INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Vergleich von JAVA- und Native-Chipkarten
Toolchains, Benchmarking, Messumgebung

Mario Fischer

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Ralf König (LMU)
Bernhard Lippmann (infineon)
Harald Rölle (LMU)

Abgabetermin: 31. Mai 2006

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Vergleich von JAVA- und Native-Chipkarten
Toolchains, Benchmarking, Messumgebung

Mario Fischer

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Ralf König (LMU)
Bernhard Lippmann (infineon)
Harald Rölle (LMU)

Abgabetermin: 31. Mai 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Zusammenfassung

In diesem, bei der Firma infineon durchgeführten Fortgeschrittenen-Praktikum soll zum einen versucht werden, einen Performance-Vergleich zwischen Java-basierten Chipkarten und ihren in Maschinensprache programmierten Pendants (Native-Karten) zu erstellen, sowie daraus abzuleiten, ob aus diesen Messergebnissen auch Schlüsse auf die Performance unbekannter Karten gezogen werden dürfen.

Des weiteren soll die bisher dort verwendete Testumgebung für Laufzeitmessungen („JNut-Box“) um eine Anwendungs-Softwareseite erweitert, sowie mögliche (und bereits erfolgte) Erweiterungen und Verbesserungen an den bisherigen Komponenten (Hard- und Firmware der JNut-Box) dargestellt und diskutiert werden.

2 Einleitung

Seit geraumer Zeit schon sind Chipkarten im täglichen Leben verbreitet. Den Anfang machte im Dezember 1986 die Telefonkarte, dort fielen die goldenen Kontakte zur Kommunikation mit der dahinterliegenden Elektronik zum ersten mal auf, damals noch mit einer primitiven, aber durchaus sicheren Schaltung.

Seitdem findet man sie in immer mehr Anwendungsgebieten, wobei der wahrscheinlich größte Boom der Karten mit integrierter Intelligenz wahrscheinlich im alltäglichen Leben garnicht der auffälligste ist: Im Fahrtwind der rasch wachsenden Zahl von Mobilfunk-Teilnehmern stieg natürlich auch die Verbreitung der SIM-Karten an, welche den Netztteilnehmer gegenüber dem Betreiber authentisieren.

Mittlerweile werden die Kleinstcomputer in den Karten immer leistungsfähiger, und damit verbunden auch freundlicher in der Programmierung: Seit einigen Jahren nun gibt es sogenannte Java Cards: Prozessoren, die einen Interpreter für in Java (genauer: einer abgespeckten Version davon) geschriebene Applikationen besitzen. Damit rückt die Chipkarte aus der Ecke der schwer zu programmierenden (Assembler oder bestenfalls C) Hardware-Plattformen und wird deshalb in naher Zukunft noch weitere Verbreitung finden.

2.1 Motivation

Deshalb stellt sich die Frage, wie es um die Leistungsfähigkeit heutiger Java Cards im Vergleich mit ihrern entsprechenden Karten ohne der Virtual Machine (die sogenannten „Native“-Karten, die nach wie vor in C oder Assembler programmiert werden) bestellt ist.

2.2 Aufgabenstellung

Zunächst sollten Grundlagen der Chipkarten erworben sowie deren „traditionelle“ Programmierung erlernt werden. Dies geschah mit der infineon-Toolchain bestehend aus der integrierten Entwicklungsumgebung (IDE) von KEIL, sowie den infineon-SmartMask-Karten (Abb. 7), die mehrmaliges Laden und Ausführen eigener Programme auf die Karte gestatten. Auch eine JavaCard-Toolchain (die einzig aus der „Aspects“-IDE besteht) sollte kennengelernt werden.

Dann sollte ein einem Java Benchmark Applet entsprechendes Programm als Native-Assembler-Version erstellt werden, um diese beiden dann gegeneinander zu testen. Anhand der Ergebnisse sollte ermittelt werden, ob damit Schlüsse auf andere Karten zulässig wären.

Schliesslich sollte noch das vorhandene Testwerkzeug, die „JNut-Box“ komfortabler bedienbar gemacht und Vorschläge zu dessen Verbesserung gemacht werden.

3 Grundlagen zu Chipkarten

Chipkarten, oft auch als Smartcard oder Integrated Circuit Card (ICC) bezeichnet[6], sind Plastikkarten mit eingebautem Chip, der eine Hardware-Logik, Speicher oder auch einen Mikroprozessor enthält.

Die Erfindung dieser Karte geht auf mehrere Personen zurück, der früheste war Jürgen Dethloff 1968 mit seinem Patent, „einen integrierten Schaltkreis in eine Identifikationskarte einzubauen“.

3.1 Aufgaben von Chipkarten

Grundsätzliche Aufgabe einer Chipkarte ist das Speichern, Verwalten und schnelle zur Verfügung stellen von (im Vergleich zu anderen Datenträgern geringen) Datenmengen: Hauptunterschied zu anderen Speichermedien ist jedoch der Schutz dieser Daten vor Veränderung und sehr oft auch gegen deren bloßes Auslesen:

Dies wird erreicht, indem der Zugriff auf den internen Speicher der Karte nicht direkt nach außen zugänglich gemacht wird, sondern nur über eine dazwischengeschaltene Instanz (quasi eine Art „Pförtner“) möglich ist, die sämtliche Zugriffe darauf nur nach Authentisierung und bei ausreichender Berechtigung gestattet. Diese Instanz ist entweder in Form einer „fest verdrahteten“ Programmlogik ausgeführt oder durch die Software eines Mikroprozessors, je nach Komplexität der Aufgabe (siehe Abschnitt „Klassifikation“).

3.2 Aufbau von Chipkarten

im Inneren unterscheidet sich die Architektur erheblich, vom reinen Speicherbaustein mit direktem Vollzugriff darauf, bis zum komplexen Krypto-Prozessor.

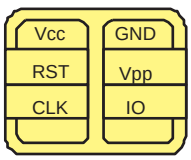

Abbildung 1: Chipkarten-Kontaktierung

Einzige Gemeinsamkeit aller Karten ist sind die genormten goldenen Kontakte auf der Oberfläche[1, S. 32]. Deren Aufgaben (Abb. 1) sind:

- GND: Bezugspotential
- Vcc: Versorgungsspannung für die Elektronik
- IO: Bidirektionale, serielle Kommunikation
- CLK: Takteingang für Synchronisation der Kommukikation (bei synchronen Karten) oder als Prozessortakt (asynchrone Kommunikation)
- RST: Auslösen eines Resets auf der Karte

- Vpp: Versorgungsspannung für Ladungspumpe beim Beschreiben von nicht flüchtigem Speicher (Non volatile memory, NVM).

Das Innenleben der Karten hingegen kann je nach Aufgabe stark variieren.

3.3 Klassifikation von Chipkarten

Eine kurze Klassifikation der Karten nach praktischen Bestandteilen soll hier nach folgenden Merkmalen gegeben werden:

- Programmlogik: Mit oder ohne CPU, oder fest verdrahtete Logik.
- Kommunikation: Synchron oder Asynchron (siehe Kapitel „Datenübertragung“)
- Extra-Hardware: Krypto-Koprozessoren ([a]symmetrrische Verschlüsselung, digitale Signatur, Hashwerte, Zufallszahlen, ...), UART[1] (serielle, asynchrone und gepufferte Kommunikation).
- Nicht flüchtiger Speicher (Non-volatile memory, NVM): Gruppierbar in:
  - (Read-only memory, ROM (Abb. 8)): Lesbar, aber nur einmal beschreibbar, oder bereits ab Werk beschrieben:
  - EEPROM[2]/Flash: Für konstante Daten und Software. Lesbar und mehrmals beschreibbar (im Vgl. zu RAM jedoch sehr langsam). Vergleichbar mit den Aufgaben einer PC-Festplatte.
- Anwendungsentwicklung (bei Karten mit CPU): Programmierung in Assembler oder C, oder Java. Übertragen der Software auf die Karte durch Belichten des ROMs in der Wafer-Fabrik oder Beschreiben („flashen“) des NVMs der Karte beim Entwickler selbst.

Die Datenpfade innerhalb einer Karte (Abb. 2) entsprechen denen zwischen den Hardware-Komponenten aller gängigen Kleincomputer: Die CPU kontrolliert die Kommunikation nach Außen sowie alle Speicherperipherie, nur Krypto-Koprozessoren haben ebenfalls direkten Zugriff auf den Arbeitsspeicher (Direct Memory Acess, DMA).

Für viele Kombinationen dieser Merkmale lassen sich Anwendungseispiele finden, ein paar bekannte Beispiele[1, S. 16 ff.]:

- Krankenversicherungskarte: keine CPU (quasi-direkter Zugriff auf den NVM), synchrone Kommunikation, keine Extra-HW., NVM.
- Telefonkarte: fest verdrahtete Logik (mit komplexem kryptographischem Zugriffsprotokoll), synchrone Kommunikation, keine Extra-HW., NVM.
- SIM-Karte in Mobiltelefonen: CPU, asynchrone Kommunikation, NVM.
- Pay-TV-Karte, GeldKarte: CPU, asynchrone Kommunikation, NVM, Krypto-Koprozessoren.

[1] Universal Asynchronous Receiver Transmitter
[2] Electrically Eraseable Programmable Read Only Memor

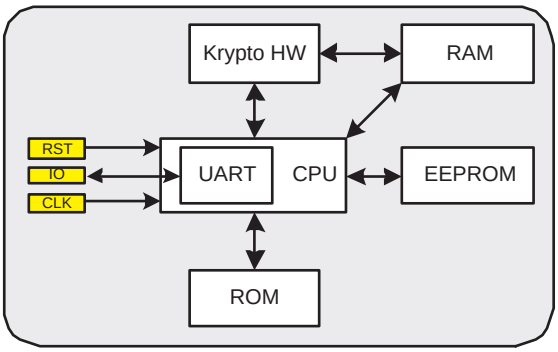

Abbildung 2: Datenpfade in der Kartenhardware

3.4 Datenübertragung

3.4.1 Bitübertragungsschicht

Allen Karten gemein ist, daß die Übertragung bidirektional und im Halb-Duplex allein über den IO-Pin läuft. Die Übertragungsprotokolle spalten sich jedoch in 2 Gruppen auf:

- Synchrone Kommunikation: Die Bitdauer hängt mit dem Takt auf dem CLK-Pin unmitelbar zusammen. Es gibt zahlreiche synchrone Protokolle (I2C-Bus, Two-Wire-Bus, ...), die regeln, welche Seite wann und wieviel senden kann/darf/muß. Diese Kommunikationsart wird vor allem bei billigen oder einfachen Karten angewandt (Telefonkarte, Krankenversicherungskarte, ...) und soll hier nicht weiter diskutiert werden.
- Asynchrone Kommunikation:

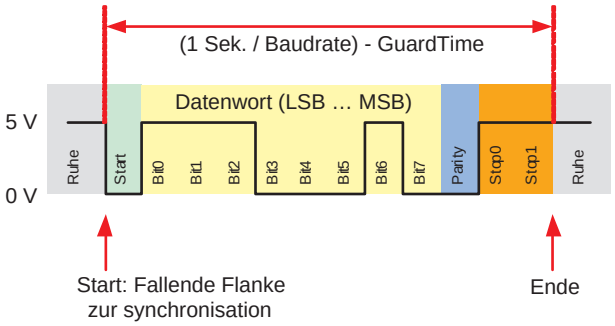

Abbildung 3: Asynchrone Übertragung des ASCII-Zeichens `'G'` (0x47).

Bei diesem Übertragungsverfahren, daß bis auf die Spannungspegel und die Bidirektionalität der Spezifikation der seriellen PC-Schnittstelle[7] entspricht, müssen Sender und Empfänger im Vorraus die Übertragungsgeschwindigkeit und Formatparameter kennen. Standard bei Chipkarten sind nach dem Reset 9600 Baud, 1 Startbit, 8 Datenbits, gerade Parität, 2 Stoppbits, (insbesondere die Baudrate darf jedoch geändert werden, wenn höhere Protokolle dies aushandeln).

Es herrscht 5 Volt Ruhepegel auf der Leitung. Mit der (immer) fallenden Flanke des Startbits synchronisiert sich der Empfänger. Es folgen die 8 Datenbits (LSB zuerst), dann ein Parity-Bit, daß die Anzahl gesendeter 1en auf eine gerade Zahl ergännzt sowie 2 Stoppbits, die dann wieder den Ruhepegel herstellen. Eine GuardTime spezifiziert den Mindestabstand zwischen (letztem) Stopp- und dem folgenden Startbit.

Auf dieser untersten Schicht der Halbduplex-Byteübertragung setzen höhere Transportprotokolle nach ISO/IEC 7816 auf, die regeln, welche Seite wann und wieviel überträgt, und sich um Zerlegung, Zusammensetzung, Wiederholung, Fehlerprüfung und Timeouts kümmern. Sie werden mit „T=0“, „T=1“ bis „T=15“ bezeichnet, wobei die ersten beiden die größte praktische Bedeutung haben. Alle diese Protokolle benutzen Daten, welche folgendermaßen strukturiert sind:

3.4.2 Datenformate

ATR: Nach dem Reset-Impuls des Chipkarten-Terminals sendet jede asynchrone Chipkarte immer eine „Answer to Reset“ (ATR): Innerhalb dieses ATRs teilt sie dem Terminal (unter anderem) mit, welche Protokolle sie beherrscht [1, S. 68 ff.], [4, S. 330 ff.].

Der Aufbau eines ATRs ist relativ komplex und sei hier nur sehr grob Umrissen (Abb. 4):
Zurest wird der Initial Character TS gesendet, der als Synchronsationsmarke dient, er ist bei Karten mit „LSB ... MSB “-Reihenfolge immer 0x3B. Danach folgt der Format-Character TS, der ein Bitmuster über die Gültigkeit der folgenden Interface-Character $TA_1 \ldots TD_1$ enthält. Der letzte dieser Interface Character codiert wiederum die folgende 4er-Gruppe an Interface Characters und so weiter, ähnlich einer verketteten Liste. In den Interface Characters werden u.A. der verwendete Protokolltyp und seine Parameter codiert. Nach den Interface Characters können sogenannte „Historical Characters“ T1 ... Tk folgen (ob sie folgen, ist ebenfalls in T0 codiert). ISO 7816 spezifiziert deren Bedeutung jedoch nicht. Abschliessend kommt noch das Prüfzeichen TCK, ein XOR-Wert aller bisherigen Zeichen.

Eine Chipkarte darf durchaus mehrere ATRs anbieten: Falls das Terminal mit einem bestimmten ATR nichts anfangen kann, löst es einen Soft-Reset[3] aus, und die Karte sendet einen neuen ATR-Kandidaten[8].

APDU: Eine „Application Protocol Data Unit“ (APDU) ist die Kommunikationseinheit zwischen der Karte und dem Terminal (ebenfalls in ISO 7816 spezifiziert). Die vom Terminal gesendeten APDUs heissen Command APDUs (Kommandos), die von der Karte Response APDUs (Antworten).

- Ein Kommando (Abb. 5 oben) ist wie folgt aufgebaut:

[3] Ohne Unterbrechung der Versorgungsspannung!

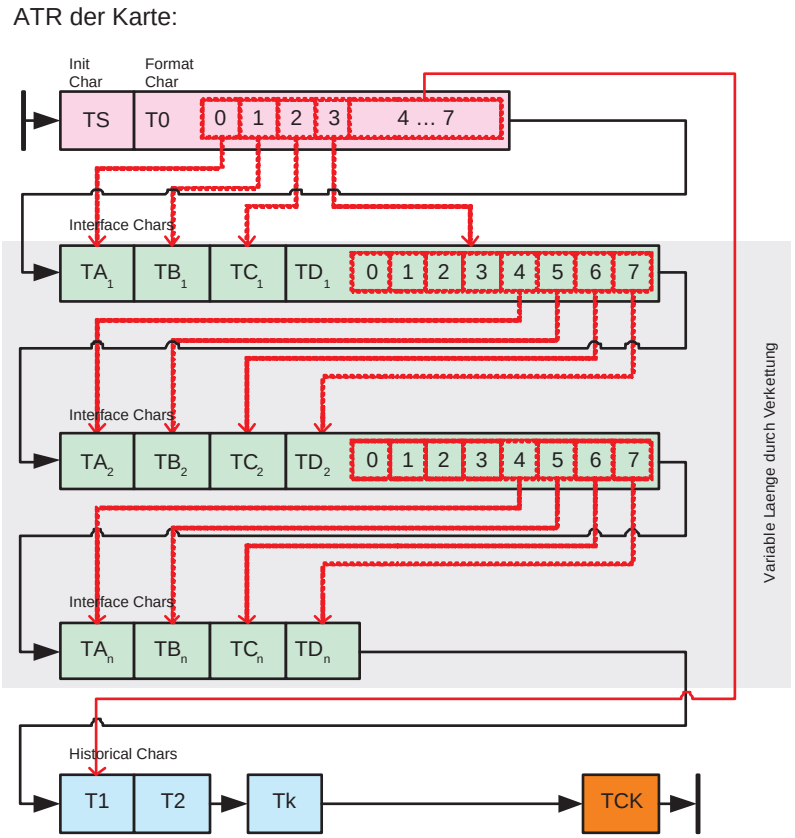

Abbildung 4: Format des ATRs einer Karte (Rote Linien zeigen, welches Bit das Vorhandensein weiterer Zeichen enkodiert)

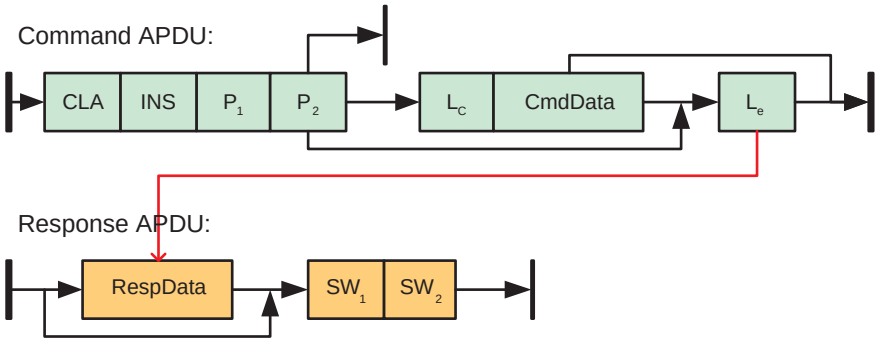

Abbildung 5: Kommando- und Antwort-APDUs mit ihren optionalen Datenfeldern

  - Header: Zuerst kommt ein Class-Byte (CLA), daß die Befehlsklasse spezifiziert, gefolgt vom eigentlichen Befehl (Instruction, INS), sowie 2 Parametern P_1, P_2.
  - Body: kann leer sein, oder optional spezifizieren wie viel Daten noch gesendet werden (Längenkennung L_c + CmdData), sowie noch optional spezifizieren, wie viele Bytes als Antwort die Karte zurücksenden darf (L_e).

- Eine Antwort (Abb. 5 unten) ist wie folgt aufgebaut[1, S. 84]:
  - Optional Antwortdaten (RespData, mit Länge L_e aus der Kommando-APDU),
  - sowie immer den 2 Statusbytes SW_1, SW_2.

Damit ergeben sich 4 Kombinationen aus Kommando mit/ohne Daten und Antwort mit/ohne Daten.

3.4.3 Höhere Protokolle und Schnittstellen

Das T=0 Protokoll: Durch seine Anwendung in den SIM-Karten von GSM-Telefonen dürfte dieses Protokoll[2, S. 60 ff.] die weiteste Verbreitung haben: Es spezifiziert Teile innerhalb des ATRs, einen Low-Pegel auf der IO-Leitung im Idle-Zustand (der ja eigentlich high wäre) um der Gegenseite einen Übertragungsfehler anzuzeigen (detetktiert durch Ungültiges Parity-Bit), sowie eine Verfeinerung der Kommando APDU (In CmdData steht eine weitere Längenkennung P_3 für noch-zu-übertragende Bytes, Datenrichtung abhängig vom Kommando selber).

Das T=1 Protokoll: Dieses Protokoll wird in JavaCards verwendet und soll deshalb ebenfalls kurz vorgestellt werden: Es ist blockorientiert und wird sowohl zwischen Karte und Terminal, also auch zwischen dem Anwendungsprogramm und der Schnittstelle zum Terminal „gesprochen“ (PC/SC-Schnittstelle). Es gibt Informations-, Empfangsbestätigungs- und System-Blöcke (I-, R-, S-Blöcke). Nutzdaten stehen im I-Block, R-Blöcke gibt es nur zwischen Anwendung und Terminal, S-Blöcke nur zwischen Karte und Terminal. Des weiteren definiert T=1 u.A. Möglichkeiten zum Baudratenwechsel (enkodiert im ATR), sowie „Zwischenmeldungen“ der Karte bei langen Antwortszeiten.

Die PC/SC-Schnittstelle: Die abstrakte PC/SC („Personal Computer / Smart Card“) Software-Schnittstelle[9] für PC-Systeme dient dazu, daß Software-Anwendungen mit beliebigen Terminals zusammenarbeiten können, ohne seine konkreten Eigenheiten zu kennen. Die Anwendung braucht sich nicht darum zu kümmern, ob das Terminal über USB, die seriellen Schnittstelle, TCP/IP, etc. angeschlossen ist, oder wie z.B. ein Soft-Reset der Karte durch das Terminal erfolgen muss. All dies ist durch Schnittstellenfunktionen abstrahiert.

Aus Anwendungssicht findet ein T=1-artiger Blocktransport der Daten nach dem Request-Response-Schema statt. Die Schnittstelle definiert ebenfalls weitere Funktionen wie z.B. das Ermitteln aller an den PC angeschlossenen Terminals, automatischen Kartenauswurf, oder Anzeigen auf einem integrierten Display, falls das Terminal das jeweilige Feature unterstützen sollte.

3.5 Software auf Chipkarten

Hauptaugenmerk für den Entwickler von On-Card-Software sind (neben kryptographischen Sicherheitsaspekten natürlich) die beschränkten Ressourcen (Speicher, Rechenleistung, Registeranzahl und -größe, IO-Geschwindigkeit) einer Chipkarte: Die Entwicklung solcher Software ist daher am ehesten mit der Entwicklung von Microcontroller-Software vergleichbar: Den Komfort vieler Hochsprachen für PC-Software gibt es hier nicht (was schon bei vergleichsweise simplen Dingen wie Strings und deren automatisches Längen-Management beginnt), Hauptwerkzeuge sind Assembler, C und seit neuestem auch Java (mit sprachlichen Einschränkungen). Zwei Entwicklungsumgebungen, die bei infineon ebenfalls verwendet werden, sollen exemplarisch vorgestellt werden:

3.5.1 Native-Cards

Die integrierte Entwicklungsumgebung (IDE) „μVision“ von KEIL (Abb. 6) enthält eine Projektverwaltung (linke Leiste), einen Quellcode-Editor (grosses Fenster), C-Compiler und Assembler für zahlreiche Prozessortypen (mit der üblichen Anzeige von Fehlern und Warnungen beim Compilieren in der unteren Leiste). Beim debuggen (auf Quelltext-Ebene) werden einem die von PC-Entwicklungsumgebungen gewohnten Features wie „Haltepunkte setzen“ oder „Variablenwert anzeigen“ geboten. Es können die unterstützten Prozessoren ebenfalls simuliert werden, deren Register kann beim debuggen ebenfalls untersucht werden.

Nach der Entwicklung, Simulation und Debugging der Software kann man die Software dann zu Testzwecken auf sogenannte SmartMask-Cards (Abb. 7) überspielen: Diese speziellen (und nur für den internen Gebrauch gedachten) Karten speichern den Maschinencode auf ihrem EEPROM und starten es die nächsten n mal nach einem Reset (den Wert für n kann man in der SmartMask-Ladesoftware auf dem PC einstellen). Nach dem $n+1$. Reset übernimmt wieder der auf der Karte sitzende SmartMask-Lader die Kontrolle und man kann wieder ein eigenes Programm laden.

Hat das Programm auch diese Phase der Evaluierung bestanden, kann sie in Großserie gehen: Die Software wird dann nicht mehr auf dem EEPROM abgespeichert, sondern in den ROM (Abb. 8) der Karte geätzt (Wafer-Fabrik). Jetzige Änderungen an der Software sind nicht mehr möglich.

Alternativ dazu existieren auch kommerzielle Karten nach der SmartMask-Methode, die man (in etwas kleinerer Serie) selber „flashen“ kann. Nachteilig daran ist jedoch, daß die Karten teurer als ihre ROM-Pendants sind, und Teile des EEPROM-Speichers nicht mehr zur Datenspeicherung zur Verfügung stehen, da sie der Programmcode selber belegt.

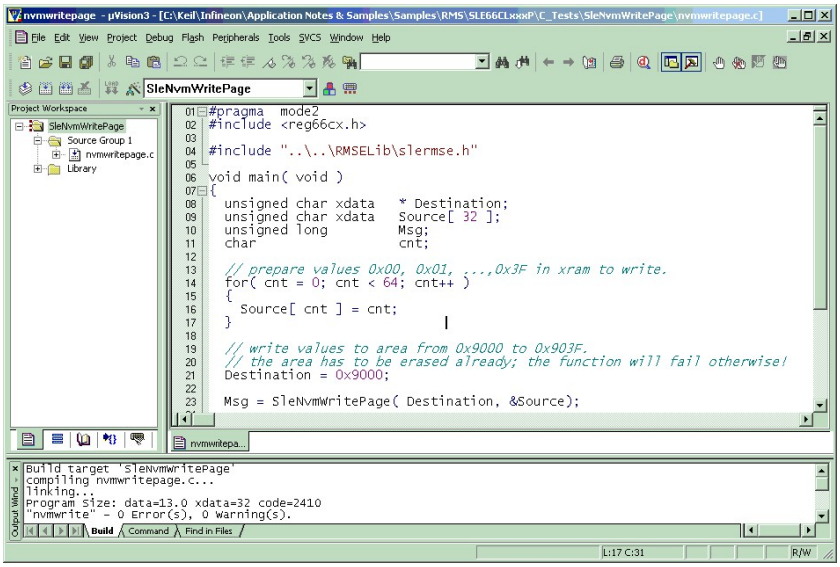

Abbildung 6: Etwicklungsumgebung „μVision“ von KEIL für Native-Karten

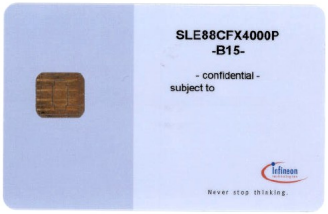

Abbildung 7: infineon SmartMask Evaluierungskarte (Aufdruck teilweise unkenntlich gemacht)

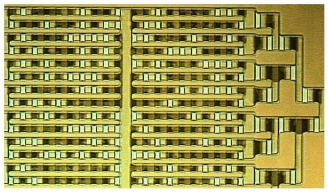

Abbildung 8: geätzte ROM-Speicherzellen unter dem Miskroskop

3.5.2 JavaCards

Java für Chipkarten[11] ist eine sehr neue Technologie, welche die Einstiegshürde für die Entwicklung eigener Software für Chipkarten (sog. Cardlets) gewaltig herabsetzt: Der Entwickler muss sich (kaum) noch mit hardware-spezifischen Besonderheiten aufhalten, denn ein Cardlet läuft auf jeder JavaCard.

Um überhaupt auf derart kleinen Plattformen laufen zu können, enthält das Java für Chipkarten im Vergleich zum „normalen“ Java deutliche Einsparungen. Es fehlen[4, S. 267],[5, S. 10] :

- dynamisches Laden von Klassen, Security Manager,
- Threads (und alles was damit zusammenhängt),
- Klonen, Garbage Collection, Finalization,
- Datentypen: `char` (16 Bit Unicode), `double` (64 Bit), `float` (32 Bit), `long` (64) und `String`, sowie mehrdimensionalen Arrays.

Der Bytecode des compilierten Cardlets wird auf den EEPROM/Flash-Bereich der JavaCard überspielt und (nach Reset und Selektieren das Cardlets) auch von dort ausgeführt, denn die Java Virtal Machine (JVM) befindet sich ebenfalls auf der Karte selbst. Es handelt sich hierbei also *nicht* um einen Java-Native-Compiler (wie es ihn z.B. zur Erzeugung nativen x86er-Codes für PCs gibt), der Bytecode wird tatsächlich *auf* der Karte selber interpretiert und ausgeführt.

Die (im Rahmen dieses Praktikums verwendete) Entwicklungsumgebung für JavaCards war „Aspects“ (Abb. 9): Sie ist vom „Look & Feel“ wie andere IDEs (vom Fensterlayout bis zur Bedienung des integrierten Debuggers). In ihr kann man ebenfalls den gesammten Entwicklungsprozess von der Quelltextbearbeitung bis zum Überspielen des Cardlet-Bytecodes auf eine Karte (über ein PC/SC-kompatibles Terminal) abarbeiten. Ausserdem kann man mit Aspects sog. Skripte ausführen, die automatisches Senden und Auswerten von APDUs zur/von der Karte ermöglichen (in Abb. 9 der Bereich Mitte Rechts), was einem das mühevolle Eintippen langer HexZahlen-Kolonnen erspart.

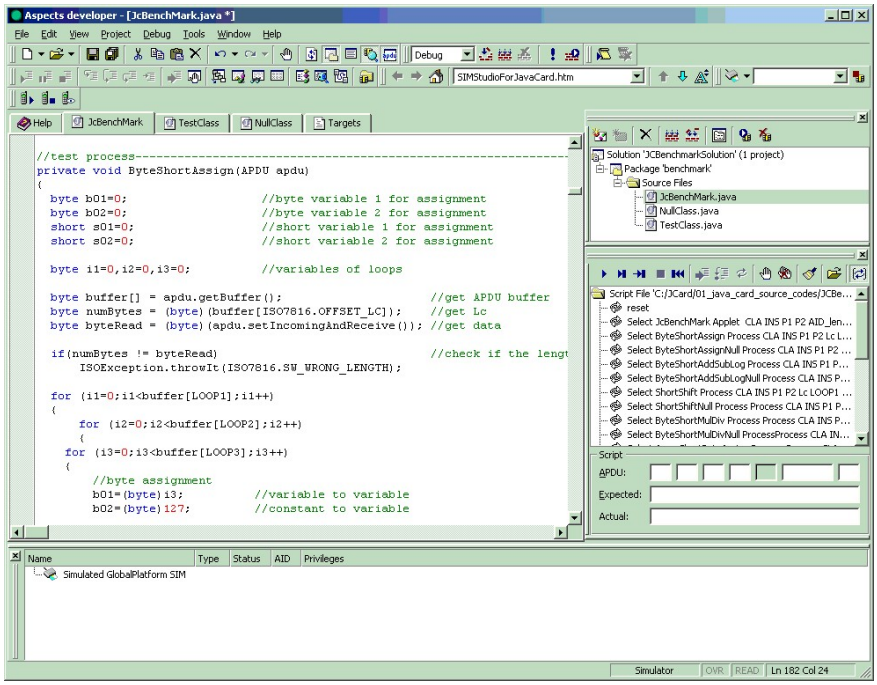

Abbildung 9: Etwicklungsumgebung „Aspects“ für JavaCards

4 Vergleich zwischen Java- und Native-Karten

Fragen der Art „wie viel schneller sind Native-Karten im Vergleich zur JavaCard“ sind so eigentlich nicht objektiv beantwortbar, da sie viel zu Allgemein formuliert sind:
Um dennoch auf diese Sorte Fragen eine sinnvolle Antwort geben zu können, sollten zunachst objektive Vergleiche zwischen Java und Native modelliert werden, deren Ergebnisse dann helfen können, Fragen wie die Eingangs gestellte begründbar beantworten zu können.

4.1 Entwurf und Durchführung des Tests

Ziel eines Vergleichs zwischen Java- und Native-Karten war die Beantwortung der folgenden Fragen:

- Wieviel schneller ist eine Native-Karte gegenüber einer Java-Karte mit gleicher Hardware-Plattform?
- Ist dieses Verhältnis konstant, oder schwankt es je nach Art des Benchmarks?
- Kann man, wenn man die Performance einer Java-Karte kennt auch auf die Performance der entsprechenden Native-Version schliessen?

4.1.1 Laufzeitverhältnisse ermitteln

Vorgabe zur Beantwortung der ersten Frage war es, die beiden Benchmark-Programme (Benchlets) möglichst „ähnlich“ zu programmieren, um einen fairen Vergleich zwischen der Native- und der Java-Version zu gewährleisten:

Es ist klar, daß man zu einem gegebenen Quelltext (konkret: dem Java-Benchmark-Applet „JcBenchmark2003“) nahezu beliebig optimierten Assembler (mit identischer Funktionalität) schreiben kann: Das sollte aber nicht das angestrebte Ziel sein, vielmehr war die Richtline, den vom javac-Compiler erzeugten Bytecode des Applets möglichst nahe angelehnt eine Version in 8051er Assembler zu programmieren, und diese beiden dann gegeneinander antreten zu lassen.

Die Architektur des 8051er Prozessors[3, S. 85 ff.] ist der einer JVM nicht unähnlich, denn er besitzt ebenfalls ein Akku-Register für allgemeine Berechnungen, und es ist deshhalb ständig notwendig, Zwischenergebnisse auf den Stack zu lagern - daher lies sich der (gewissermaßen reverse-engineerte, siehe Abb. 11) Java-Bytecode relativ gut (wenn natürlich auch nicht zeilenweise) in 8051er Assembler umschreiben.

4.1.2 Informationsgewinn über unbekannte Karten

Mit den anderen beiden Fragen war folgende Idee verknüpft:

Gegeben seien 2 Pärchen von Chipkarten; jedes Paar hat identische Architektur und besteht aus einer Java- und einer Native-Version.
Bei einer der 4 Karten hat man keine Möglichkeit die Laufzeit zu messen[4]. Die Idee ist nun, die theoretische Laufzeit der unbekannen Karte über die Laufzeitverhältnisse zwischen Native und Java zu errechnen.
Bevor man auf die unbekannte Karte schliessen kann, sollte jedoch zuerst anhand eines weiteren Kartenpärchens ermittelt werden, ob dieses Java/Native-Laufzeitverhältnis auch konstant ist.

Der Testablauf sollte folgendermaßen aussehen (Abb. 10):

- Laufzeitverhältnis beim ersten Kartenpaar N_1 und J_1 messen: $x_1 = t_{J1}/t_{N1}$.
- Laufzeitverhältnis bei einem weiteren Kartenpaar N_2 und J_2 zur Kontrolle messen: $x2 = t_{J2}/t_{N2}$. Gilt in etwa $x_1 = x_2$?
- Falls dies gilt, dann die Laufzeit t_{J3} der Java-Version der unbekannten Karte K_3 messen. Nun kann die Laufzeit der unbekannten Native-Karte N_3 berechnet werden via:
  $t_{N3} = t_{J3}/x_1$

Java-Seite des Benchmarks Das Java-Benchlet besteht aus diversen Unterroutinen, innerhalb derer 3 verschlachtelte Schleifen durchlaufen werden (die

[4]Die konkrete Situation bei infineon war, daß sich eine der 4 Karten nicht programmieren und somit nicht messen ließ, da keine Programmiertools dazu vorhanden waren. Die Java-Version dieser unbekannten Karte war jedoch programmierbar (denn JavaCards sind auf keine Herstellersoftware angewiesen).

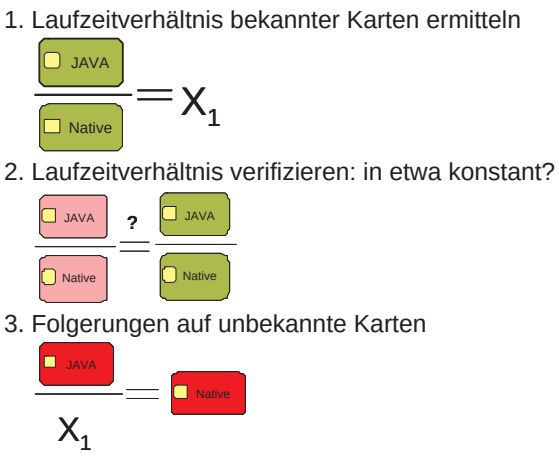

Abbildung 10: Idee zum Schluß auf Laufzeiten unbekannter Karten

Anzahl Durchläufe werden der Karte jeweils in der ersten APDU vom PC mitgeteilt), innerhalb derer jeweils verschiedene mathematische Operationen durchgeführt werden (Assignments, Bit-Shifting).

Listing 1: Byte-Short-Assignment Quelltext

```
// PARAMETER (LOOP*) VOM CCTERMINAL EMPFANGEN
for (i1=0; i1<buffer[LOOP1]; i1++)
{
  for (i2=0; i2<buffer[LOOP2]; i2++)
  {
    for (i3=0; i3<buffer[LOOP3]; i3++)
    {
      b01=(byte)i3;          //variable to variable
      b02=(byte)127;         //constant to variable
      s01=(short)i3;         //variable to variable
      s02=(short)127;        //constant to variable
    }
}
// ANTWORT AN CCTERMINAL SENDEN
```

Zwar enthält das Java-Benchlet noch andere Tests, deren Nachbildung hätte in Native-Assembler allerdings keinen Sinn ergeben, da es sich dort um rein Java-spezifische Vorgänge wie Objekt-Instanziierung gehandelt hätte. Tests von Operationen auf dem NVM fanden nicht statt, da diese zu komplex in Assembler zu progammieren gewesen wären. Auch auf Benchmarking von Kryptofunktionen wurde verzichtet, denn hierbei wäre kaum ein Unterschied zu erwarten gewesen, da derartige Operationen nicht innerhalb der JVM stattfinden, sondern lediglich von ihr an die Kryptoprozessoren weitergegeben würden.

Erster Schritt war somit, das vom javac-Compiler erzeugte Bytecode-File zu untersuchen, die Code-Teile zu finden, die den nachzubildenden Native-

Benchmark entsprechen und diese zu verstehen (insbesondere das Handling verschachtelter Schleifen).

Kurz die Bedeutung der vorkommenden JVM-Befehle in Abb. 11:
`sload X`/`sstore X` laden/speichern Werte vom/zum den Akku zum/vom X Plätze unterhalb des obersten Stackelents; `sconst X` lädt die Konstante X in den Akku, `push X` schiebt die Konstante X auf den Stack; `add` addiert den Stacktop zum Akku, `if_scmplt` führt einen bedingten (Akku $\leq$ Stacktop) Sprung aus, und `baload X` führt einen Arrayzugriff aus.

Man erkennt anhand des Bytecode-Listings in Abb. 11 die allgemeine (und daher schachtelbare) Struktur einer `for (init; compare; increment) {body}`-Schleife in Java wie in Abb. 12 darfestellt:

- Zuerst immer der Initialisierungsblock (INIT)
- Unbedingter Sprung (Maschinenbefehl `goto`) zum Vergleich, ob die Schleifenbedingung noch zutrifft (COMPARE). Dieser Vergleich wird letztendlich immer durch einen elementaren Maschinenbefehl der Art „if greater / equal / less zero go to“ durchgeführt: In Abb. 11 ist es der Befehl `if_scmplt SPRUNGMARKE`.
- Im Erfolgsfall leitet dieser bedingte Sprung in den Schleifenrumpf (BODY),
- an dessen Ende die Erhöhung der Laufvariable (INCREMENT) steht.
- Danach folgt wieder der COMPARE-Block.

Native-Seite des Benchmarks Zum gegebenen Java-Bytecode sollte nun ein 8051er-Assemblerprogramm erstellt werden, daß an den Code aus Abb. 11 möglichst gut angenähert ist:

Die Programmierung der Schleifen erfolgte hier insbesondere so, daß sie schachtelbar sind: Somit durften also die Schleifenvariablen (`i1, i2, i3`) nicht in den Hilfsregistern des 8051er gehalten werden (denn bei einer gewissen Schachtelungstiefe wären davon keine mehr übrig), sondern mußten auf dem Stack zwischengesichert werden.

Wichtig bei dieser Nachbildung der for-Schleifen-Blöcke ist (Abb. 13), daß jeweils auch genau *ein* bedingter und *ein* unbedingter Sprung pro Schleifendurchlauf enthalten sind (bedeutsam für die Performanz der Schleife), sowie daß die Struktur selber schachtelbar ist.

4.2 Ergebnisse des Tests

Leider konnte die Idee vom Informationsgewinn durch Laufzeitverrhältnisse nicht angewandt werden: Es zeigte sich, daß das Verhältnis - stark abhängig vom verwendeten Kartentyp - stark streut:
Ein Beispiel:

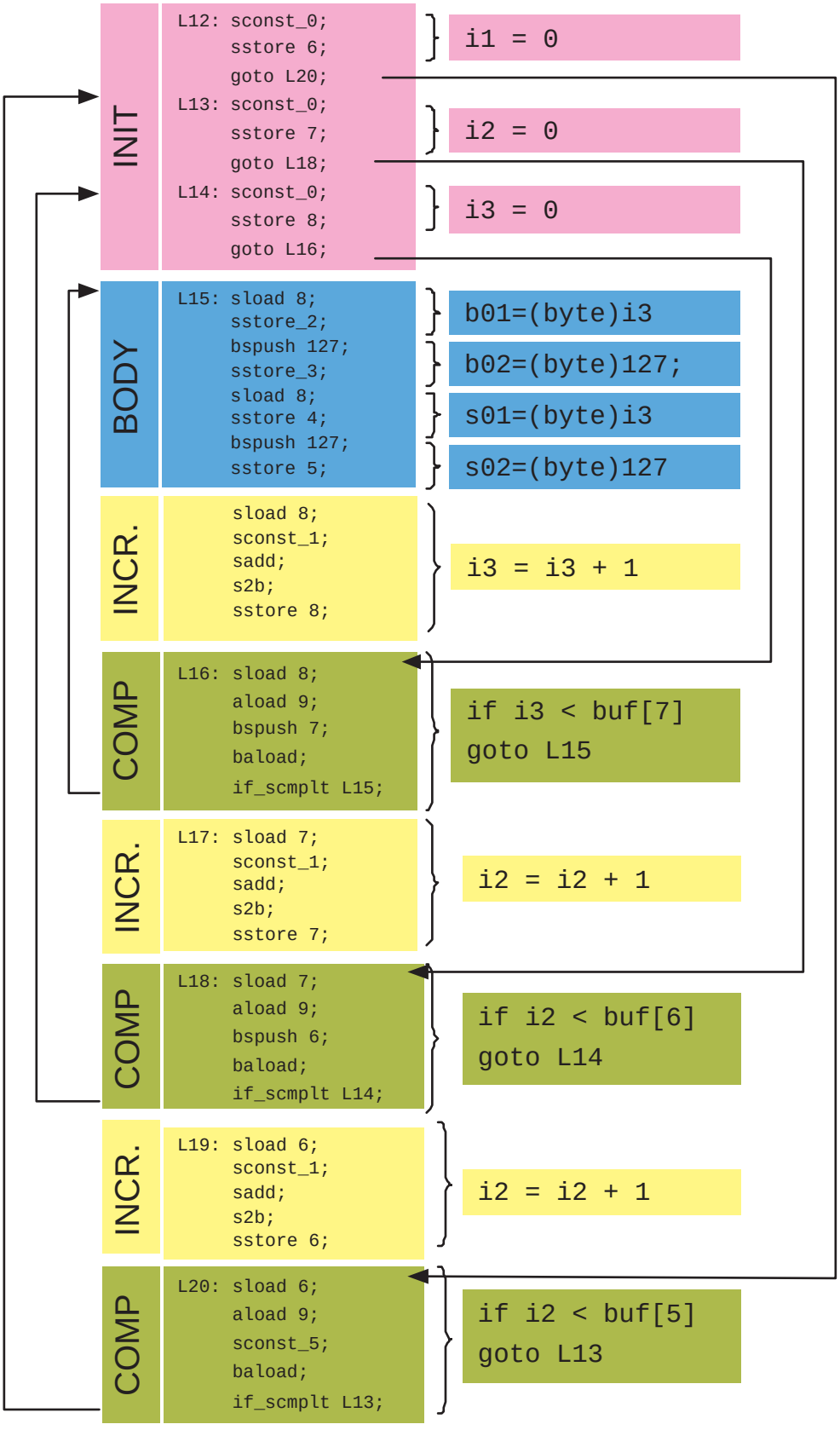

Abbildung 11: 3 verschachtelte for-Schleifen im Java-Bytecode

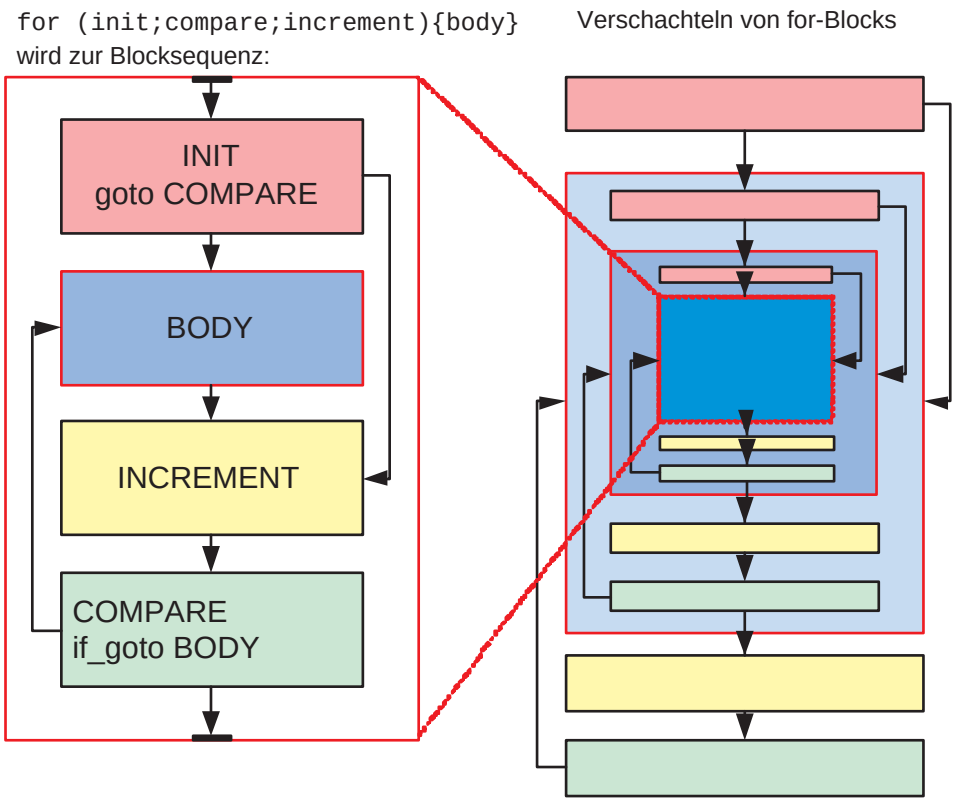

Abbildung 12: Blockdarstellung einer for-Loop im java-Bytecode

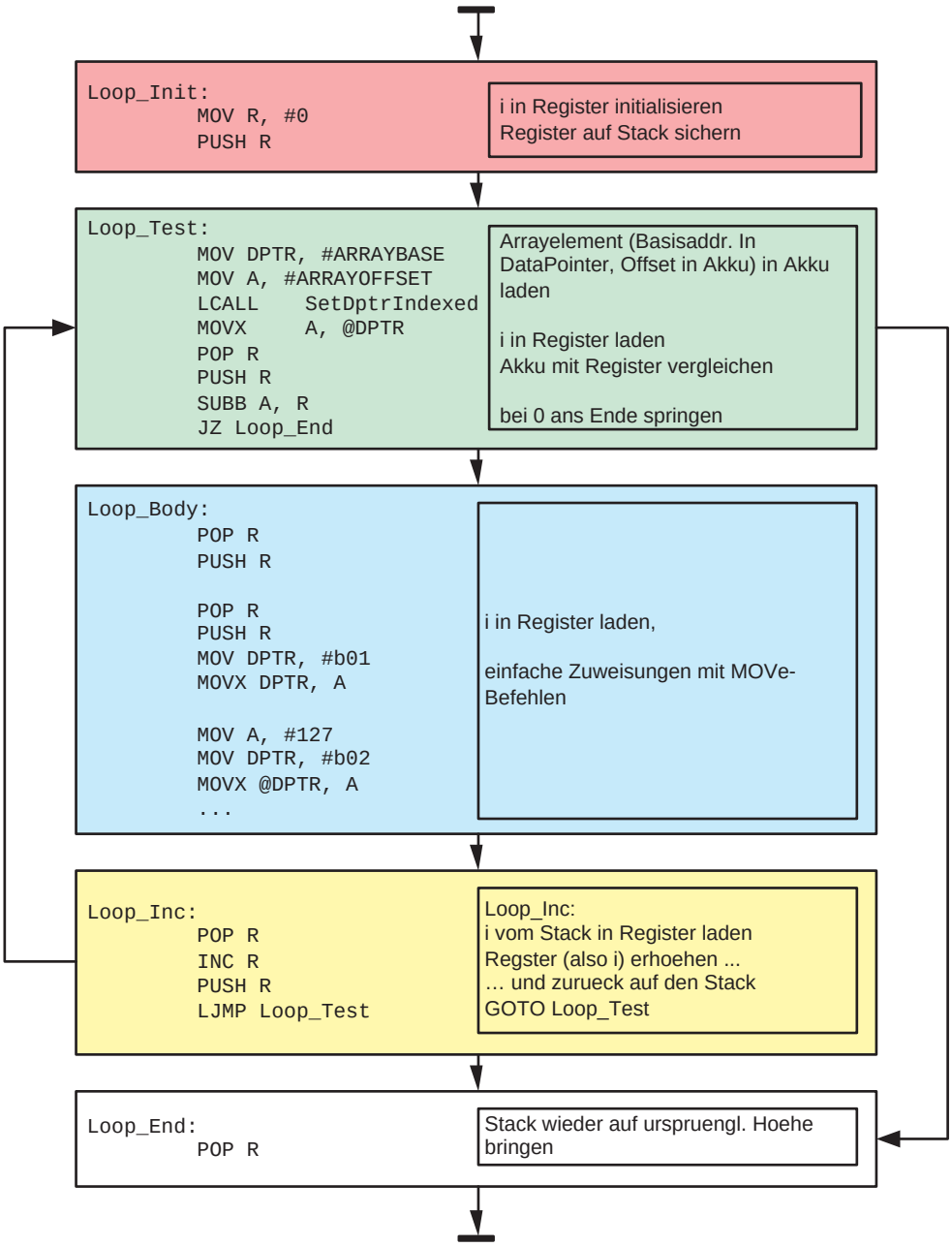

Abbildung 13: Blockdarstellung einer for-Loop im 8051er-Assembler

- Infineon SLE66_card_1 (8 Bit Core):
  Native: 116,23 μs.
  Java: 2603,64 μs.
  $\Rightarrow$ Die Java-Version ist etwa um Faktor 22 langsamer.

- Infineon SLE66_card_2 (8 Bit Core):
  Native: 109,23 μs.
  Java: 1546,21 μs.
  $\Rightarrow$ Die Java-Version ist etwa um Faktor 14 langsamer.

Obwohl also die Native-Laufzeiten in etwa vergleichbar waren, war bei Karte 2 Java schneller: Grund ist vermutlich, daß die (jüngere) Karte 2 eine neuere JVM-Software einsetzt, somit ist also eine Vorraussetzung zur Anwendung der Schluß-Idee nicht mehr gegeben.

Die Ergebisse zusammengefasst:

- Das Verhältnis streut recht weit (Faktor 14-22= und nicht wie erhofft konstant), möglicherweise wegen verschiedener JVMs auf den Karten.

- Weitere mögliche Verzerrungen beim Vergleich: Bei neuerer Hardware existieren echte Maschinenbefehle, die JVM-Befehle direkt unterstützen können.

- Über unbekannte Karten können deshalb nur vage Schätzungen gemacht werden.

- Das T=1-Protokoll bremst die JavaCard zusaätzlich, da es bei längeren Berechnungen regelmäßige Rückmeldungen der Karte verlangt, wodurch sie ihre eigentliche Arbeit unterbrechen muß.

5 Benchmark-Zeitmessung bei Chipkarten

Das einfache Prinzip eines Laufzeit-Benchmarks einer Chipkarte lautet:
„Sende das Kommando zum Starten des Benmarks und stoppe die Zeit bis zur Antwort der Karte nach dem Benchmark“.
Kompliziert wird es allerdings bei der Implementierung einer solchen Stoppuhr: Abhängig davon, wo man die Stoppuhr zwischen dem IO-Pin der Karte und der Software im PC, die den Benchmark anstößt, ansetzt, hat man mit Problemen wie den Aufwand für einen „Lauschangriff“ oder Verlust an Messgenauigkeit zu kämpfen.
Ein paar Ansatzmöglichkeiten der Stoppuhr, von „oben nach unten“ wandernd, sollen kurz erläutert und bewertet werden:

5.1 Zeitmessung durch Software im PC mit dessen Uhr

Entwicklungsumgebungen wie z.B. Aspects geben in ihrem Kommunikations-Logfenster zu jeder APDU zwar jeweils die Zeitspanne in Millisekunden an, die seit der letzten APDU vergangen ist: Dieser Wert eignet sich allerdings nur als Schätzwert und variiert bei Wiederholungen zu stark für Benchmarking-Anforderungen[5, S. 57 ff.]:

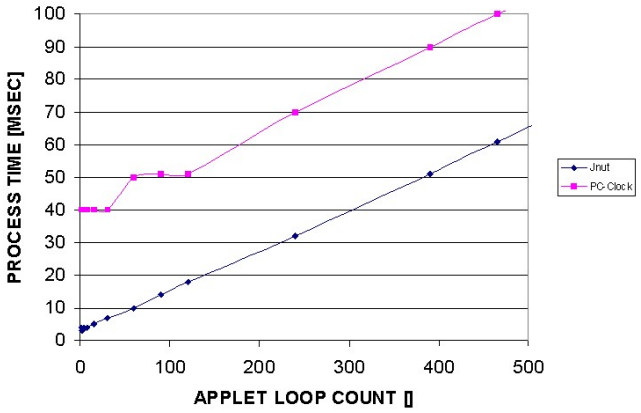

Abbildung 14: Ungenauigkeiten beim PC als Stoppuhr

Grund dafür ist, daß ein PC, und insbesondere eine Anwendungssoftware (wie Aspects) nicht für Echtzeit-Anwendungen wie Zeitmessung im Mikrosekundenbereich geeignet ist, wie eine Messung von infineon (Abb. 14) verdeutlicht (Abweichungen von einer Geraden bei rosa Linie). Typische PC-Betriebssysteme garantieren ihren Anwendungsprogrammen eben keine harte Echtzeit (Stichwort Scheduling), deshalb ist es ihnen nicht möglich den exakten Zeitpunkt, wann die APDUs, die Start und Stopp des Benchmarks anzeigen, über die Schnittstelle zum Terminal ein- und ausgehen exakt zu bestimmen.

Der Aufwand für einen tief im Betriebssystem verankerten Treiber, der die Kommunikation mit dem Terminal regelt und quasi „nebenbei“ noch alles mit Zeitstempeln versehen mitloggt, wäre sehr hoch, insbesondere, da dieser Treiber

ja für darauf aufsetzende Anwendungen transparent seien müsste. Ausserdem kann der Treiber selber durch andere Hardware-Interrupts unterbrochen werden und dadurch nicht exakt mitstoppen.

Zur völligen Disqualifikation einer Stoppuhr innerhalb eines Treibers führt allerdings die Tatsache, daß er immer noch oberhalb des Karten-Terminals ansetzt:
Ein Byte, daß von der Karte gesendet wird, muß ersteinmal durch das Terminal hindurch, bevor es überhaupt zur PC-Schnittstelle gelangt: Und innerhalb des Terminals sitzt im Allgemeinen wieder ein Mikroprozessor, der es einliest, eventuell interpretiert, und dann erst an den PC weiterleitet: Dies ist insbesondere bei PC/SC-Lesern der Fall, oder allgemein allen Kartenlesern, die nicht über RS-232 sondern z.B. USB an den PC angeschlossen sind, da hierbei eine Übertragungsprotokoll-Umsetzung erfolgen muss, und dies kostet Rechenzeit. Auch darf man hierbei nicht argumentieren, daß durch die Umsetzung zwar Verzögerungen entstünden, diese aber bei Bildung von Zeitdifferenzen herausfielen, denn es gibt keine Garantie, daß diese Verzögerung immer konstant sein muss (Jitter-Effekt).

5.2 Zeitmessung an einem Transparentleser

Die einzigen Terminals, die ohne Delay und Jitter arbeiten können, sind sogenannte Transparent-Leser: Diese bestehen im einfachsten Fall nur aus einem elektrischen Spannungswandler zwischen den TTL-Pegeln des IO-Pins und den ±12Volt-Pegeln an der seriellen Schnittstelle des PCs (folglich funktionieren sie auch nur mit asynchronen Chipkarten) und einem Resetauslöser über separate Steuerleitungen der RS-232-Schnittstelle (RTS, DTR).

Das konkret zur Verfügung gestandene Terminal (in Abb. 16 unten zu sehen), daß eben auch im Transparent-Modus arbeiten kann, musste allerdings erst auf die Eigenschaft der Verzögerungsfreiheit hin untersucht werden: Ein Blick unter den Deckel des Gerätes zeigte zahlreiche größere Prozessoren im Inneren, es bestand also die Gefahr, daß die Kommunikation beim möglichen Weg durch diese Bausteine (trotz Transparent-Modus) gebremst werden könnte: Somit musste also erst getestet werden, ob dem so ist ode nicht:

Dazu wurden 2 Messungen mit einem 2-Kanal-Speicheroszilloskop durchgeführt: Bei beiden Messungen wurde der erste Kanal mit dem IO-Pin der Chipkarte verbunden, und der andere Kanal mit dem TxD- (1. Messung) bzw. dem RxD-Anschluss (2. Messung) am PC-Anschluss des Terminals verbunden (siehe in Abb. 20):

Bei der 1. Messung wurde ein Kartenreset ausgelöst, was eine ATR-Antwort der Karte zur Folge hatte, und damit Spannungsflanken am IO-Pin und dem TxD-Pin: Trigger[5] für das Oszilloskop war eine Flanke am IO-Pin (diese Flanke kann garnicht nach einer Flanke am TxD-Pin kommen): Ergebnis war, daß beide Flanken gleichzeitig auftraten (zu sehen in Abb.15: Die Flanken bei beiden Kanälen (gelb, weiss) am Bildschirm des Oszilloskops stehen vertikal untereinander). Bei der 2. Messung wurde ein Kommando an die Karte geschickt, Trig-

[5] Auslöser f'ur das Speichern der Messwerte im Oszilloskop

ger war diesmal der RxD-Pin (aus analogem Grund zur 1. Messung). Ergebnis waren auch hier gleichzeitige Spannungsflanken. Somit war also nachgewiesen, daß dieses Terminal Verzögerungs- (und damit insbesondere Jitter-) frei Daten zwischen Karte und PC austauscht.

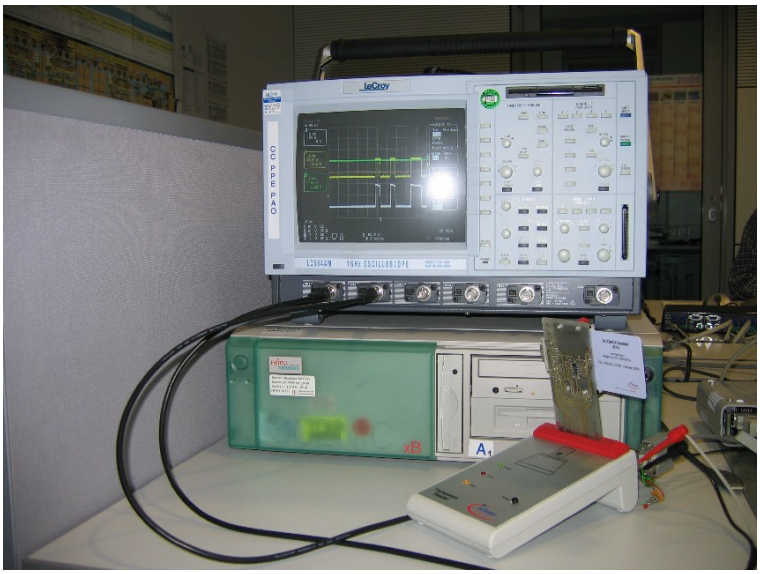

Abbildung 15: Jitter-Messung mit 2-Kanal-Oszilloskop

Problem am Transparent-Leser ist allerdings (naben weiteren mit der JNut-Box dabei), daß er nur „Raw“-Zugriff auf das Terminal und die Karte bietet, Anwendungen wie Aspects jedoch auf die höhere PC/SC-Schnittstelle aufsetzen.

5.3 Zeitmessung am IO-Pin der Karte

Quasi „ganz unten“ setzt die Messung am IO-Pin der Karte an: Hierzu wurde bereits von infineon eine kleine „IO-Sniffer-Platine“ gebaut (in Abb. 16 das, was ins Terminal eingesteckt ist): Diese hat auf der einen Seite die Kontaktflächen einer Chipkarte an genormter Stelle (und kann und somit in jeden Leseschlitz eingesteckt werden) und auf der anderen Seite eine Standard-Kontaktiereinrichtung. Alle Kontakte sind entsprechend miteinander verbunden, jedoch am IO-Pin sitzt paralell ein Pegelwandler, dessen Ausgang mit dem TxD-Pin einer üblichen RS-232-Schnittstelle (der „Snifferport“) verbunden ist.

Verbindet man den Snifferport mit einem PC und startet ein (am besten HexDump-fähiges) Terminalrogramm, so kann man die Kommunikation zwischen Karte und Terminal - analog zum Lauscher in der Telefonleitung - mithören.

5.3.1 Die JNut-Box

Mit der IO-Sniffer-Platine alleine ist es allerdings nicht getan (ebenso wie mit einem Jitter-freien Transparentleser): Lauscht man mit einem PC-Programm

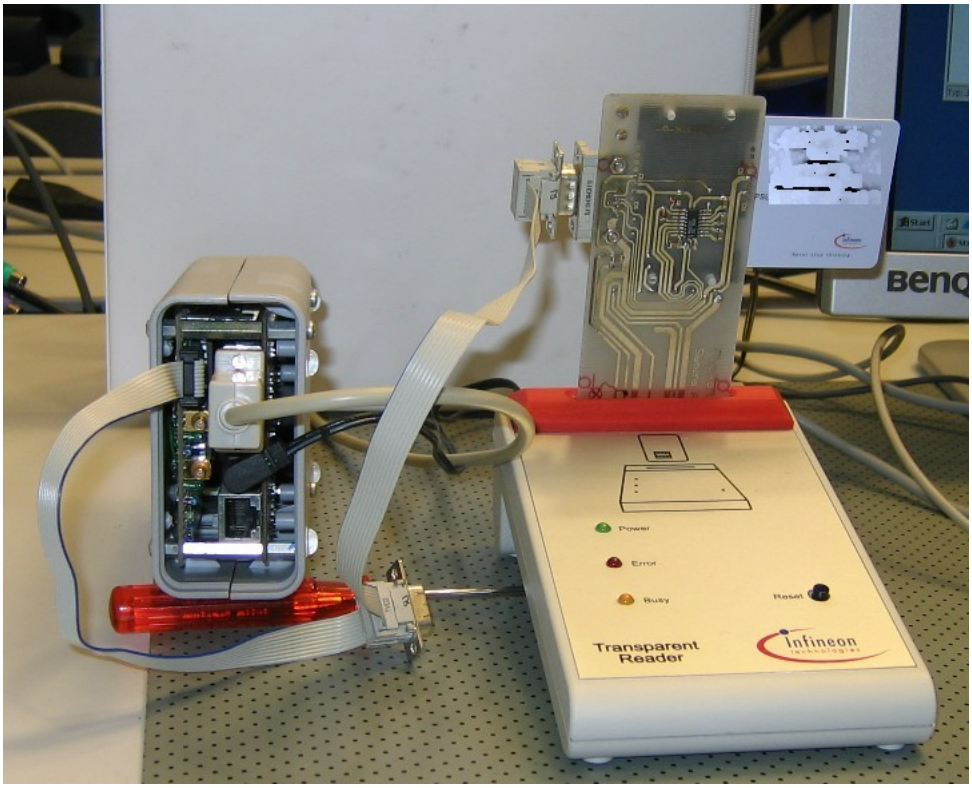

Abbildung 16: Messaufbau mit JNut-Box (links), Katenterminal (unten), IO-Abgriff(oben), Karte (rechts)

am Snifferport, so steht man wieder vor dem Problem, daß dieses nicht Echtzeitfähig ist.

Deshalb wurde von infineon eine extra Hardware entwickelt und bereitgestellt, die sogenannte „JNut-Box“: Diese ist ein Microcontroller mit unter anderem 2 seriellen Schnitstellen (eine zum Lauschen am Snifferport, die andere um Kommandos zu empfangen und Stoppzeiten zurückzuliefern). Hauptaspekt bei der JNut-Box ist, daß sie echtzeitfähig ist, was konkret bedeutet, daß sie sich beim Lauschen am Snifferport von keinerlei andernen Störquellen (wie Interrupts, weiteren Programmen, ...) unterbrechen lässt, und dadurch größtmögliche Präzision bei den Stoppzeiten erreichen kann.

Der schematische Setup des Messplatzes aus Abb. 16 ist in Abb. 17 dargestellt: Die JNut-Box lauscht über die Sniffer-Platine am IO-Kanal, über PCs wird der Benchark gestartet und die Ergebnisse anschliessend erfasst (müssen nicht notwendigerweise 2 verschiedene PCs sein).

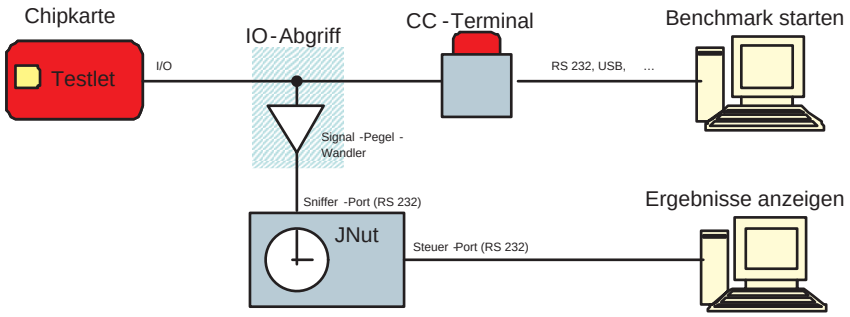

Abbildung 17: Schema des Messplatzes mit der JNut-Box

Die JNut-Box Hardware (Abb. 16 links) besteht aus einem ATMEL ATMega-Microcontroller auf einer Platine aus dem sog. „ethernut“-Projekt[6][10].

Programmieren lässt sich die JNut-Box (neben dem bei Microcontrollern oft unausweichlichen Assembler) auch in Standard-C: Für ATMEL-Prozessoren existiert eine Portierung des GNU-C-Compilers gcc samt Standard-Library, was Entwicklung, Erweiterung und Verstandnis der Firmware deutlich erleichtert.

Die Firmware der JNutBox besteht grundsätzlich aus Kommando-Interpreter, String-Matcher und Stoppuhr: Über die serielle Kommando-Schnittstelle übergibt man der JNut-Box ein sogenanntes Start- und Stopp-Pattern in Form eines Hex-Strings: Schaltet man daraufhin die JNut-Box „scharf“, so beginnt sie an ihrer zweiten seriellen Schnittstelle zu lauschen: Liest sie das Start-Pattern ein („match“), so schaltet sie ihre interne, mikrosekunden-genaue Stoppuhr ein, und stoppt sie erst wieder, wenn sie das übergebene Stopp-Pattern eingelesen hat („match“). Den gemessenen Zeitwert sendet sie dann über die Kommando-Schnittstelle an den PC.

Zusätzlich bietet die JNut-Box noch die Möglichkeit, zu den Zeitpunkten beim Matchen des Start- und Stopp-Patterns Triggerflanken an ein Oszilloskop

[6]Ethernut ist ein Projekt aus Open-Source Hard- und Software für einen Embedded-Microcontroller mit Ethernet-Schnittstelle und TCP/IP-Stack.

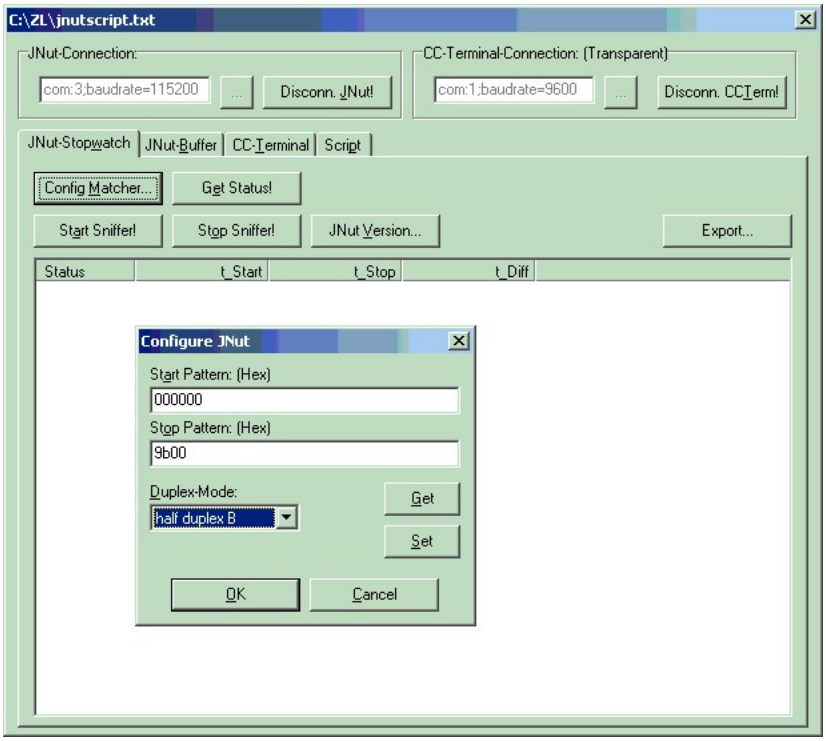

Abbildung 18: Screenshot JNutStop-Programm: Im Vordergrund der Dialog zum Start/Stopp-Pattern setzen

zu senden, um damit den Stromverbrauch der Karten während des Benchmarks messen zu können (goldene Steckbuchsen der JNutBox in Abb. 16).

Zusammenfassend: Man hat mit diesem Tool das universellste Werkzeug zur Zeitmessung, da es unabhängig vom Terminal (Transparent, PC/SC, RS-232, USB, ...), verwendeter PC-Software und Chipkartentyp funktioniert. Nachteilig ist, daß es umständlich zu konfigurieren ist (kryptische Kommandos über ein Terminalprogramm), und daß man, resultierend daraus, daß es auf unterster Schicht aufsetzt, Low-Level-Kartenprotokoll-besonderheiten beachten muss, wenn man das Start- und Stopp-Pattern auswählt:
Arbeiten Karten im T=1-Protokoll, ist zu beachten, daß die APDUs im Allgemeinen nicht so auf der IO-Leitung Übertragen werden, wie sie auf der Applikationsschicht sichtbar sind, sondern zerlegt, unterbrochen oder anderweiitig verändert werden können. Ebenfalls ist möglich, daß sich die auf dem IO-Pin die Baudrate ändern kann (wenn die Karte dazu fähig ist).

5.4 Die JNutStop-Software

Die erwähnte kryptische manuelle Konfiguration der JNut-Box über ein Terminalprogramm legte den Gedanken nahe, ein Programm zur komfortablen Steuerung der JNutBox zu entwickeln (Abb. 18):

Anforderungen an diese fensterorientierte Windows-Software waren (geord-

net nach Priorität).

- Setzen des Start- und Stopp-Patterns, Ein- und Ausschalten der Stoppuhr (über Dialogfenster).
- Ausgabe der Stoppzeiten, idealerweise in einem einfachen CSV[7]-Format, da es sich gut zur Weiterverarbeitung in Tabellenkalkulationen eignet.
- Anzeige der von der JNut-Box mitgelesenen Daten (Sniffer Buffer Content).
- Ansteuerung eines Chipkarten-Terminals (entweder PC/SC oder Transparent, die Wahl fiel auf Transparent).
- Möglichkeit, sowohl Terminal als auch JNut-Box durch Skripte zu steuern

Die Entwicklung fand mit dem Tool „C++Builder“ von Borland statt, da sich damit die Erstellung graphischer Oberflächen ähnlich einfach wie z.B. mit VisualBasic gestaltet, er in der Abteilung bei infineon verbreitet ist (somit also später auch von Anderen weiterentwickelt werden könnte), und die Programmiersprache C++ ist.

Die Kommunikation mit der seriellen Schnittstelle unter Windows ist -ähnlich wie unter UNIX- stark File-orientiert: Damit gestaltete sich sowohl die Kommunikation mit der JNut-Box sowie dem Chipkartenterminal sehr einfach, nur für den Reset-Befehl an das Terminal war der Zugriff auf die RTS- DTR-und Statusleitungen nötig).

Die Fähigkeiten der Skriptsprache sollten alles das umfassen und automatisieren, was man innerhalb des Programms auch über die Fenster erreichen kann:

- Öffen, Schliessen und Setzen der Schnittstellenparameter zur JNut-Box und zum Kartenterminal
- Senden zum, Empfangen von und Reset des Kartenterminals
- Starten, Stoppen, Zurücksetzen sowie setzten des Start- und Stopp-Patterns der JNut-Box
- Ausgabe der gestoppten Zeitwerte in eine Datei (bzw. Textbox oder Zwischenablage)

Weitere Forderung war es, (möglichst) kompatibel zum sogenannten SunScript-Format zu bleiben (das z.B. auch von Aspects verwendet wird). Um diese Kompatibilität zu erfüllen wudern die JNutStop-spezifischen Skriptanweisungen allesammt als SunScript-Kommentare (eingeleitet durch die Zeichen `//`) gestaltet.

Ein Beispiel für ein Skript:

- Die Karte soll resetet werden,
- die Patterns gesetzt,

[7] comma-separated values

- danach soll der Benchmark auf der Karte mit der Hex-Befehlssequenz `0x01 0x02 0x03 0x04` gestartet werden
- nach dem Benchmark sendet die Karte `0x0a 0x0b 0x0c` zurück.
- die Stoppzeit soll ausgegeben werden

Ein JNut-Skript dafür sieht dann so aus:

Listing 2: Demo JNutStop-Script

```
//JNUT open com3;baudrate=1152000
//JNUT set_start 0x01 0x02 0x03 0x04
//JNUT set_stop  0x0a 0x0b 0x0c
//JNUT start
//CCTERM open com1;baudrate=9600
//CCTERM reset
//SLEEP  1000
//CCTERM send 0x01 0x02 0x03 0x04
//CCTERM recv
//JNUT fetch
//LOG Laufzeit: \$match
```

Die Ausgabe ist dann z.B. „`Laufzeit: 500000`“ im Ausgabefenster, falls der Benchmark eine halbe Sekunde gedauert hat (Die Variable `$match` wird durch die Stoppzeit ersetzt).

Mit Hilfe dieser Skriptsprache ist es einfach, große Mengen an Benchmarks durchlaufen zu lassen (grösere Skripte selber lassen sich wiederum leicht z.B. mit Perl erzeugen), und deren Ergebnisse in Form von Semikolon-getrennter-Listen nach z.B. Excel zu importieren und graphisch aufzubereiten.

6 Verbesserungungen bei der Zeitmessung

Das Benchmarking unter Verwendung der JNut-Testumgebung zeigte, daß noch zahlreiche Dinge erweitert und verbessert werden könnten, sowohl auf Seite der JNut-Hardware, Firmware, und der JNut-Stop-Anwendungssoftware:

- Hardware: Der verwendete IO-Port-Sniffer bietet keine Möglichkeit, beim Sniffing zu unterscheiden, ob das übertragene Byte von der Karte an das Terminal gesendet wurde oder umgekehrt (Abb. 19):

  Dies erschwert eine Protokollanalyse, da eben die Kommunikation beider in nur einem Sniffing-Buffer der JNut-Box landet, und man im Allgemeinen dann nicht mehr feststellen kann, wer Sender und wer Empfänger eines Bytes war. Zur Ermittlung der Datenflußrichtung wurden verschiedene Lösungsmöglichkeiten diskutiert, die kurz vorgestellt werden sollen:

  - Die JNut-Box lauscht zwischen Terminal und PC (Abb. 20): Hier ist die Leitung bereits 2 mal unidirektional, somit können beide

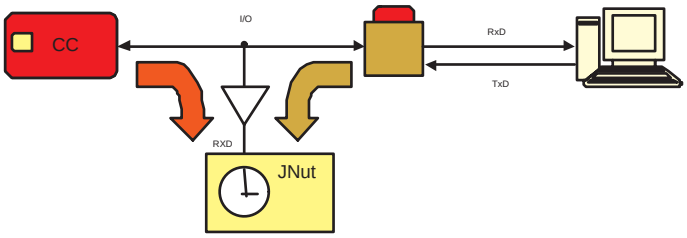

Abbildung 19: Problem beim Sniffen auf 1 bidirektionalen Leitung: Wer ist der Sender?

Kanäle getrennt voneinander aufgezeichnet werden. Nachteil an dieser Lösung ist allerdings, daß die JNut-Box keine 2 einzelnen Kanäle gleichzeitig aufzeichnen kann, und man auf PC-Seite auf RS-232-Leser festgelegt ist (die ausserdem noch Jitterfrei arbeiten müssen, sowie einen PC/SC-Treiber mitbringen sollten).

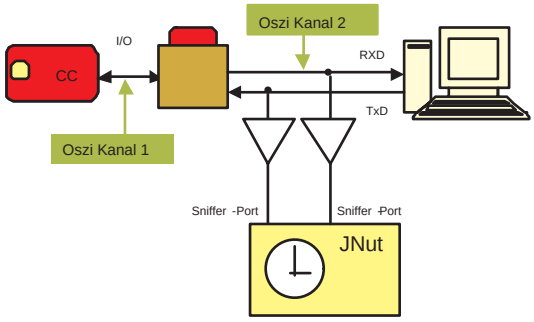

Abbildung 20: Sniffen auf 2 unidirektionalen Leitungen

– Ein „man in the middle“-Angriff auf die IO-Leitung (Abb. 21): Dazu muss die Leitung entweder aufgetrennt werden, oder auf elektrische Art festgestellt werden, wer Sender und Empfänger ist: Eine derartige Schaltung wurde skizziert:

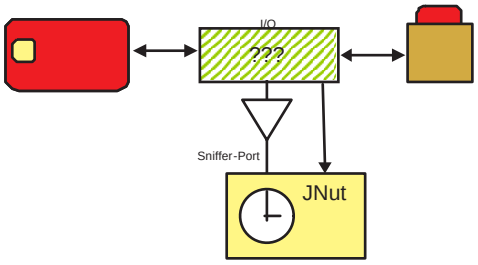

Abbildung 21: Für beide Seiten transparentes Aufplitten der IO-Leitung

Prinzipielle Idee wäre gewesen, Zwei „2zu1“-Datenwandler[8] mitein-

[8]eine derartiges Splitting der Datenpfade müssen auch Transparentleser zur Anbinung an die 2fach-unidirektionale serielle Schnittstelle des PCs vornehmen

ander zu verbinden, so daß in der Mitte der Datenverkehr gesplittet vorliegt, und die Richtung ermittelt werden kann (Abb. 22).

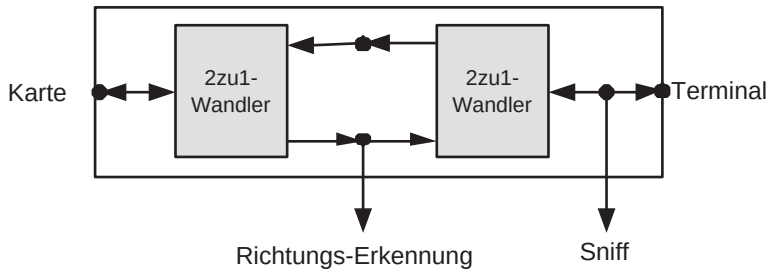

Abbildung 22: Erkennen der Datenflußrichtung durch 2 Splitter

Der praktische Aufbau einer solchen Box wäre jedoch elektrotechnisch äusserst anspruchsvoll gewesen (Optokoppler, Echo-Unterdrückung, Tiefpässe, ...), deshalb (und aus Zeitgründen) blieb es leider nur bei der Theorie.

- Firmware: Nach dem Reset einer Karte kann diese encodiert in ihrer ATR dem Terminal mitteilen, daß sie eine schnellere Baudrate fahren kann: Kommt das Terminal diesem Angebot nach, so muss auch der Sniffer der JNut-Box auf diese schnellere Baudrate mit-umgeschalten werden, da sonst falsch mitgelesen wird. Bislang existiert nur eine manuelle Baudraten-Umschaltung am Snifferport (einfache JNut-Box-Firmware-Erweiterung), eine automatische ATR-Auswertung und entsprechende Baudumschaltung wäre für solche Karten sehr wünschenswert.

- Anwendungssoftware:

  - Eine Implementierung des T=1-Protokolls für den Transparent-Leser, samt Feautures wie Baudratenwechsel nach dem ATR würde die Kommunikation mit Karten dieses Protokolls deutlich erleichtern.
  - Alternativ (und möglicherweise einfacher in der Implementierung) wäre die Verwendung der PC/SC-Schnittstelle zur Terminalansteuerung, anstatt eines Transparent-Kartenterminals: Damit entfiele die Implementierung des sehr komplexen T=1-Protokolls (da dies unterhalb der PC/SC-Schnittstelle erst umgesetzt wird) und man wäre flexibler bei der Auswahl der zu verwendenden Kartenterminals.
  - Da die Hardware der JNut-Box auch die Möglichkeit gibt, via Trigger ein Ozilloskop den Stromverbrauch messen zu lassen, wäre es praktisch, diese gemessenen Werte vom Oszilloskop abzuholen, und mit in die Ausgabedatei schreiben zu lassen (also in etwa analog zur Skript-Variable `$match` eine Variable `$current`, die den Stromverbrauch während des Benchmarks hält).

Neben diesen Ideen wurden noch andere diskutiert, z.B. warum man die JNut Box nicht als reine Logging-Box verwendet, die jedes Byte lediglich mit einem exakten Zeitstempel versieht und man dieses Log dann später innerhalb eines PCs analysiert: Es wäre komfortabler, derartige Analysesoftware für PCs zu schreiben zumal dieser Teil der Arbeit ja dann nichtmehr zeitkritisch wäre. Dies ist aber leider nicht möglich, da das Weiterreichen der gesniffetn Daten + Zeitstempel (genauer: dessen Senden über den UART der Kommando-Schnittstelle) die Präzision der internen Uhr gefährdet (UART Sende-Interrupts).

7 Fazit

All diese Ideen zur Befriedigung der Wünsche an einen besseren JNut-Messplatz sind natürlich noch nicht ausgereift oder vollstandig, jedoch einige davon sicherlich mit vertretbarem Aufwand realisierbar. Dies bedarf allerdings einer genaueren Prüfung dieser Vorschläge, für die im Rahmen dieses Praktikums leider kein Platz mehr war, oder elektrotechnisch sehr anspruchsvoll waren (wenn auch durchaus für einen Informatiker, der keine Angst vor Lötkolben und Messgerät hat, zu schaffen).

Abschließend bleibt zum Thema JavaCard zu bemerken, daß diese Technologie (insbesondere, wenn sie in Verbindung mit den kontaktlosen RFIDs tritt) sicherlich bald einen starken Aufschwung erleben wird: Innerhalb des Praktikums wurde selbst erlebt, wie hoch die Einstiegshürde für Native-Programmierung ist (mehrere Wochen bis zum ersten „Hello World“ von der Karte), und im Vergleich dazu die zum JavaCard-Programmieren (einige Stunden bis zum selben Ergebnis).

Literatur

[1] Volpe, Francesco P./ Volpe, Safinaz: Chipkarten - Grundlagen, Technik, Anwendungen, Verlag Heinz Heise Hannover, 1996.

[2] Guthery, Scott B./ Jurgensen, Timothy M.: Smart Card Developer's Kit-Macmillian Technical Publishing, Indianapolis, Indiana, 1998

[3] Köhn, K.P./ Schultes, R: 8051 -Prozessoren - Einführung, Applikationen, Programmierung, Franzi's Verlag GmbH Poing, 1994

[4] Rankl, Wolfgang / Effing, Wolfgang:
Handbuch der Chipkarten - Aufbau, Funktionsweise, Einsatz von Smart Cards
Carl Hanser Verlag München Wien, 1999

[5] Erdmann, Monika:
Diplomarbeit: Benchmarking von Java Cards,
LMU München 2004,
`http://www.nm.ifi.lmu.de/pub/Diplomarbeiten/erdm04`

[6] Wikipedia: Artikel „Chipkarte“:
`http://de.wikipedia.org/wiki/Chipkarte`

[7] Wikipedia: Artikel „RS232“:
`http://de.wikipedia.org/wiki/RS232`

[8] Wikipedia: Artikel „ATR“:
`http://de.wikipedia.org/wiki/ATR`

[9] PC/SC Working Group: PC/SC-Standard Overview,
`http://www.pcscworkgroup.com/specifications/overview.php`

[10] Ethernut: Projekt Homepage,
`http://www.ethernut.de/`

[11] Sun Microsystems: Java Card Technology Home Page,
`http://java.sun.com/products/javacard/`