPIO Ports

Die C-Lösung des T=0 Protokolls für den Parallelport lässt sich einfach auf den Java Internet Server portieren. Es sind lediglich der Zugriff auf die I/O-Ports und das Timing anzupassen. Die Verbindung des Kartenlesers mit dem eingebetteten System ist im Versuchsaufbau beschrieben. Es ist genauso wie bei der Anbindung an das Parallelport realisiert. Der Unterschied liegt lediglich darin, dass das ARM Board mit einer Spannung von 3,3 Volt arbeitet. Die Emittergrundschaltung ist dafür anzupassen. Außerdem sind die Eingänge über Spannungsteiler geschaltet. Diese sorgen für eine Spannungsumsetzung von 5 auf 3,3 Volt. Die Widerstände sind sehr hoch dimensioniert um den Kartenleser nicht unnötig zu belasten.
Das ARM Board ist eine memory-mapped Architektur. D.h. der Hauptspeicher wird mit dem Speicher der Peripherie auf einen linearen Speicherbereich abgebildet.
Für jedes Modul werden verschiedene Register im Speicher zur Verfügung gestellt.
Für den Zugriff auf die I/O-Pins sind die Register IOPMOD und IOPDATA wichtig. In IOPMOD können die Ports als Ein- oder Ausgänge definiert werden. IOPDATA bildet den Zustand der Pins ab. Mit dem Register IOPCON können Sonderfunktionen für Pins eingeschaltet werden. Beispielsweise ist

es möglich, einen Pin als Timerausgang zu definieren.
Für das Timing wird Timer 1 des Boards verwendet. uCLinux benutzt bereits Timer 0. Die 32 Bit Timer sind im Toggle und Intervall Modus verwendbar. Das erzeugte Signal kann auf einen I/O-Pin als Sonderfunktion gelegt werden. Nach der Initialisierung mit einem ganzzahligen Wert, und Starten des Timers, zählt der Zähler bis 0. Das geht solange, bis der Timer angehalten wird. Ein Zählschritt dauert genau 1/50000000s = 20ns.
Um die Bitzeit von 123 usec zu erzeugen, ist der Timer auf den Wert 6150 zu initialisieren. Nach dem Starten kann auf den Nulldurchgang gepollt werden. Es ist auch möglich, einen Interrupt zu erzeugen.
Über das Register TMOD können die Timer gestartet und angehalten werden. Außerdem ist der Modus und der Anfangszustand im Toggle Modus definierbar. Der Modus ist für das Timing nicht von Bedeutung. Er ist lediglich interessant, um das Signal zu messen, und damit den Timingwert zu verifizieren. Mit Hilfe der beiden TDATA Register können die Timer initialisiert werden. Die Register, in denen das Zählen statt findet, heißen TCNT.
Der Timerbaustein ist in seiner Funktionalität leider eingeschränkt. Es ist nicht möglich die Timer mit einem externen Takt zu versorgen. Damit ist die Zählfunktion, die in vielen Mikrokontrollern zur Verfügung steht, nicht realisierbar.
Für das Bittiming wäre das die optimale Lösung. Mit dem Takt des Kartenlesers ist damit bis 372 zu zählen, das ergibt genau eine Bitweite.
In A.2 sind die verwendeten Register mit ihren Adressen aufgeführt.
A.3 zeigt die Initialisierung der Pins und des Timer 1.
Die Pins beziehen sich auf den Port CN4 des ARM Boards [3]. Pin 20 ist mit Masse verbunden. Pin 19 ist der Ausgang, um die Emittergrundschaltung anzusteuern. Pin 18 ist über einen Spannungsteiler mit dem Reset des Kartenlesers verbunden. Mit Pin 17 wird die Datenleitung abgehört.
Die Funktion `timer_init()` wird für die Kommunikation nicht benötigt. Damit kann lediglich Pin 3 mit dem Toggle Signal des zweiten Timer belegt werden. Das ist in der Entwicklung nötig, um die Bitzeit zu verifizieren. Die Konstruktion `reg | (1<<x)` liefert den Inhalt von reg mit einer 1 an Position x. `reg & ~(1<<x)` liefert den Inhalt von reg mit einer 0 an Position x.
Die Funktionen in A.4 dokumentieren den Zugriff auf die reservierten Pins.
Für das Bit Timing werden eine Bitzeit und eineinhalb Bitzeiten gebraucht. Die Funktionen `wait_bit_time()` und `wait_bit_time_and_a_half()` werden dafür eingesetzt. Siehe A.5.
Abbildung 26 zeigt das Struktogramm für das Erzeugen einer Bitzeit.

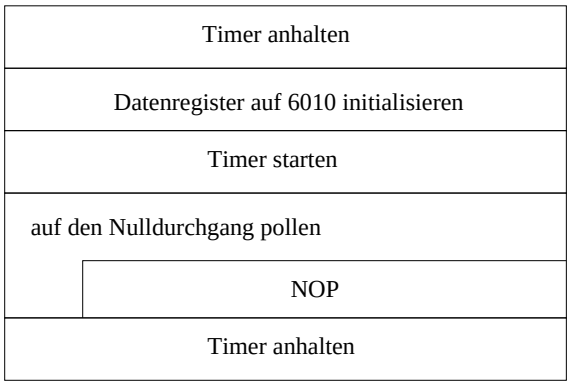

Abbildung 26: Erzeugen einer Bitzeit mit Timer 1

Der Timer wird je nach benötigter Zeit initialisiert und gestartet. Anschließend ist ein Polling auf den Nulldurchgang realisiert. Allerdings ist es besser, das Timing wie bereits erwähnt bei einer Integration des Java Internet Servers zu berücksichtigen. Die optimale Lösung ist ein Timer, der extern mit dem Takt des Kartenlesers versorgt werden kann.

5.2.3 C-Lösung

Wie bereits erwähnt soll hier auf die C Variante des Protokolls näher eingegangen werden. Mit dem Programm ist es möglich Kommandos mit und ohne Daten an das ARM Board zu schicken. Über ein Get Response können Daten vom eingebetteten System geholt werden. Mit dieser Funktionalität des T=0 Protokolls auf der Leitungsschicht, können alle 4 Kommunikationsvarianten auf der Anwendungsschicht erzeugt werden.
Abbildung 27 zeigt die Terminalseite der Kommunikation. Mit dem Programm Cardping wird das Protokoll getestet. Es werden neben dem Reset beispielhaft 5 APDUs geschickt.

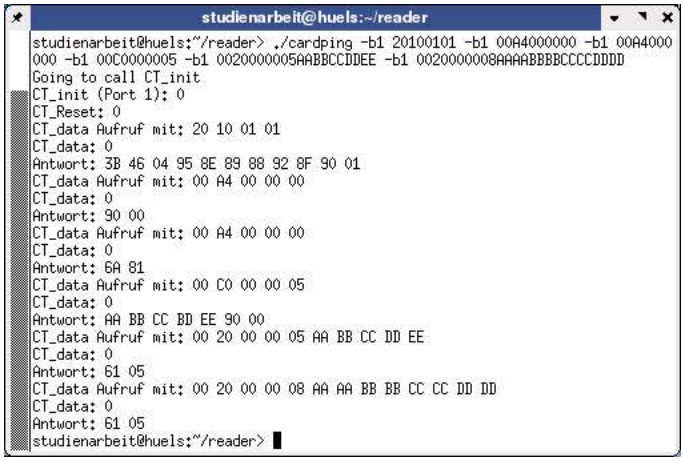

Abbildung 27: Test des T=0 Protokolls

Nach dem Reset antwortet das ARM Board mit der ATR 0x3B 0x46 0x04 0x95 0x8E 0x89 0x88 0x92 0x8F. Die ATR ist für die Softwareimplementierung des T=0 Protokolls konstruiert.
Mit 0x3B wird die direct convention festgelegt. Das Format Byte 0x46 legt fest, dass TB1 übertragen wird. Es folgen 6 Historical Characters und keine weiteren Definitionen für spezielle Protokolle. Mit TB1 0x04 werden für die Datenübertragung 4 weitere Stopbits eingeschaltet. Das ist gerade bei einer Softwareimplementierung für eine sichere Kommunikation sinnvoll.
In der Praxis werden die Historical Characters oft verwendet um firmeneigene Informationen, wie z.B. den Firmennamen in ASCII Codierung, in der ATR abzulegen.
Die 6 Historical Characters hier entsprechen der Folge UNIHRO.
Als Classbyte wird 0x00 verwendet. 0xA4 ist ein Kommando ohne Daten. Mit 0x20 ist ein Befehl definiert, der Daten an das ARM Board sendet. Es werden einmal 5 und einmal 8 Datenbytes übertragen. Das Instructionbyte 0xC0 steht standardmäßig für das Get Response beim T=0 Protokoll. Hier werden 5 Datenbytes in der Antwort erwartet.
Er ist zu erkennen, dass das ARM Board auf die Kommandos 0xA4 mit dem Returncode 0x90 0x00 antwortet. Das steht für ein korrektes Ausführen des Kommandos. Die Befehle mit Daten sind mit 0x61 0x05 bestätigt. 0x05 bedeutet, es liegen 5 Byte im Puffer der Karte. Mit dem Get Response wird die Bytefolge 0xAA 0xBB 0xCC 0xDD 0xEE vom Board abgeholt. Außerdem sendet der Java Internet Server die Bestätigung 0x90 0x00. Die Kommandos und Returncodes werden nur beispielhaft verwendet und sind für eine tatsächliche Anwendung anzupassen. Es soll hier lediglich die Funktion der

Software gezeigt werden. Abbildung 28 zeigt die uCLinux Konsole des Java Internet Servers.

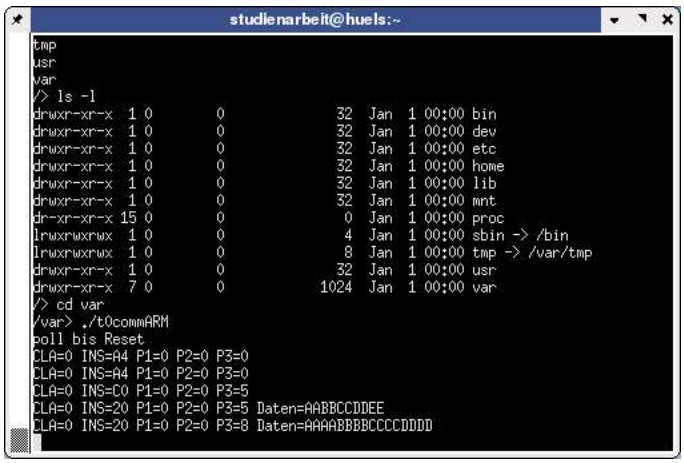

Abbildung 28: Konsole des Java Internet Servers

Das Programm `t0commARM` schreibt alle empfangenen APDUs und Daten auf die Standardausgabe und erzeugt die passenden Returncodes bzw. Datenfolgen.
Zu Beginn der Kommunikation wird ein Polling auf die Resetleitung durchgeführt. Nach einem Reset antwortet das eingebettete System mit der ATR wie in der Funktion `generate_atr()` zu sehen. Unter A.6 ist die Funktion aufgelistet. Laut Standard muss die ATR im Intervall zwischen 400 und 40000 etu nach dem Reset geschickt werden. Zwischen den Bytes dürfen 9600 etu Wartezeit liegen. Es fällt auf, dass die ATR zwei mal gesendet werden muss. Die Wartezeit dazwischen wurde von dem Test der GSM Karte verwendet. Die Funktion `wait_for_command()` wird in einer Endlosschleife aufgerufen. In A.7 ist die Funktion zu finden.
Die Funktion liest für jedes Kommando 5 Byte. In einem `switch()` Block werden die unterschiedlichen Instruction Bytes interpretiert. Die Funktion liefert als Rückgabewert den Kommandotyp. Im Fehlerfall werden Fehlerkonstanten zurückgegeben. Falls das Terminal ein Kommando mit Daten sendet, schickt das ARM Board das INS Byte als Acknowledge und liest die Daten ein.
Das Kommando, die Daten und ein Integerwert, der die Länge der Daten beinhaltet, werden Call By Reference übergeben.
A.8 zeigt die Endlosschleife mit dem Aufruf der Funktion `wait_for_command()`. Die Returncodes sind hier fest eingetragen. Bei einer Anwendung ergeben

sich die Rückgabewerte aus der Befehlsabarbeitung.
Die Funktion `read_bytes()` wird hier nicht aufgeführt. Der Aufbau ist der gleiche wie bereits bei der Protokollaufnahme vorgestellt. Die Funktion `send_io_byte()` erzeugt ein Byte auf der Datenleitung. Abbildung 29 zeigt das Struktogramm für die Funktion. Sie ist in A.9 zu finden.

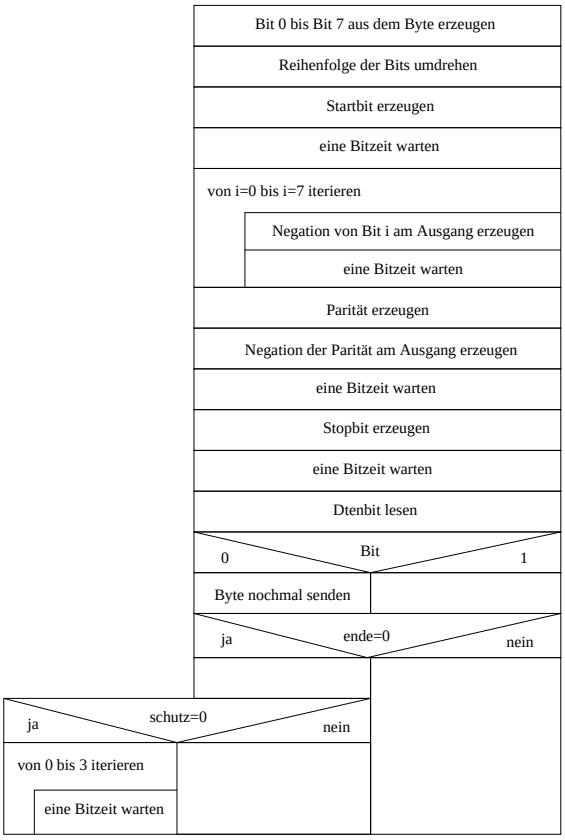

Abbildung 29: Funktion zum Senden eines Bytes

Die Bitfolge wird mit einem Startbit eingeleitet. Danach folgen die Datenbits und die Parität. Das Setzen des Stopbits schließt die Kommunikation ab. Falls `ende` 0 ist, so wird eine weitere Bitzeit abgewartet. Das entspricht den 2 Bits für die guard time. Ist `schutz` auf 1, werden die 4 weiteren Stopbits abgewartet. Das ist nur für die Datenübertragung wichtig. Für die Übertragung der ATR ist `schutz` 0, d.h. die ATR wird immer mit 2 Stopbits übertragen. Die Variable `ende` ist 1, falls sich das aktuelle Byte um das letzte einer zusammenhängenden Bytefolge handelt. Genau dann

ist es nicht nötig, die Stopbits abzuwarten. Die Datenleitung ist lediglich auf den logischen Zustand 1 zu bringen. Ein Stopbit ist immer zu warten. Nach diesem Stopbit wird der Zustand der Datenleitung gelesen. Ist die Leitung logisch 0, so hat der Kartenleser einen Paritätsfehler signalisiert. Bei einem Paritätsfehler wird das aktuelle Byte nochmal gesendet. Wegen der invertierenden Eigenschaft des Emitterfolgers werden alle Bits invertiert gesendet. A.10 dokumentiert die Hilfsfunktionen, die gebraucht werden.
Der Vollständigkeit halber werden in A.11 die Funktionen zur Erzeugung der Returncodes mit und ohne Daten aufgeführt. Es ist zu beachten, dass das erste Datenbyte dem Instruction Byte des Get Response entsprechen muss. Der Störabstand simuliert hier die Ausführungszeit eines Befehls auf der SmartCard.

5.2.4 Kertasarie VM

Diese Arbeit ist der erste Schritt eines Projektes, dass zum Ziel hat, ein USB Token herzustellen, welches in Java programmiert werden kann. Als Java VM soll die am Lehrstuhl für Rechnerarchitektur entwickelte Kertasarie VM eingesetzt werden. Sie wurde speziell für den Einsatz in ressourcenbeschränkten eingebetteten Systemen entworfen und hat eine typische Codegröße von 80-120 KByte. Folgende Designziele wurden bei der Entwicklung berücksichtigt [8].

- Java 2 Standard
- Netzwerkzentrierung (Sockets, Reflection, Serialisierung, TinyRMI)
- Portabilität (geschrieben in ANSI C, eigenes Thread Modell)
- Skalierbarkeit
- Integrierter Remote Debugger
- Eigenes GUI System
- Echtzeitfähigkeit (priority inheritance ist implementiert)

Für den Zugriff auf die I/O-Ports ist eine Java API vorhanden. Mit der Klasse `BitPort()` können die Pins gesetzt und ausgelesen werden. Die API greift auf einen Gerätetreiber zu, es ergeben sich Latenzen im 3 stelligen usec Bereich. Das kann für die Implementierung des T=0 Protokolls nicht verwendet werden.

Eine komplette Java Lösung ist nicht machbar. Für ein Abtasten des Protokolls sind die Latenzen der Zugriffe auf die I/O-Ports direkt aus Java zu groß. Das Protokoll muss in C programmiert werden. Die C Funktionen werden in einem Treiber für die JVM abstrahiert. Der Zugriff geschieht über eine dafür entwickelte Schnittstelle.

5.2.5 Java Native Interface

Das JNI erlaubt es C-Funktionen außerhalb der JVM aufzurufen. Das wird gebraucht, um z. B. Methoden des Betriebssystems zu verwenden, die es in Java nicht gibt. Dadurch geht natürlich die Plattformunabhängigkeit der Software verloren. Die Methoden in Java müssen dazu mit dem Schlüsselwort `native` gekennzeichnet werden. Damit erkennt der Byte Code Interpreter, dass eine Funktion des JNI aufgerufen werden muss.
Für die virtuellen Machinen im PC und Workstation Bereich steht das Kommando javah zur Verfügung. Der Befehl erzeugt für eine Klasse ein Headerfile mit den Rümpfen der nativen Methoden. Diese File darf nicht verändert werden. Entsprechend der Definitionen des Headerfiles, kann der C Code erzeugt werden. Für die C Funktionen ist ein dynamisches Modul zu erzeugen, das von der JVM geladen wird.
Im folgenden wird ein einfaches JNI Headerfile vorgestellt um die Schnittstelle zu erläutern. In dem Beispiel soll die C Funktion `getpass()` für eine Passworteingabe auf der Konsole verwendet werden. Diese Funktionalität ist in Java nicht vorgesehn.

Javateil:

```
package enigma;

public class Crypt
{
  private native static String getpass(String prompt);

  static
  {
    System.loadLibrary("crypt");
  }
  public static void main()
  {
    String prompt="";
    getpass(prompt);
  }
}
```

Headerfile:

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

#ifdef _Included_enigma_Crypt
#define _Included_enigma_Crypt
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: class_0024enigma_0024Crypt */
/*
 * Class:     enigma_Crypt
 * Method:    getpass
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_enigma_Crypt_getpass
  (JNIEnv *, jclass, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Für eine eindeutige C-Funktion wird zum Funktionsnahmen (hier `getpass`) der Klassenname, das Package und das Präfix Java vorangestellt. Über den Zeiger `JNIEnv` können Funktionen des JNI aufgerufen werden. Die Methode `getpass()` ist als Klassenmethode definiert, deswegen wird als weiterer Parameter der Typ `jclass` übergeben. `jstring` steht für den eigentlichen Parameter der Java Methode. Alle Typen aus Java sind in C nachgebildet, sie haben als Präfix den Buchstaben j. Das JNI stellt eine Menge von Funktionen zur Verfügung. Beispielsweise kann mit der Funktion `GetStringUTFChars()` ein Java String in einen C String gewandelt werden. Das ist nötig um die Parameter in C bearbeiten zu können. Auch höhere Datentypen wie Objekte oder Arrays sind im JNI vorhanden. Nach den Vorgaben des Headerfiles können die C Funktionen geschrieben werden. Im Folgenden ist die C Implementierung zum obigen Beispiel angegeben.

```
#include <unistd.h>
#include "enigma_Crypt.h"

/*
 * Class: enigma_Crypt
 * Method: getpass
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */

JNIEXPORT jstring JNICALL Java_enigma_Crypt_getpass (
  JNIEnv *p_env,
  jclass cl,
  jsting jprompt)
{
  const char *prompt = (*p_env)->GetStringUTFChars(p_env, jprompt, NULL);
  const char *password = getpass(prompt);
  jstring jpassword = (*p_env)->NewStringUTF(p_env, password);
  return jpassword;
}
```

Daraus ist eine dynamische Bibliothek zu generieren.
Diese kann mit `System.loadLibrary(lib)` in Java geladen werden.

5.2.6 Schnittstelle für native Funktionen der Kertasarie VM

Im eingebetteten Bereich und damit auch bei der Kertaserie VM ist das Handling des JNI anders, als oben beschrieben. Es wird darauf geachtet, das Interface einer embedded VM möglichst schlank zu halten.
Der Einsatz von `javah` ist für die Kertasarie VM nicht möglich. Die Namen der Funktionen, die extern aufgerufen werden sollen, müssen in eine Tabelle in der VM eingetragen werden. Diese Tabelle befindet sich im File `table.c`. Jeder Eintrag steht für eine native Funktion, zu der die entsprechende Funktionsadresse gespeichert ist. Das macht es nötig bei der Entwicklung mit nativen Funktionen die VM komplett neu zu übersetzen.
Die Funktionsnamen müssen in die Tabelle in richtiger alphabetischer Reihenfolge geschrieben werden. Das ist nötig, da die Suche in der Tabelle als Binärsuche implementiert ist. Im ersten Schritt wird mit dem Eintrag in der Mitte der Tabelle verglichen. Durch die alphabetische Reihenfolge ergibt sich, dass der Eintrag in der oberen oder unteren Hälfte der Tabelle liegt. Im nächsten Schritt wird wieder mit der Mitte der gefundenen Tabellenhälfte verglichen. Das ist solange zu wiederholen, bis der Eintrag gefunden ist. Bei einer Tabelle mit der Länge n sind maximale `log2n` Suchschritte für einen Eintrag nötig. Das ist viel effektiver als ein lineares Durchsuchen der Tabelle.

Der Aufruf von C Funktionen geschieht dreistufig. Abbildung 30 zeigt den Aufbau.

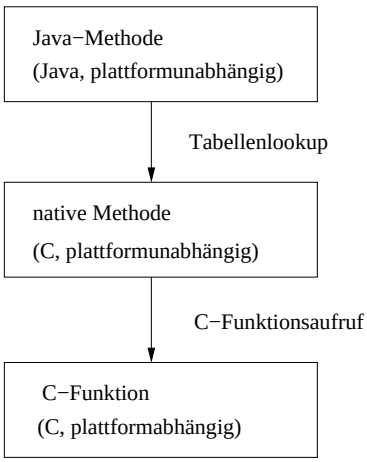

Abbildung 30: Aufruf nativer Funktionen

Die Java Methoden müssen mit dem Schlüsselwort `native` gekennzeichnet werden. Das veranlasst den Java Interpreter die dazugehörige plattformunabhängige C Funktion in der Tabelle zu suchen. Die plattformunabhängige C-Funktion muss sich im Verzeichnis `api/native` befinden. Der letzte Schritt ist der Aufruf der plattformabhängigen C-Funktion. Diese muss in einem Unterverzeichnis von platform liegen, hier `platform/linux`.
In eingebetteten Systemen ist häufig der Speicherplatz sehr beschränkt. Deshalb ist es sinnvoll, nur die Teile in die JVM zu kompilieren, die auch gebraucht werden.
Für die nativen Funktionen ist dazu das File `include/table.h` vorgesehen. Darin werden zu jedem API Package entsprechende Makros definiert. Abhängig davon, ob das Makro definiert ist oder nicht, werden die dazugehörigen nativen Funktionen in die Tabelle aufgenommen oder nicht. So läßt sich die JVM sehr einfach für eigene Bedürfnisse anpassen.
Native Methoden habe in der JVM Kertasarie folgendes Interface:

```
object_t* nativeMethod(register thread_t* t);
```

Der Zeiger `t` zeigt auf eine Struktur des Typs `thread_t`. Damit ist der Zugriff auf Laufzeitvariablen des aufrufenden Threads möglich. Es lassen sich damit z. B. der Programmzähler oder die oberste Stackposition ermitteln. Der Rückgabewert ist eine Objektreferenz.
Folgende Werte sind möglich:

- `NULL` die Methode wirft keine Exception und blockiert nicht
- `(object_t*)1` die Methode blockiert; der Programmzähler wird dekrementiert, die Methode wird neu aufgerufen

- anderer Wert ist ein Zeiger auf ein Exception Objekt

Die Parameterübergabe zwischen der Java und der C Funktion geschieht über einen Stack. Bei einem nativen Methodenaufruf wird ein Stackframe erzeugt, das unter Anderem die Parameter enthält. Die C Funktion liest die Parameter durch das Auslesen des Stacks. Ein eventueller Rückgabewert ist auch auf den Stack zu legen. Bei Instanzmethoden liegt zusätzlich eine Referenz auf das aufrufende Javaobjekt auf dem Stack. Es ist darauf zu achten, den Stack entsprechend des Methodenrumpfes aufzuräumen.
Zugegriffen wird auf den Stack über den Zeiger `t`.
Das Element `t->topofStack` enthält die oberste Stackposition. Um einen 32 Bit Wert vom Stack zu nehmen, genügt es eine 1 zu subtrahieren. Mit den Makros `GET_WORD` und `GET_WORD_DEC` können die Inhalte des Stacks in lokale Variablen kopiert werden. Das Makro `GET_WORD_DEC` dekrementiert zusätzlich den Stackpointer.
Abbildung 31 zeigt den Stack für eine Instanz und eine Klassenmethode.

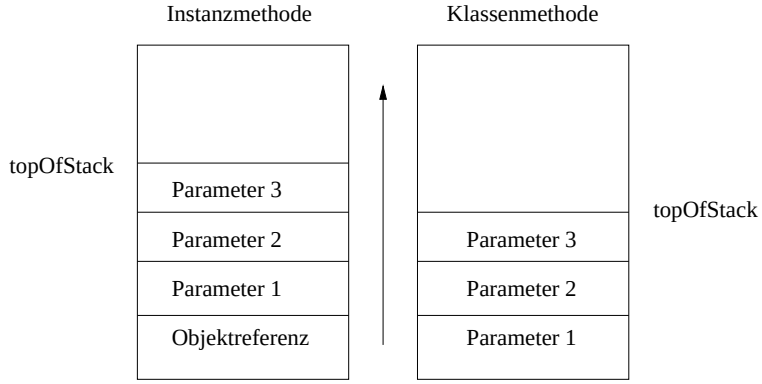

Abbildung 31: Parameter auf dem Stack

Nach der Funktionsausführung muß der Stack geleert sein. Anschließend ist der Rückgabewert auf den Stack zu legen. Das kann mit den Makros `PUT_WORD` und `PUT_INC_WORD` gemacht werden. Bei einer Methode mit dem Rückgabewert void bleibt der Stack leer.

5.2.7 T=0 Treiber für die Kertasarie VM

Um aus Java Programmen auf das T=0 Protokoll zugreifen zu können, ist ein Treiber nötig. Es soll hier die Verwendung der Befehle des T=0 Protokolls ermöglicht werden. Die Abstraktion auf Anwendungsebene geschieht hier noch nicht. Das ist komplett in Java zu schreiben. Dabei muß eine Art

Monitor entstehen, der es ermöglicht mehrere Java Anwendungen zu verwalten. Die Anwendungen müssen eine `select()` Methode implementieren. Mit dieser Methode kann eine entsprechende Anwendung über eine APDU ausgewählt werden.
Abbildung 32 zeigt das Klassendiagramm.

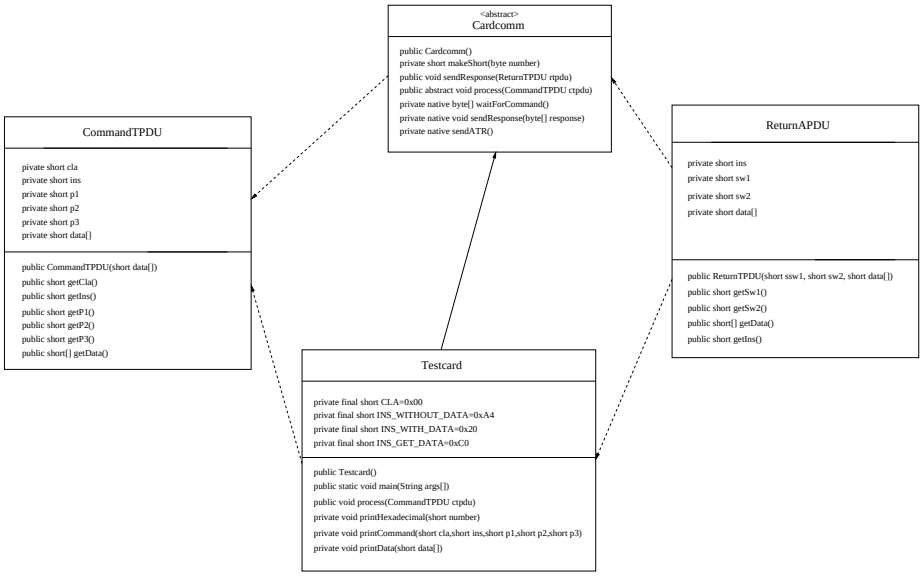

Abbildung 32: Klassendiagramm

In der Klasse `Cardcomm` wird über die nativen Funktionen auf das Kommunikationsprotokoll zugegriffen. Die Klasse ist abstrakt. Eine direkte Instantiierung ist nicht möglich. Abstrakte Klassen können nur über Vererbung an andere Klassen verwendet werden. Dabei wird die Subklasse gezwungen, alle abstrakten Methoden zu überschreiben.
Hier ist die Methode `process()` als abstrakt gekennzeichnet. Die Methode muss in der Unterklasse erscheinen. Sie wird von der Superklasse bei jeder empfangenen TPDU aufgerufen.
Die Klassen CommandTPDU und ReturnTPDU werden zur Kapselung der entsprechenden TPDUs verwendet. Im Klassendiagramm steht der durchgezogene Pfeil für Vererbung, die gestrichelten Pfeile beschreiben Aggregationen. Die Vererbung beschreibt eine besteht-aus, die Aggregation eine ist-Teil-von Beziehung.
In B1 ist die Klasse `Testcard` aufgelistet. Damit kann das T=0 Protokoll getestet werden. Die Verwendung des Treibers ist an die Java Card API angepasst. Die Beispielklasse hat demzufolge den gleichen Aufbau, wie eine

Applet mit dem Java Card Framework. Die Kommunikation über das T=0 Protokoll ist für die Klasse transparent. Die Funktionalität dazu ist in der Klasse `Cardcomm` und der dazu gehörenden nativen Funktionen gekapselt.
Die Klasse `Testcard` erbt die Klasse `Cardcomm`. Wie im Klassendiagramm beschrieben, muss die abstrakte Methode `process()` überschrieben werden. In einer Anwendung wird die Methode `process()` automatisch beim Eintreffen einer TPDU aufgerufen. Als Parameter wird eine Instanz der Klasse `CommandTPDU` übergeben, in dem das Kommando gekapselt ist.
Auf die verschiedenen instruction Bytes wird in einem `switch()` Block reagiert. Mit der Methode `sendResponse()`, die von der Klasse Cardcomm zur Verfügung gestellt wird, kann eine Antwort an den Kartenleser gesendet werden. Dabei ist ein Objekt der Klasse `ReturnTPDU` zu übergeben. Die Klasse `ReturnTPDU` kapselt die Bytes `sw1`, `sw2` und ggf. die Datenbytes einer Antwort.
In den case Blöcken sind anwendungsspezifische Funktionen aufrufbar. Diese können auf die Kommandos reagieren. Die Beispielklasse schreibt aber lediglich die ankommenden Daten auf die Standardausgabe und erzeugt Beispielantworten.
Alle Bytes erscheinen in hexadezimaler Schreibweise auf der Konsole. So wie es im SmartCard Bereich üblich ist. In Java fehlen dazu Funktionen in der Core API.
Die hexadezimale Ausgabe wird über die Funktion `printHexadecimal()` realisiert. Dabei werden die beiden Halbbytes eines Bytes getrennt behandelt. Liegt der Wert des Halbbytes zwischen 0 und 9, so wird die Ziffer direkt ausgegeben. Bei 10 bis 15 erfolgt eine Abbildung auf die Werte 0x41 bis 0x46. Das steht für A bis F in ASCII Codierung. Um die Buchstaben auf der Konsole zu erhalten, wird auf den Typ `char` gekastet.
Abbildung 33 zeigt das Struktogramm für die Funktion `printHexadecimal()`.

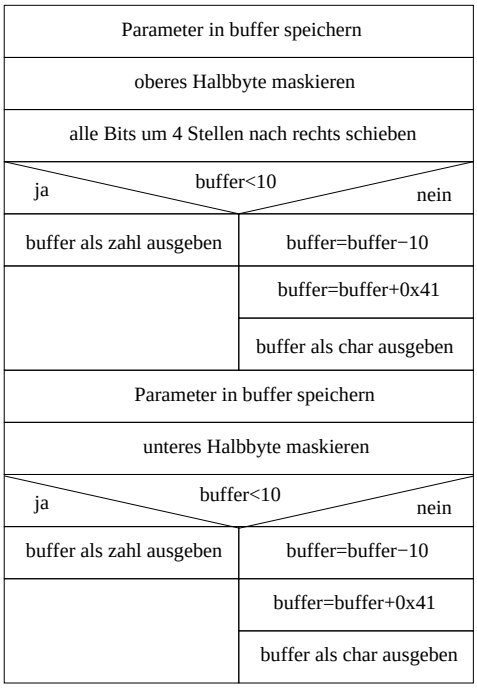

Abbildung 33: Funktion für die Ausgabe von Hexadezimalwerten in Java

In der Klasse `Cardcomm` in B.4 sind die nativen Funktionen für das T=0 Protokoll zu finden. Im Konstruktor wird bei jeder Initiierung eine ATR erzeugt. Die Funktion `waitForCommand()` wird in einer Endlosschleife aufgerufen. Das Kommando und ggf. die Daten werden von `waitForCommand()` in einem Bytearray zur Verfügung gestellt. Für eine Instanz der Klasse `CommandTPDU` ist ein Shortarray nötig. Dazu ist die Methode `makeShort()` implementiert. Für jedes Kommando findet ein Aufruf der abstrakten Funktion `process()` statt.
Die Funktion `sendResponse()` ermöglicht das Erzeugen von Antworten für das T=0 Protokoll.
Zu übergeben ist dabei ein Objekt vom Typ `ResponseTPDU()`.
In der Datei `api_t0comm.c`, im Verzeichnis `/api/native` der Kertasarie VM, sind die C-Funktionen für das T=0 Protokoll abgelegt. Die Funktionalität ist die selbe wie in den Funktionen in Anhang A. Lediglich die Parameterübergabe ist für die Kertasarie VM angepasst. Die Parameter werden, wie in 5.2.6 vorgestellt, über einen Stack übertragen.
Im folgenden sind die Tabelleneinträge für die nativen Funktionen aufgelistet.

```
COND_COMP(TOCOMM,"generateReturnCode.(IIII[B)V")
COND_COMP(TOCOMM,"sendATR.()V")
COND_COMP(TOCOMM,"waitForCommand.()[B")
```

In den Klammern stehen die Abkürzungen für die Parameter. Der Typ des Rückgabewerts ist nach der Klammer anzugeben.

I steht für Integer. Die erste Funktion hat vier Integerwerte für ins, sw1, sw2 und die Länge der Antwortdaten. Es wird kein Rückgabewert erzeugt. Die Methode zum Erzeugen einer ATR besitzt weder einen Parameter, noch einen Rückgabewert. Das Kommando und die Daten sind bei der Funktion `waitForCommand` als ein Bytearray abstrahiert.
Der folgende Codeausschnitt zeigt beispielhaft das Auslesen der Parameter aus dem Stack.

```
/* public native void generateReturnCode(int ins,int sw1,int sw2,int datalegth,
                                          byte[] data) */
object_t* generateReturnCode(register thread_t* t)
{
  array_t * bytes=(array_t *)GET_WORD_DEC(t->topOfStack);
  unsigned char * data=bytes->data[0].charData;
  signed long datalength=GET_WORD_DEC(t->topOfStack);
  signed long sw2=GET_WORD_DEC(t->topOfStack);
  signed long sw1=GET_WORD_DEC(t->topOfStack);
  signed long ins=GET_WORD_DEC(t->topOfStack);
  object_t* obj = (object_t*)GET_WORD(t->topOfStack);
  .
  .
  .
  return NULL;
}
```

Die Parameter werden von der VM aufsteigend auf den Stack gelegt. Das Auslesen geschieht demzufolge in umgekehrter Reihenfolge. Es muss hier mit dem Array begonnen werden. Nach jedem Parameter ist die Stackposition zu dekrementieren. Jedoch nicht bei dem letzten Element, hier dem aufrufenden Objekt. Dabei soll der Stackzeiger auf dem ersten Element stehen bleiben.
Das Erzeugen eines Rückgabewerts geschieht in der selben Weise mit dem Macro `PUT_WORD`.
In `waitForCommand()` wird dazu mit der Methode `ex_initNewArray()` ein Bytearray erzeugt. Der Rückgabewert ist ein Zeiger auf den Typ `array_t`. Wird für den Zeiger `NULL` zurückgegeben, so ist kein Speicherplatz mehr verfügbar.
Dabei muss mit der Funktion `gc_collectGarbage(NON_INTERRUPTIBLE)` der Garbage Collector aufgerufen werden. Dannach ist das Generieren des Arrays neu zu versuchen.
Im Unterverzeichnis `/include` muss zum File `api_t0comm.c` ein passende Headerfile erzeugt werden.
Das Eincompilieren der Funktionen des T=0 Protokolls kann in der Kertasarie VM über die Datei `table.h` in `include` gesteuert werden. Der folgende Ausschnitt zeigt das Macro für das T=0 Protokoll.

```
#ifdef TOCOMM_COMPONENTS
#define TOCOMM_COMPONENT_INCLUDED(x) x,
#else
#define TOCOMM_COMPONENT_INCLUDED(x)
#endif
```

Die nativen Funktionen, welche in der Tabelle `table.c` eingetragen sind, befinden sich in der Datei `api_t0comm.c`. Die Funktionen sind nicht wirklich

nativ, sie sind sehr einfach gehalten und platformunabhängig. Die wirklich nativen und damit platformabhängigen Funktionen befinden sich im File `pl_t0comm.c`, das im Unterverzeichnis `/platform/linux` abgelegt ist. Abbildung 34 zeigt den Test des T=0 Protokolls mit der Kertasarie VM.

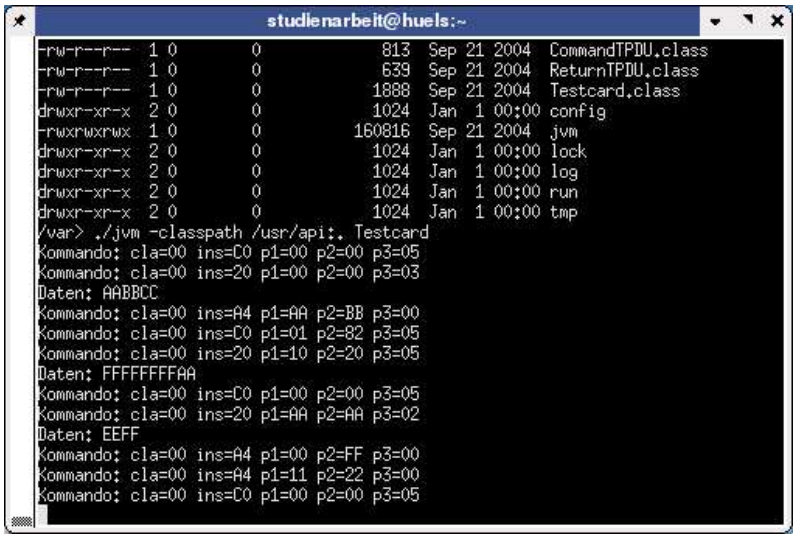

Abbildung 34: Test des T=0 Protokolls

Folgende TPDUs wurden übertragen.

```
TPDU1=00C0000005
TPDU2=0020000003 AABBCC
TPDU3=00A4AABB00
TPDU4=00C0010205
TPDU5=0020102005 FFFFFFFFAA
TPDU6=00C0000005
TPDU7=0020AAAA02 EEFF
TPDU8=00A4112200
TPDU9=00C0000005
```

6 Probleme

Problem bei der Implementierung des T=0 Protokolls ist das Timing. Der Timer des ARM Internet Server kann nicht mit dem Takt des Kartenlesegeräts versorgt werden. Bei der Hardwareentwicklung eines Tokens muss darauf geachtet werden, dass die Zählfunktion von den Timern zur Verfügung gestellt wird. Damit kann eine etu exakt erzeugt werden.
Ein alternativer Ansatz für die bestehende Architektur ist die Verwendung eines externen Zählers. Der Zähler wird mit dem Takt des Kartenlesers verbunden. Ist der Teiler von 372 erreicht, so muß ein Impuls an einem I/O Port erzeugt werden. Der Reset muss über einen Ausgang des Java Internet Servers gesteuert werden. Durch die Aktivierung des Resets wird die Erzeugung einer Bitzeit eingeleitet. Abbildung 35 zeigt den Aufbau des Zählers.

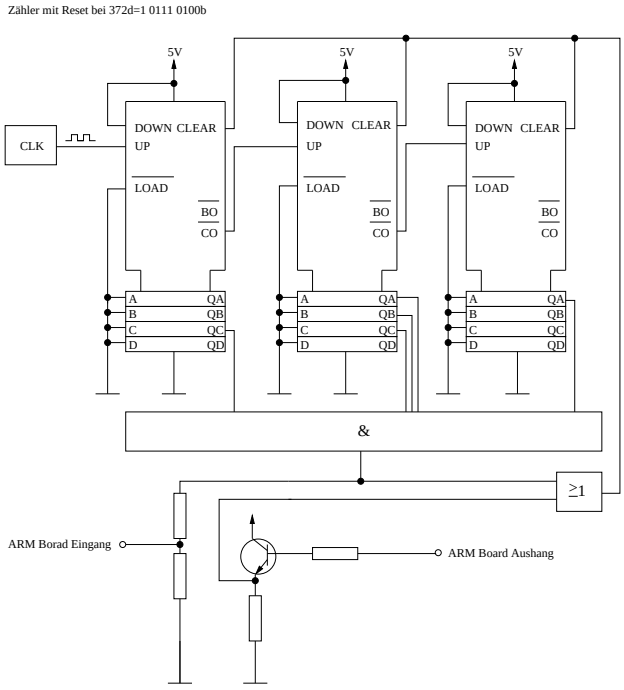

Abbildung 35: Zähler für das Timing

Dieser zusätzliche Hardwareaufwand ist allerdings nicht wünschenswert. Besser ist es, einen passenden Timer zu verwenden.

7 Zusammenfassung

In dieser Studienarbeit konnte gezeigt werden, dass es möglich ist, den Java Internet Server um das T=0 Protokoll zu erweitern. Auf der PC-Seite kann das Protokoll über den USB angesprochen werden. Verwendet wird dazu ein USB-SmartCard-Reader. Die Hardware, die für eine Verbindung mit dem ARM Internet Server nötig ist, wurde mit einer Versuchsplatine aufgebaut. Das Protokoll reserviert dazu drei der GPIO Ports. Es wurde dafür eine C-Lösung vorgestellt. Ausserdem war es möglich einen Treiber für die Kertasarie VM zu schreiben. Damit können Anwendungen in Java programmiert werden, die das T=0 Protokoll benutzen. Es konnte beispielhatf gezeigt werden, dass die Kommunikation zwischen einem Host und dem ARM Internet Server über das T=0 Protokoll möglich ist. Im Rahmen dieser Arbeit konnte für das Bittiming keine optimale Lösung implementiert werden. Ursache dafür ist, das der Java Internet Server nötige Hardwarefunktionen nicht zur Verfügung stellt.

8 Ausblick

Um das T=0 Protokoll in einem Token verwenden zu können, muss das Timingproblem gelöst werden. Bester Ansatz ist die Verwendung eines Timers, der mit einem externen Takt versorgt werden kann. Dadurch ist eine für die Praxis nötige Stabilität der Implementierung möglich. Für ein Token ist der ARM Internet Server geometrisch zu verkleinern. Dafür muss ein Prozessor mit kleinerer Abmessung ausgesucht werden. Außerdem ist darauf zu achten, die Hardware "sicher" aufzubauen. Ein unerlaubter Zugriff von außen darf nicht möglich sein. Das Token kann verwendet werden, um geheime Daten zu speichern, oder kryptographische Algorithmen sicher auszuführen.

Literatur

[1] Böhm Oliver: Java Software Engineering unter Linux, SuSE AG Nürnberg, 2002

[2] Cap Clemens, Vorlesung Smart Cards Universität Rostock, 2002

[3] Hartwig Volker, Diplomarbeit Hardware-Implementierung eines minimalen Java-Servers, 2002

[4] Herold Helmut, Arndt Jörg: C-Programmierung unter Linux, SuSE AG Nürnberg, 2002

[5] Lindner, Bauer, Lehmann: Taschenbuch der Elektrotechnik und Elektronik, Fachbuchverlag Leipzig-Köln , 1995

[6] Rankl Wolfgang, Effing Wolfgang: Handbuch der Chipkarten, Carl Hanser Verlag München Wien, 1998

[7] Tavangarian Djamshid, Vorlesung Prozessorarchitekturen Universität Rostock, 2002

[8] Versick Daniel, Studienarbeit Portierung von uClinux und der Kertasarie VM auf den ARM Internet Server, 2003

[9] Walter Klaus-Dieter: Messen, Steuern, Regeln mit Linux, Franzis Verlag GmbH Poing, 2001

[10] iKey3000 Datasheet, www.de.rainbow.com, Rainbow Technologies Inc., 2003

[11] Ort Ed: Writing a Java Card[tm] Applet, www.sun.com, 2001

A C-Funktionen

Um einen Überblick der Softwarelösung zu bekommen, sind im Anhang A die Grundlegenden C-Funktionen aufgelistet.

A.1 Funktion zum Lesen von Bytes

```
void read_byte(int port)
{
  struct timeval anfang;
  struct timeval ende;
  int i = 0;
  while (1) // Abbruch durch Strg-C, Ausgabe durch Signal
  {
    gettimeofday(&anfang,NULL);
    while(read_io_bit(port)==1); // Polling bis Leitung auf 0 gezogen wird
    gettimeofday(&ende,NULL);
    pollzeiten[j]=ende.tv_usec-anfang.tv_usec;
    for(i=0;i<=BITANDHALFTIME;i++);
    bits[j][0]=read_io_bit(port); // erstes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][1]=read_io_bit(port); // zweites Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][2]=read_io_bit(port); // drittes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][3]=read_io_bit(port); // viertes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][4]=read_io_bit(port); // fünftes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][5]=read_io_bit(port); // sechstes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][6]=read_io_bit(port); // siebtes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][7]=read_io_bit(port); // achtes Bit
    for(i=0;i<=BITTIME;i++);
    bits[j][8]=read_io_bit(port); // parity Bit
    for(i=0;i<=BITTIME;i++); // warte 1 Bit dann ist Leitung wieder 1
    j++;
  }
}
```

A.2 Adressen der verwendeten Register

```
BASE_ADDR 0x3FF0000
IOPDATA   BASE_ADDR+0x5008
IOPMODE   BASE_ADDR+0x5000
IOPCON0   BASE_ADDR+0x5004
TMOD      BASE_ADDR+0x6000
TDATA0    BASE_ADDR+0x6004
TNC1      BASE_ADDR+0x6010
```

A.3 Initialisierung der Pins und des Timer

```
void pins_init()
{
  // Pin 19 auf 1 --> ist Ausgang für Emittergrundschaltung
  *(volatile unsigned int *)IOPMODE=*(volatile unsigned int *)IOPMODE | (1 << 1);
  // Pin 18 auf 0 --> ist Eingang für den Reset
  *(volatile unsigned int *)IOPMODE=*(volatile unsigned int *)IOPMODE & ~(1 << 6);
  // Pin 17 auf 0 --> ist Eingang für die Datenleitung
  *(volatile unsigned int *)IOPMODE=*(volatile unsigned int *)IOPMODE & ~(1 << 8);
}
void timer_init()
{
  // Pin 3 auf Timer 1 konfigurieren
  *(volatile unsigned int *)IOPCON0=*(volatile unsigned int *)IOPCON0 | (1 << 31);
  // zu zählende Takte definieren --> ergibt 123usec
  *(volatile unsigned int *)TDATA1=6150;
  // Toggle Mode definieren
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD | (1 << 4);
  // erster Zustans soll 0 sein, bei Timer=0 auf 1 schalten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD & ~(1 << 5);
}
```

A.4 Funktionen zum Lesen und Schreiben der Bits

```
unsigned char read_io_bit()
{
  // liest das Bit auf der Datenleitung ein
  volatile unsigned int inhalt;
  inhalt=*(volatile unsigned int *)IOPDATA & (1 << 8);
  if(inhalt==0) return 0;
  return 1;
}
unsigned char read_reset_bit()
{
  // Bit auf der Resetleitung lesen
  volatile unsigned int inhalt;
  inhalt=*(volatile unsigned int *)IOPDATA & (1 << 6);
  if(inhalt==0) return 0;
  return 1;
}
void set_io_bit(unsigned char bit)
{
  // Bit auf der Datenleitung setzen
  if(bit==1)
  {
    *(volatile unsigned int *)IOPDATA=*(volatile unsigned int *)IOPDATA | (1 << 1);
  }
  else
  {
    *(volatile unsigned int *)IOPDATA=*(volatile unsigned int *)IOPDATA & ~(1 << 1);
  }
}
```

A.5 Erzeugung der Bitzeiten

```
void wait_bit_time()
{
  // Funktion erzeugt eine Bitzeit über den Timer 1
  // Timer anhalten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD & ~(1 << 3);
  // zu zählende Takte definieren --> ergibt 123usec --> 1 Bit
  *(volatile unsigned int *)TDATA1=6010;
  // Timer starten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD | (1 << 3);
  // Pollen bis Timer abgelaufen ist, mit kleinem Störabstand
  while(*(volatile unsigned int *)TNC1>=30);
  // Timer anhalten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD & ~(1 << 3);
}
void wait_bit_time_and_a_half()
{
  // Funktion erzeugt eineinhalb Bitzeiten über den Timer 1
  // Timer anhalten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD & ~(1 << 3);
  // zu zählende Takte definieren --> ergibt 185usec --> 1,5 Bit
  *(volatile unsigned int *)TDATA1=9010;
  // Timer starten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD | (1 << 3);
  // Pollen bis Timer abgelaufen ist, mit kleinem Störabstand
  while(*(volatile unsigned int *)TNC1>=30);
  // Timer anhalten
  *(volatile unsigned int *)TMOD=*(volatile unsigned int *)TMOD & ~(1 << 3);
}
```

A.6 Funktion zur ATR Erzeugung

```
void generate_atr(char atr[], int length)
{
  // die ATR erzeugen
  int j;
  set_io_bit(0); // IO Bit erst mal auf 0 legen, Transistor hochohmig, Leitung hat 5V
  while(read_reset_bit()==1); // Polling bis Reset auf 0 gezogen wird
  while(read_reset_bit()==0); // Polling bis Reset auf 1 gezogen wird (Reset ist Impuls)
  for(j=0;j<10000;j++) wait_bit_time(); // warte 10000 etu
  for(j=0;j<=length-1;j++) send_io_byte(invert_bit_sequence(atr[j]),0,0); // Bytes der ATR
  for(j=0;j<6860;j++) wait_bit_time(); // warte 6860 etu
  // die ATR muss laut ISO nochmal gesendet werden
  for(j=0;j<=length-1;j++) send_io_byte(invert_bit_sequence(atr[j]),0,0); // Bytes der ATR
}
```

A.7 Funktion zum Einlesen von Kommandos

```
int wait_for_command(unsigned char *commandbuffer, unsigned char *databuffer, unsigned char *dataanz)
{
  // ein Kommando vom Kartenleser einlesen
  int i,j = 0;
  unsigned char command[5];
  unsigned char cla, ins, p1, p2, p3;
  if(read_bytes(5,command)==1)
  {
    printf("ERROR\n");
    generate_return_code(0x68,0x00);
    return 1;
  }
  cla=command[0]; // der Übersichtlichkeit halber zwischenspeichern
  ins=command[1];
  p1=command[2];
  p2=command[3];
  p3=command[4];
  for(i=0;i<=7;i++) *(commandbuffer+i)=command[i];
  if(cla==CLA)
  {
    switch(ins)
    {
      case INS_WITH_DATA: // warte Störabstand
                          for(j=0;j<=STOERABSTAND;j++) wait_bit_time();
                          // sende das INS Byte, um Lesebereitschaft zu signalisieren
                          send_io_byte(invert_bit_sequence(ins),1,1);
                          read_bytes(p3,databuffer); // Daten vom Leser einlesen, Anzahl=p3
                          *dataanz=p3;
                          return INS_WITH_DATA;
      case INS_WITHOUT_DATA: return INS_WITHOUT_DATA;
      case INS_GET_DATA: return INS_GET_DATA;
      default: return INS_NOT_PROVIDED;
    }
  }
  else
  {
    return CLA_NOT_PROVIDED;
  }
}
```

A.8 Aufruf der Funktion zum Einlesen von Kommandos

```
while(1)
  {
    switch(wait_for_command(command,data,&dataanz))
    {
      case 1: //mache nichts
              break;
      case INS_WITH_DATA: // Daten von Leser
                          print_command_and_data(&command,&data,dataanz);
                          generate_return_code(0x61,0x05);
                          break;
      case INS_WITHOUT_DATA: // Kommando ohne Daten
                             print_command(&command);
                             generate_return_code(0x90,0x00);
                             break;
      case INS_GET_DATA: // Leser fordert Daten an
                         print_command(&command);
                         generate_data_and_return_code(command[1],answer_data,5,0x90,0x00);
                         break;
      case INS_NOT_PROVIDED: // INS Byte wird nicht unterstützt
                             generate_return_code(0x6A,0x81);
                             break;
      case CLA_NOT_PROVIDED: // CLA Byte wird nicht unterstützt
                             generate_return_code(0x68,0x00);
                             break;
      default: break;
    }
  }
```

A.9 Funktion zum Erzeugen eines Bytes

```
void send_io_byte(unsigned char byte,int ende,int schutz)
{
  // ein Byte auf die Leitung legen, mit Start-,Paritäts- und Stopbits
  int i=0,s=0;
  unsigned char bits[8];
  unsigned char fehler;
  // Emitterschaltung INVERTIERT, d.h. Signal invers auf die Leitung legen
  gauss_algorithm(byte,bits);
  invert_bits(bits); // Bits müssen invers gesendet werden, Emitterschaltung
  set_io_bit(1); // ein Startbit, durch 1 leitet Transistor, Leitung wird auf Masse gezogen
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[0]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[1]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[2]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[3]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[4]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[5]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[6]);
```

```
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(bits[7]);
  wait_bit_time(); // ein Bit lang warten
  set_io_bit((~generate_parity(bits)) & 1); // Parität setzen muß auch invertiert werden
  wait_bit_time(); // ein Bit lang warten
  set_io_bit(0); // Stopbit
  wait_bit_time(); // ein Bit lang warten
  fehler=read_io_bit(); // prüfen, ob Reader Paritätsfehler signalisiert
  if(ende==0) // beim letzen Byte einer Folge muss nicht weiter gewartet werden
  {
    wait_bit_time(); // ein Bit lang warten
    if(schutz==1) // die zusätzliche Schutzzeit wird erst nach der ATR verwendet
    {
      wait_bit_time(); wait_bit_time();
      wait_bit_time(); wait_bit_time();
    }
  }
  if(fehler==0) send_io_byte(byte, ende, schutz); // bei einem Paritätsfehler Byte nochmal senden
}
```

A.10 Hilfsfunktionen

```
void invert_bits(unsigned char * bitarray)
{
  // jedes Bit der Bitfolge umdrehen
  int i=0;
  for(i=0;i<=7;i++)
  {
    *(bitarray+i)=~*(bitarray+i); // jedes Bit invertieren
    *(bitarray+i)=*(bitarray+i) & 1; // maskieren des LSB, alle anderen auf 0
  }
}
unsigned char invert_bit_sequence(unsigned char number)
{
  // dreht die Reihenfolge der Bits um
  unsigned char bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7;
  bit0=number & 1; // erst mal alle Bits erzeugen
  bit1=(number & 2) >> 1;
  bit2=(number & 4) >> 2;
  bit3=(number & 8) >> 3;
  bit4=(number & 16) >> 4;
  bit5=(number & 32) >> 5;
  bit6=(number & 64) >> 6;
  bit7=(number & 128) >> 7;
  return (bit7*1+ // jetzt die Bits in umgekerter Reihenfolge aufschreiben
          bit6*2+
          bit5*4+
          bit4*8+
          bit3*16+
          bit2*32+
          bit1*64+
          bit0*128);
}
```

```
void gauss_algorithm(unsigned char number,unsigned char *bits)
{
  // macht aus einer Dezimalzahl eine Binärzahl
  int i=0;
  for(i=7;i>=0;i--)
  {
    *(bits+i)=number % 2;
    number=number / 2;
  }
}
unsigned char generate_parity(unsigned char * bits)
{
  // gerade Parität wird verwendet, das Paritätsbit erweitert
  // die Anzahl der Einsen auf einen geraden Wert
  int i,anz=0;
  for(i=0;i<=7;i++)
  {
    if(*(bits+i)==1) anz++; // falls ein Bit 1 ist, wird anz inkrementiert
  }
  if((anz % 2)==0) return 0; // Anzahl der Einsen ist gerade, Parität 0 zurückgeben
  return 1; // Anzahl der Einsen ist ungerade, Parität 1 zurückgeben
}
```

A.11 Funktionen zum Erzeugen der Antworten

```
void generate_return_code(unsigned char sw1,unsigned char sw2)
{
  // einen zwei Byte Returncode auf die Datenleitung legen
  int i,j;
  for(j=0;j<=STOERABSTAND;j++) wait_bit_time(); // warte Störabstand
  send_io_byte(invert_bit_sequence(sw1),0,1); // Antwort erzeugen
  send_io_byte(invert_bit_sequence(sw2),1,1);
}

void generate_data_and_return_code(unsigned char ins,unsigned char *data,int data_length,
                                   unsigned char sw1,unsigned char sw2)
{
  // Daten und einen zwei Byte Returncode auf die Leitung legen
  int i,j;
  for(j=0;j<=STOERABSTAND;j++) wait_bit_time(); // warte Störabstand
  send_io_byte(invert_bit_sequence(ins),0,1); // erstes Datebbyte ist das ins Byte
  for(j=0;j<data_length;j++)
  {
    send_io_byte(invert_bit_sequence(*(data+j)),0,1);
  }
  send_io_byte(invert_bit_sequence(sw1),0,1);
  send_io_byte(invert_bit_sequence(sw2),1,1);
}
```

B Java Klassen

Anhang B gibt einen Überblick über die benötigten Klassen für das T=0 Protokoll. Die nativen Funktionen in der Kertasarie VM sind nicht aufgeführt, sie entsprechen im wesentlichen der C-Funktionen mit entsprechenden Schnittstellen, die ein Aufrufen aus einer Java Klasse ermöglichen.

B.1 Klasse Testcard

```
public class Testcard extends Cardcomm
{
  private final short CLA=0x00;
  private final short INS_WITHOUT_DATA=0xA4;
  private final short INS_WITH_DATA=0x20;
  private final short INS_GET_DATA=0xC0;

  private void printHexadecimal(short number)
  {
    // Ausgabe eines Bytes in hexadezimaler Schreibweise
    int buffer;
    buffer=(number & 0xF0) >> 4;
    if(buffer<10) System.out.print(buffer);
      else System.out.print((char)(0x41+(buffer-10)));
    buffer=(number & 0x0F);
    if(buffer<10) System.out.print(buffer);
      else System.out.print((char)(0x41+(buffer-10)));
  }
  private void printCommand(short cla, short ins, short p1, short p2, short p3)
  {
    // Kommando ausgeben
    System.out.print("Kommando: cla="); printHexadecimal(cla);
    System.out.print(" ins="); printHexadecimal(ins);
    System.out.print(" p1="); printHexadecimal(p1);
    System.out.print(" p2="); printHexadecimal(p2);
    System.out.print(" p3="); printHexadecimal(p3);
    System.out.println("");
  }
  private void printData(short[] data)
  {
    // Daten ausgeben
    if(data==null)
    {
      System.out.println("keine Daten");
    }
    else
    {
        System.out.print("Daten: ");
      for(int i=0;i<data.length;i++) printHexadecimal(data[i]);
      System.out.println("");
    }
  }
}
```

```
  public void process(CommandTPDU ctpdu)
  {
    short cla=ctpdu.getCla();
    short ins=ctpdu.getIns();
    printCommand(cla,
                 ins,
                 ctpdu.getP1(),
                 ctpdu.getP2(),
                 ctpdu.getP3());
    if(cla==CLA)
    {
      switch(ins)
      {
        case INS_WITHOUT_DATA: // Kommando ohne Daten
                               this.sendResponse(new ReturnTPDU((short)0,(short)0x90,(short)0x00,null));
                               break;
        case INS_WITH_DATA: // Kommando mit Daten
                            printData(ctpdu.getData());
                            this.sendResponse(new ReturnTPDU((short)0,(short)0x61,(short)0x05,null));
                            break;
        case INS_GET_DATA: // Leser fordert Daten an, mit 5 Byte antworten
                           short rdata[]={(short)0xAA,
                                          (short)0xBB,
                                          (short)0xCC,
                                          (short)0xDD,
                                          (short)0xEE};
                           this.sendResponse(new ReturnTPDU((short)ins,(short)0x90,(short)0x00,rdata));
                           break;
        default: // Instruction Byte wird nicht unterstützt
                 this.sendResponse(new ReturnTPDU((short)0,(short)0x6A,(short)0x81,null));
      }
    }
    else
    {
      // Class Byte wird nicht unterstützt
      this.sendResponse(new ReturnTPDU((short)0,(short)0x68,(short)0x00,null));
    }
  }
  public static void main(String args[])
  {
    new Testcard();
  }
}
```

B.2 Klasse CommandTPDU

```
public class CommandTPDU
{
  private short cla;
  private short ins;
  private short p1;
  private short p2;
  private short p3;
  private short data[];
  public CommandTPDU(short data[]) // Kommando ohne Daten
  {
    int i;
    cla=data[0];
    ins=data[1];
    p1=data[2];
    p2=data[3];
    p3=data[4];
    if(p3!=0)
    {
      this.data=new short[p3];
      for(i=0;i<p3;i++) this.data[i]=data[i+5];
    }
  }
  public short getCla()
  {
    return cla;
  }
  public short getIns()
  {
    return ins;
  }
  public short getP1()
  {
    return p1;
  }
  public short getP2()
  {
    return p2;
  }
  public short getP3()
  {
    return p3;
  }
  public short[] getData()
  {
    return this.data;
  }
}
```

B.3 Klasse ReturnTPDU

```
public class ReturnTPDU
{
  private short ins;
  private short sw1;
  private short sw2;
  private short data[];
  public ReturnTPDU(short ins, short sw1, short sw2, short data[])
  {
    int i;
    this.ins=ins;
    this.sw1=sw1;
    this.sw2=sw2;
    if(data!=null)
    {
      this.data=new short[data.length];
      for(i=0;i<data.length;i++) this.data[i]=data[i];
    }
  }
  public short getIns()
  {
  }
  public short getSw1()
  {
    return sw1;
  }
  public short getSw2()
  {
    return sw2;
  }
  public short[] getData()
  {
    return this.data;
  }
}
```

B.4 Klasse Cardcomm

```
public abstract class Cardcomm
{
  private native byte[] waitForCommand();
  public native void sendATR();
  public native void generateReturnCode(int ins,int sw1,int sw2,int dataanz,byte[] data);
  private short makeShort(byte number)
  {
    if(number<0) return (short)(256+number);
    return (short)number;
  }
  public Cardcomm()
  {
    int i;
    this.sendATR();
    while(true)
    {
      byte[] daten=waitForCommand();
      short[] sdaten=new short[daten.length];
      for(i=0;i<daten.length;i++) sdaten[i]=makeShort(daten[i]);
      CommandTPDU ctpdu=new CommandTPDU(sdaten);
      this.process(ctpdu);
    }
  }
  public void sendResponse(ReturnTPDU rtpdu)
  {
    short data[];
    data=rtpdu.getData();
    if(data==null)
    {
      generateReturnCode((int)rtpdu.getIns(),(int)rtpdu.getSw1(),(int)rtpdu.getSw2(),0,null);
    }
    else
    {
      byte[] bdata=new byte[data.length];
      for(int i=0;i<data.length;i++) bdata[i]=(byte)data[i];
      generateReturnCode((int)rtpdu.getIns(),(int)rtpdu.getSw1(),(int)rtpdu.getSw2(),data.length,bdata);
    }
  }
  public abstract void process(CommandTPDU ctpdu);
}
```