Entwurf und Implementierung einer Applikationsverwaltung für den SHAP-Bytecodeprozessor

Großer Beleg

an der

FAKULTÄT INFORMATIK
DER TECHNISCHEN UNIVERSITÄT DRESDEN

Eingereicht am:
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR VLSI-ENTWURFSSYSTEME, DIAGNOSTIK UND ARCHITEKTUR

Von: Peter Ebert
Matrikel-Nr.: 3129797
Marianne-Bruns-Str. 10, 01219, Dresden

Bearbeitungszeitraum: vom 15.06.2010
bis 14.12.2010

Betr. HSL: Prof. Dr.-Ing. habil. Rainer G. Spallek

Betreuer: Dipl.-Inf. Thomas B. Preußer

Dresden, 14. Dezember 2010

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt sowie die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Peter Ebert, Dresden, 14. Dezember 2010

I. Inhaltsverzeichnis

II. Tabellenverzeichnis

III. Abbildungsverzeichnis

Sämtliche Abbildungen wurden vom Autor dieser Arbeit mit dem Programm Microsoft Office Visio 2007 erstellt.

IV. Abkürzungsverzeichnis

0x	Zahl in hexadezimaler Schreibweise
ack	Acknowledgement (Bestätigung)
API	Application Programming Interface
App	Applikation
ASIC	application-specific integrated circuit
Bd	Baud, Einheit der Symbolrate
bzw.	beziehungsweise
CVS	Concurrent Versions System
EIA-232	Electronic Industries Alliance 232
FPGA	field-programmable gate array
GUI	graphical user interface
ID	Identifikationsnummer
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JVM CP	Konstanten Pool der JVM
PC	Personal Computer
RS-232	Recommended Standard 232 (siehe EIA-232)
RXTX	(Receiver/Transmitter)
SHAP	Secure Hardware Agent Platform
TU	Technische Universität
UART	universal asynchronous receiver/transmitter
VLSI	very-large-scale integration

1. Einleitung

1.1 SHAP

SHAP ist ein Projekt des Lehrstuhls VLSI-Entwurfssysteme, Diagnostik und Architektur der Technischen Universität Dresden. Es wurde 2006 ins Leben gerufen und bedeutet *Secure Hardware Agent Platform*. Darin stecken drei wesentliche Aspekte: Ziel des Projektes ist die Entwicklung einer Agenten-Plattform. Auf ihr sollen komplexe Aufgaben verteilt gelöst werden. Dazu wird die Programmiersprache Java genutzt. Sie bietet neben hoher Mobilität und Plattformunabhängigkeit auch wichtige Sicherheitsmerkmale. Um echtzeitfähig und mobil zu sein, wird SHAP direkt in Hardware implementiert und ersetzt somit die originale *Java Virtual Machine* (JVM)–Software, die gewöhnlich auf einem Betriebssystem aufsetzt. *Anwendungsspezifische Integrierte Schaltungen* (ASICs) können diese Hardware stellen. Aus Forschungsgründen werden aktuell programmierbare Schaltkreise verwendet.

1.2 Ausgangssituation

Wird das SHAP-System auf einem *Field Programmable Gate Array* (FPGA) programmiert, passiert erst einmal nichts. Es wartet auf ein auszuführendes Java-Programm. Dieses muss allerdings erst zu einer .shap-Datei umgewandelt werden. Zuständig dafür ist der sogenannte *ShapLinker*. Er geht von der mit Namen anzugebenden Startklasse aus und sammelt alle dazu benötigten weiteren Programm- und Runtime-Klassen. Zusätzlich wird die Klasse `Bootloader` als erste zu ladende Klasse hinzugefügt, deren Aufgabe es ist, die Initialisierung auf dem SHAP durchzuführen. Diesem Pool werden in der .shap-Datei drei Blöcke nachgestellt: die Class Patches, String Table und String Patches. Neben der .shap- kann auch eine nützliche .log-Datei erstellt werden. Über ein Terminalprogramm und eine serielle Schnittstelle kann mit dem als SHAP programmierten FPGA kommuniziert werden. Nun wird die erstellte .shap-Datei gesendet und stößt so den Initialisierungsvorgang an. Sobald dieser beendet ist, startet das eigentliche Java-Programm seine Arbeit. Ausgaben werden über die Standardausgabe, im Normalfall ist das der *Universal Asynchronous Receiver Transmitter* (UART), zurückgeschickt und vom Terminalprogramm sichtbar gemacht. Soll ein neues Java-Programm starten, muss es wie zuvor in eine neue .shap-Datei umgewandelt und initialisiert werden. Eine parallele Abarbeitung ist deshalb nur insofern möglich, als dass man seine Programme in ein größeres, umschließendes Programm packt und gleichzeitig hoch lädt. Späteres Hinzufügen einer Applikation ist also nicht möglich, ohne den vorigen Zustand auf dem SHAP inklusive Programm zu beenden.

1.3 Zielsetzung

Ziel dieses Beleges ist die Entwicklung eines Systemdienstes auf SHAP, der es zulässt, im Nachhinein weitere Applikationen hoch zu laden, zu starten, zu verwalten und auch zu beenden. Er deckt somit den gesamten Lebenszyklus der Applikationen ab. Er soll von einem externen Terminal komfortabel gesteuert werden. Die Ausgabe der Programme soll gemultiplext über die serielle UART-Schnittstelle ausgegeben werden, sodass eine bessere Fernsteuerung für SHAP erreicht wird.

1.4 Aufbau

Im Rahmen dieses Beleges läuft die aktuelle SHAP-Version, vom lehrstuhlinternen CVS-Server, vom 15. Oktober 2010 auf einem *Spartan-3* FPGA der Firma *Xilinx*. Die gesamte Kommunikation wird über die serielle UART-Schnittstelle EIA-232 bewältigt. Der Terminalrechner besitzt einen 64 Bit *Intel Core 2 Duo* Prozessor und wurde mit *Microsoft Windows 7 Professional* 64 Bit ausgestattet. *Java Runtime Environment* (JRE) und *Java Development Kit* (JDK) sind in der Version 1.6.0_22 installiert und als Entwicklungsumgebung wird *Eclipse* 3.6.1 verwendet. Zur Programmierung des *Spartan-3* wurde *Xilinx's ISE Design Suite* 12.3 benutzt.

In Kapitel 2 wird beschrieben, wie eine Kommunikation auf Byte-Ebene zwischen Threads aufgebaut und erweitert und schließlich dieser Prototyp auf die SHAP-Terminal-Umgebung portiert wird, sodass eine fehlerfreie Paketvermittlung steht. Kapitel 3 gibt einen Überblick über das erste Hochladen, danach über das Nachladen eines Programmes. Es schließt mit dem Verfahren, welches es ermöglicht, die neue Applikation auszuführen. Nachdem verschiedene Programme nun laufen, wird in Kapitel 4 die Implementierung aller Verwaltungsaufgaben erläutert. Kapitel 5 gibt abschließend einen Ausblick auf Optimierungs- und Weiterentwicklungsmöglichkeiten sowie ein Fazit der entstandenen Software.

2. Konzept & Implementierung einer Kommunikation zwischen Terminal und Systemdienst

Der erste Schritt ist das Einlesen in das SHAP-Projekt[1], speziell in die Veröffentlichung zur Arbeitsweise des `Bootloaders`.[2] Er ist der Kernangriffspunkt im System, den es zu verstehen und zu ändern gilt. Zwei wesentliche Teile erweitern im Rahmen dieses Beleges das SHAP-Projekt - zunächst der Systemdienst. Er muss als Basisprogramm auf der SHAP-Maschine laufen. Die eigentlichen Programme sollen über ihn gestartet und verwaltet werden. Der zweite Teil ist ein eigens angepasstes Terminalprogramm. Es soll auf dem Terminalrechner laufen, einen dafür konzipierten Kommunikationskanal öffnen und darüber Informationen über Ausgaben, Eingaben oder die zu startenden Programme, aber auch Steuerbefehle, übertragen.

2.1 Übertragungsprotokoll

Bevor nun Daten verschickt und empfangen werden können, muss aufbauend auf der seriellen Schnittstelle ein Protokoll definiert werden. Es regelt die Art der Informationsübertragung zwischen Systemdienst und Terminal, sodass beide Seiten die Bitströme richtig interpretieren können. Zur Definition eines Protokolls ist nun folgendes wesentlich: Welche Verwaltungsaufgaben soll der Systemdienst verrichten? Welche verschiedenen Kommandos sollen dazu auf dem Terminal angeboten werden? Welche Kommandos resultieren in Kommunikation mit der SHAP-Seite? Welche Informationen und Daten müssen einem Befehl beigefügt werden? Und wie kann eine eindeutige Interpretation gewährleistet werden?

Die Aufgabenstellung gibt Auskunft über die Verwaltungspflichten des Systemdienstes. Namentlich werden Hochladen, Kommunikation, Überwachung und Steuerung einer Applikation genannt. Das heißt, die Eingaben müssen dem gewünschten Programm zugeführt und die Ausgabe entweder zum Terminal-PC übertragen, zwischengespeichert oder ignoriert werden. Überwachung meint das Auslesen von Statusinformationen. Steuerung dagegen ist ein recht allgemeiner Begriff. Es wurde zunächst nur die Terminierung einer Applikation und der Soft-Reset des Systems integriert. Für weitere zukünftige Aufgaben wird im Protokoll der nötige Freiraum gelassen. Soll heißen, das führende Opcode-Byte kann noch weitere Paketarten codieren.

Somit ergeben sich der Reihe nach acht Befehle: `start`, `input`, `listen`, `unlisten`, `mute`, `status`, `kill` und `reset`. Um diese Funktionen überhaupt nutzen zu können, ist ein weiterer Hilfsbefehl nötig: `list` gibt alle laufenden Pro-

[1] Martin Zabel, Thomas B. Preußer, Peter Reichel, Rainer G. Spallek: *SHAP - Secure Hardware Agent Platform*. TUDpress, 2007, S. 119 - 126.

[2] Thomas B. Preußer & Rainer G. Spallek: *Java-Programmed Bootloading in Spite of Load-Time Code Patching on a Minimal Embedded Bytecode Processor*, CSREA Press, 2008, S. 260 - 264.

gramme mit Identifizierungsnummer zurück, um diese eindeutig auswählen zu können. Darüber hinaus wurde der Befehl `clear` eingefügt. Er soll den Zwischenspeicher einer Programmausgabe leeren, um eine bessere Orientierung zu erreichen. Zur Steigerung der Benutzerfreundlichkeit haben auch die bekannten Kommandos `help` und `exit` hier ihre Funktion. `Help` beschreibt dem Nutzer kurz die möglichen Kommandos samt eventuellen Parametern. `Exit` schließt den Terminalklienten. Das gleiche passiert bei der Eingabe von `quit`. Es ergibt sich die Summe von zwölf Terminal-Kommandos. Abgesehen von `help` und `exit` verlangen alle Kommandos eine Kommunikation mit dem Systemdienst und resultieren daher in verschiedenen Übertragungspaketen. Zusätzlich ist ein weiteres Paket nötig, das DATA genannt wird. Es dient vor allem dem Systemdienst, welcher damit seine angeforderte Information zum Terminal zurücksendet. DATA und INPUT haben dem Konzept nach den gleichen Paketaufbau. Zusätzlich aber ist ihr Auftreten disjunkt. Das heißt, INPUT wird nur auf der Hinrichtung, DATA nur auf der Rückrichtung verwendet. Dem Protokoll brauch deshalb nur eine solche Paketform hinzugefügt werden. Die unterschiedliche Interpretation auf den Plattformen unterscheidet die weitere Vorgehensweise. Wegen seiner allgemeineren Bedeutung wird als Bezeichnung im Folgenden nur noch DATA verwendet. Die Sequenzdiagramme aller kommunikationsstiftenden Befehle sehen folgendermaßen aus:

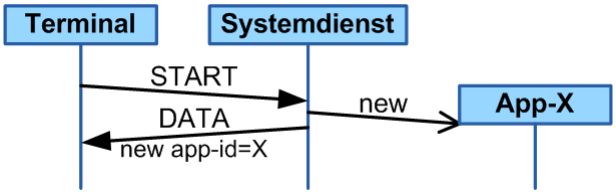

Abb. 1: Sequenzdiagramm für den `start`-Befehl

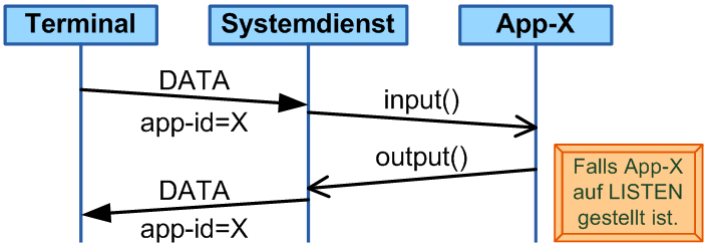

Abb. 2: Sequenzdiagramm für den `input`-Befehl und den Output

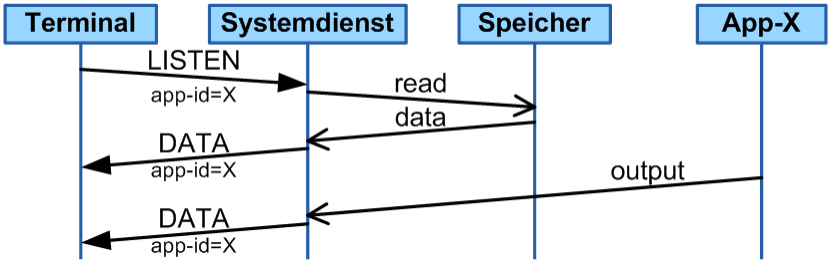

Abb. 3: Sequenzdiagramm für den `listen`-Befehl

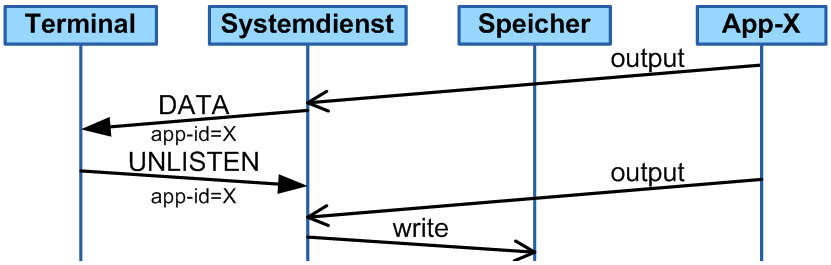

Abb. 4: Sequenzdiagramm für den `unlisten`-Befehl

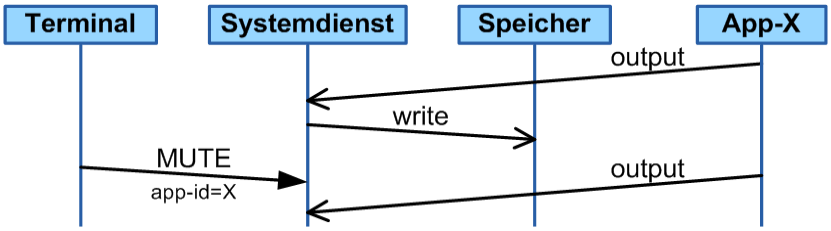

Abb. 5: Sequenzdiagramm für den `mute`-Befehl

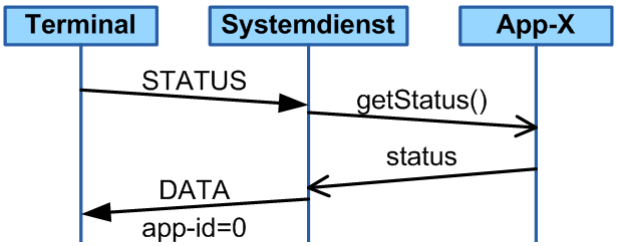

Abb. 6: Sequenzdiagramm für den `status`-Befehl

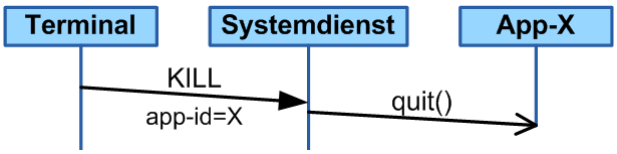

Abb. 7: Sequenzdiagramm für den `kill`-Befehl

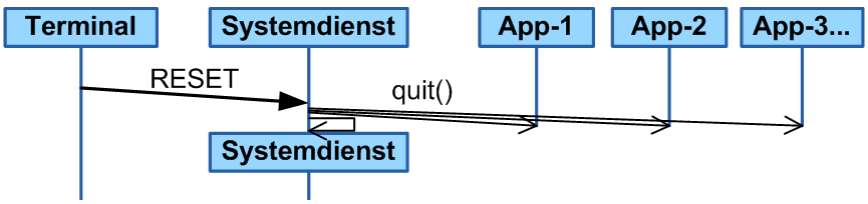

Abb. 8: Sequenzdiagramm für den `reset`-Befehl

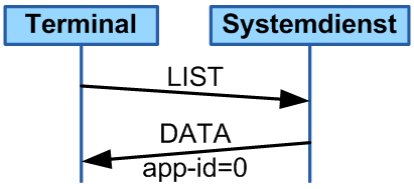

Abb. 9: Sequenzdiagramm für den `list`-Befehl

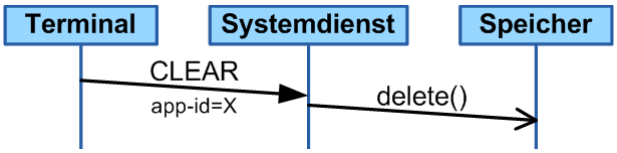

Abb. 10: Sequenzdiagramm für den `clear`-Befehl

Alle Paketarten des Protokolls sind in nachstehender Tabelle dargestellt. Byte 0 enthält den Befehlscode. 246 andere Befehle können somit nachgerüstet werden. Welche Information den Codes anhängt, ist ebenfalls zu erkennen.

Tabelle 1: Protokolldefinition zur Kommunikation zwischen Terminal und Systemdienst

CMD	byte0	byte1	byte2	byte3	byte4 etc…	Antwort			
LIST	0 1					0 3	0 0	*length*	*app-Liste*
START	0 2	*length*			*payload*	0 3	0 0	*length*	*neue app-id*
DATA	0 3	*id*	*length*		*payload*				
LISTEN	0 4	*id*				0 3	0xid	*length*	*data*
UNLISTEN	0 5	*id*							
MUTE	0 6	*id*							
CLEAR	0 7	*id*							
STATUS	0 8	*id*				0 3	0 0	*length*	*status infos*
KILL	0 9	*id*							
RESET	0 A								

(Quelle: Autor)

Die Notwendigkeit der eindeutigen Applikations-`id` und des Nutzdatenteils `payload` ist trivial. Da die Größe der Nutzdaten erst zur Laufzeit feststeht, ist auch die Paketgröße nicht vorher bestimmbar. Hier sind zwei Ansätze denkbar.

1. Bestimmung der Größe während des Packprozesses und Beifügen dieser Information in den Header an definierter Stelle.
2. Markierung des Anfangs oder des Endes eines Paketes.

Um auf einem Bitstrom eine eindeutige Marke zu setzen, muss allerdings gewährleistet werden, dass diese Marke in den restlichen Paketbits nicht vorkommt. Das zu realisieren ist umständlich und hat dabei keinen Vorteil gegenüber der ersten Methode. Dies wäre nur der Fall, wenn angefangen werden soll zu senden, noch bevor das Datenvolumen bekannt ist. Dies ist hier nicht nötig.

Die UART lässt sich in einigen Parametern konfigurieren. SHAP legt folgende Einstellungen fest:

- Baudrate: 115200 Bd
- Datenbits: 8
- Parität: keine
- Stopbits: 1
- Flusskontrolle: Xon/Xoff

Es werden also Bytes gesendet. Um kongruent auf den UART-Paketen aufzusetzen, ist die Größe eines Pakets in unserem Protokoll ein ganzzahliges Vielfaches eines Bytes. Die Paketlänge gibt dieses Vielfache an. Mit zwei `length`-Bytes im Header ergibt sich eine maximale Nutzdatengröße von $2^{16}\ Byte = 64\ KByte$. Es wird erwartet, dass das Starten eines Programmes und damit das Hochladen der nötigen Daten die größten Pakete generiert. Die typische Größe einer .shap-Datei bewegt sich zwischen 2^{15} und 2^{17}Bytes. Die Übertragung der Datei ist in einem einzigen Paket also nicht möglich. Durch Zerlegung und separater Vermittlung der Teile kann dies allerdings erreicht werden. Dieses Verfahren erscheint zunächst als Umweg, birgt aber Vorteile. Im späteren Verlauf wird darauf näher eingegangen.

Sicherungsverfahren zur korrekten Reihenfolge, Fehlererkennung oder gar Fehlerkorrektur werden nicht implementiert. Bei seriellen Schnittstellen können sich Daten nicht überholen. Zur Fehlererkennung wurde im EIA-232-Standard ein optionales Paritätsbit definiert. Das gesamte SHAP-Projekt verzichtet allerdings darauf. Erfahrungsgemäß ist dies aber keine übliche Störquelle. Eine Fehlerkorrektur ist somit ebenfalls nicht vorhanden.

2.2 Prototyp

Die Entwicklung des Prototyps hatte drei Ziele. Erstens: die Schaffung je eines Grundgerüstes für das Terminal und den Systemdienst. Das meint die Auslotung der jeweiligen Grundfunktionalitäten. Zweitens: das Erstellen einer funktionierenden und fehlertolerierenden Frontend-Benutzereingabe. Drittens und am wesentlichsten ist: die Installation einer Kommunikation zwischen beiden Teilen, die ausschließlich über eine Byte-sendende und eine Byte-empfangende Funktion geschieht.

Der Prototyp hat noch kein FPGA vorausgesetzt und basiert auf der Java-API. Er besteht aus einem `SystemDaemon`-Thread, der eingehende Bytes auswertet und einem `ShapTerminal`-Thread, der die User-Eingabe auswertet. Weiterhin gibt es noch einen `TerminalReader`-Thread, der vom Systemdienst eingehende Bytes auswertet und einen Dummy-Applikations-Thread, der im Verlauf gestartet, verwaltet und terminiert werden kann.

Vom Prototyp sind rückblickend vor allem die Benutzereingabe und die Paketerstellung in sehr ähnlicher Form übernommen worden. Für die Eingabe läuft in der `run()`-Methode ein `BufferedReader` auf dem `InputStream System.in`. Sie liest zeilenweise, prüft und startet bei Erkennung eines gültigen Kommandos die korrespondierende Methode. In ihr findet eine nochmalige, kommandoabhängige Prüfung statt. Zur Generierung der Pakete wird die Methode `wrap()` verwendet. Sie hat den Opcode, die Applikations-ID und die Nutzdaten als Argumente. Daraus wird die Headerlänge ermittelt, der Header zusammengebaut und die Nutzdaten angehängt.

2.3 Portierung des Prototyps auf die Entwicklungsumgebung

Um den Prototypen auf SHAP zu starten, bedarf es einiger Schritte. Zunächst muss zur Programmierung des FPGAs das Tool XILINX ISE Design Suite installiert werden. Dabei ist eine Registrierung notwendig. Auf Basis der im Lehrstuhl-CVS bereitliegenden Dateien wird nun ein Bit-File erstellt. Diese Datei entspricht der SHAP-Hardware. Mit ihr wird das FPGA programmiert. Anschließend wartet das System auf vom UART kommenden Input, interpretiert diesen als .shap-Datei und initialisiert damit SHAP. Diese .shap-Datei wird vom ShapLinker erstellt und beinhaltet im Prototyp-Fall den Systemdienst und die Dummy-Thread-Klasse `APP`. Der Linker funktioniert dabei folgendermaßen: er ist ein Apache-Ant[3]-Projekt und besteht aus drei Teilen. Teil 1 erstellt die `shap-rt.jar` aus allen API-Klassen. Teil 2 generiert aus den Quelldateien die ausführbare `ShapLinker.jar`, die in Teil Drei gestartet wird. In der Build-Datei `build.xml` des Ant-Projektes muss dazu ein neuer Eintrag für den Systemdienst vorgenommen werden, um dem Linker die nötigen Informationen zu übergeben, was er bei Eingabe von „`ant SystemDaemon.shap`“ zu tun hat:

```
<target name="SystemDaemon.shap" depends="compile,ShapLinker">
    <ShapLinker main="SystemDaemon">
    </ShapLinker>
</target>
```

Während der Verarbeitung werden alle nötigen User-Programm-Klassen mit der Runtime zur „`SystemDaemon.shap`“-Datei verrechnet. Als Versuchsterminal wurde *Terra Term Pro* verwendet. Es interpretiert jedes ankommende Byte als Zeichen. Damit kann verifiziert werden, ob der Systemdienst läuft, solange das ShapTerminal noch keine funktionsfähige, paketorientierte Kommunikation erlaubt.

2.3.1 Byte-Kommunikation mit Hilfe von RXTX

Haben die Threads des Prototyps sich noch direkt in Java mit Sende- und Empfangsmethoden unterhalten, so ist das nicht mehr möglich, wenn der ShapLinker aus dem Systemdienst eine .shap-Datei erzeugt und diese auf dem SHAP startet. Der einzige Kommunikationskanal ist die serielle Schnittstelle. Das bestehende SHAP nutzt diese als Standardausgabe, wenn ein auf dem SHAP laufendes Programm den `OutputStream System.out` oder den `InputStream System.in` verwenden will. Dabei werden `native`[4] Methoden genutzt. Das sind in Java nutzbare, aber nicht in Java geschriebene Funktionen, da sie plattformabhängig sind. Ein Ziel von Java ist seine Portierbarkeit. Deshalb sind diese Methoden bzw. eine generelle, einheitliche Kommunikation über die serielle Schnittstelle in der Java-API nicht vorgesehen. *Sun* hat dafür eigens die Comm-API geschaffen. Windowssysteme werden allerdings nicht

[3] Apache Ant ist ein in Java geschriebenes Programm zur automatisierten Erstellung von ausführbaren Programmen aus Quellcode. Weiterführende Informationen sind verfügbar unter: http://ant.apache.org

[4] In SHAP ist eine native Methode ein Befehl im Bytecode der in Mikrocode, welcher Befehle zur Steuerung des Prozessors anbietet, implementiert wird.

mehr unterstützt. Soweit die Recherchen ergeben haben, ist die am weitesten verbreitete und zudem einzige Alternative *RXTX*. Dabei ist RX die Abkürzung für Receiver und TX die für Transmitter. Zur Installation sind zwei Komponenten nötig. Zum einen die API-Bibliothek `RXTXcomm.jar` und zum anderen die Datei `rxtxSerial.dll`.

Nach Einbindung von RXTX wurde basierend auf dem Beispiel-Code einer „Event based two way communication“[5] die Klasse `TerminalSerialComm` erstellt. Sie unterscheidet sich vom Original im Wesentlichen davon, dass die Threads `SerialWriter` und `SerialReader` verschwunden sind. Ihre Funktionen sind nun in die Methoden `sendBytes()` und `readBytes()` migriert. Der `main`-Thread `ShapTerminal` übernimmt das Schreiben, der Thread `TerminalReader` das Lesen. Abbildung 11 zeigt die Kommunikationsstruktur.

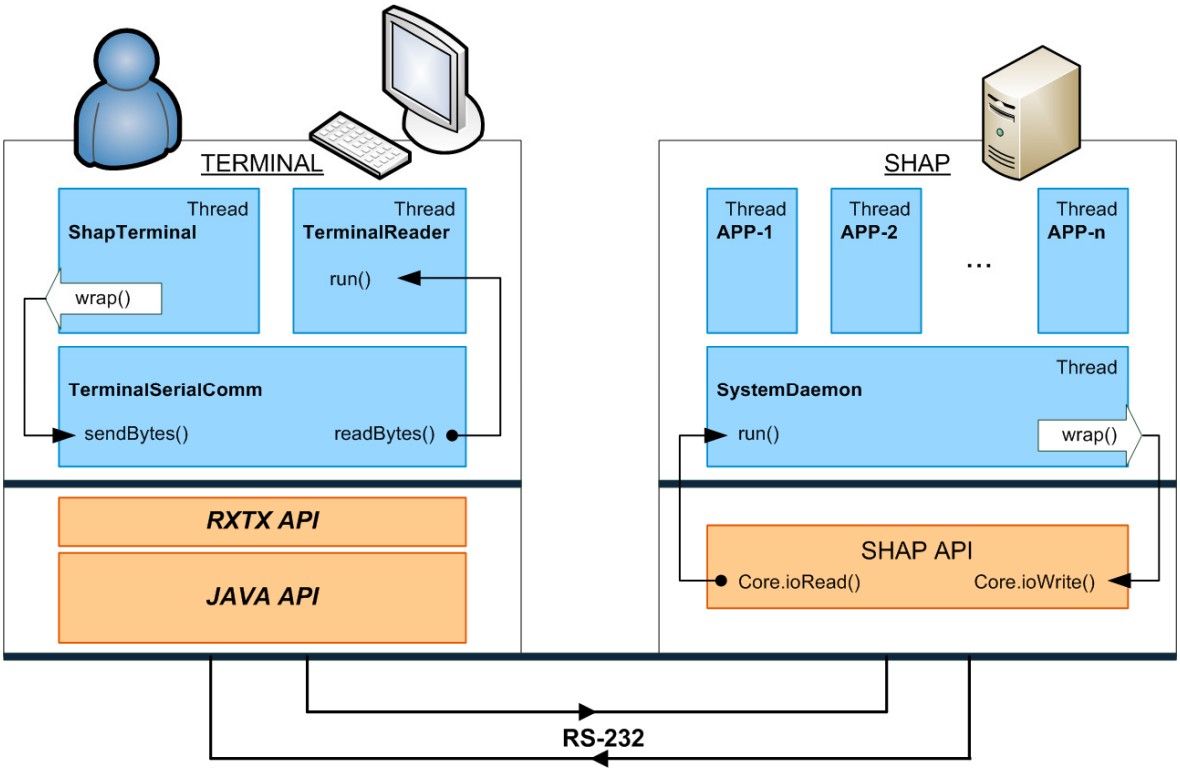

Abb. 11: Übersicht der Kommunikation über die serielle Schnittstelle

Dem Terminal liegen die Java-API und die Erweiterung RXTX zu Grunde und es nutzt deren Stream-Methoden `read()` und `write()`. Applikationen auf SHAP dürfen einzig auf der SHAP-Runtime-API aufbauen. Diese bietet in der Klasse `java.lang.Core` die nativen Methoden `ioRead()` und `ioWrite()` an, die je nach Parameter unterschiedliche Hardware ansprechen und `int`-Werte lesen bzw. schreiben. In unserem Fall steht die Adresse 0x80000000 für die serielle Schnittstelle RS-232. Für Debug- und Statuszwecke ist ebenfalls die Adresse 0x81000000 interessant. Sie steht

[5] Verfügbar unter: http://rxtx.qbang.org/wiki/index.php/Event_based_two_way_Communication

für die 7-Segment-Anzeige des FPGA. Mit dem entsprechenden Code[6] lassen sich Zeichen, und bei Verwendung der Adresse 0x81000001 auch direkt Zahlen, in hexadezimaler Schreibweise darstellen.

In der ersten Version, die RXTX implementierte, wurde auf dem Codebeispiel „Two way communication with the serial port“[7] aufgebaut. Da unser Thread `Terminal-Reader` ständig versucht vom Stream zu lesen, hat er seinen CPU-Kern zu 100 Prozent ausgelastet. Durch die Verwendung eines `SerialPortEventListener`s, der auf das Ereignis `DATA_AVAILABLE` anspringt, konnte der Wert selbst bei hohem Programmoutput durchschnittlich auf unter ein Prozent gesenkt werden. Besonders wichtig ist dabei die Ausführung der Methode `notifyOnDataAvailable(true)` auf dem benutzten `SerialPort`-Objekt und die Synchronisierung der `readbytes()`-Methode über dieses Objekt.

Bevor nach dem Starten des Terminals auf dem Terminalrechner die Verbindung mit SHAP steht, wird zuerst die statische Methode `getAvailableSerialPorts`() der Klasse `TerminalSerialComm` ausgeführt, die alle freien seriellen Schnittstellen auflistet. Der Benutzer soll jetzt den gewünschten Port auswählen. Erst dann wird eine neue Instanz der `TerminalSerialComm` erzeugt, dessen Parameter aus der Datei `config.txt`[8] gelesen werden, insofern beim Lesen kein Fehler auftritt. Im Fehlerfall werden die Default-Werte geladen, wie sie in der `help.txt` beschrieben sind. Mit der nichtstatischen Methode `TerminalSerialComm.connect()` wird schließlich die eigentliche Verbindung hergestellt. Wenn der Verbindungsaufbau erfolgreich war, gibt sie den booleschen Wert `true` zurück und die Methode `ShapTerminal.sendInitShapFile()` sendet daraufhin die Datei `SystemDaemon.shap` zur Initialisierung des SHAP-Systems.

2.3.2 Vermittlung von Paketen

Es können nun Bytes gesendet werden. Das Übertragungsprotokoll ist allerdings paketorientiert. Das heißt, für eine fehlerfreie Interpretation müssen diese Pakete auch seriell gesendet und empfangen werden. Dafür muss verhindert werden, dass mehrere Threads diese Kommunikationsfunktionen zeitlich überschneidend ausführen. In Java kann dieses Problem mit Synchronisierung gelöst werden. Es gibt zwei Arten: sogenannte synchronisierte Methoden (*synchronized methods*) und synchronisierte Anweisungen (*synchronized statements*). Letzteres ist im Allgemeinen aus Leistungsgründen vorzuziehen. Im Terminal existiert dieses Problem nicht, da hier nur ein Thread sendet, das `ShapTerminal`, bzw. ein Thread vom Stream liest, der `TerminalReader`. Auf SHAP-Seite aber werden viele Applikationen mit ihren Threads laufen und nur der

[6] Für den hexadezimalen Code eines Zeichens siehe weiterführenden Link je Zeichen unter: http://en.wikipedia.org/wiki/Seven-segment_display_character_representations

[7] Verfügbar unter: http://rxtx.qbang.org/wiki/index.php/Two_way_communcation_with_the_serial_port

[8] Die Datei `config.txt` muss im gleichen Ordner wie die ShapTerminal.jar liegen, so wie es auch für `help.txt` der Fall ist.

Systemdienst darf Ankommendes aus erster Hand erhalten. Die Applikationen erhalten ihren Input nur dann, wenn ein INPUT-Paket mit ihrer Applikations-ID beim Systemdienst erkannt wurde. Generell haben Applikationen innerhalb ihres Horizontes keine Information über die tieferliegende, paketvermittelnde Schicht und versuchen ihre Bytes direkt von `System.in` zu lesen und auf `System.out` zu schreiben. Diese Streams werden bei der Initialisierung von SHAP durch das Hochladen der ersten .shap-Datei mit `Core.getSystemIn()` und `Core.getSystemOut()` erstellt und in der Klasse `System` als statische finale Objekte gespeichert. Und genau hier bei der Generierung muss angesetzt werden.

Um den Betrieb ohne Systemdienst weiterhin zu gewährleisten, werden zunächst zwei Zustände unterscheiden: „Systemdienst läuft“ und „Systemdienst läuft nicht“. Dafür wird in der Klasse `System` die boolesche Variable `isSystemDaemonRunning` erstellt und mit `false` initialisiert. Um zu verhindern, dass nachgeladene Applikationen diesen Wert ändern können, und um zu gewährleisten, dass wesentliche Teile von SHAP darauf direkten Zugriff haben, muss sie als `protected` deklariert werden. Sie wird auf `true` gesetzt, wenn die Klassentabelle am Ende des `Bootloaders` die Klassen mit denNamen „SystemDaemon“ und „HeartBeat“ enthält. Über die `public`-Methode `isSystemDaemonRunning()` kann die Variable im Nachhinein ausgelesen werden.

Rückblick: in den Methoden `getSystemIn()` und `getSystemOut()` wird jeweils ein Input- bzw. OutputStream erstellt und deren abstrakte Methoden `read()` und `write()` konkret definiert. Sollte der Systemdienst nicht laufen und damit das Flag nicht gesetzt sein, verfährt SHAP wie gehabt und verweist sofort auf die nativen Methoden `ioRead()` und `ioWrite()`.

Läuft der Systemdienst, so müssen alle Ausgaben in ein Paket gewandelt werden. Doch kann einem extern geschriebenen Programm nicht vorgeschrieben werden, wie und mit welchen Methoden es auf `System.out` schreiben soll. Es existieren drei Schreibmethoden: `write(byte[] b)` schreibt ein ganzes Byte-Array und benutzt dazu die Methode `write(byte[] b, int offset, int length)`, die wiederum die Methode `write(int i)` benutzt, die `Core.getSystemOut()` implementiert. Diese Methode kann so umgeschrieben werden, dass für jeden `int`–Wert ein Paket generiert wird. Da in der Regel DATA-Pakete gesendet werden, steigt der Overhead auf das Vierfache der Nutzdaten – ein prinzipiell inakzeptabler Wert, nur leider ohne große Möglichkeit, dies zu ändern. Vorstellbar ist, die Daten zu sammeln und als größeres Paket zu schicken. Doch woher weiß der Systemdienst, ob die Applikation überhaupt noch Ausgabe produzieren wird, sodass er nicht im Wartezustand verharrt? Weiterhin kann es für den Nutzer wichtig sein, die Ausgabe sofort zu erhalten. Im Endeffekt gibt es bei der Verwaltung über das Terminal aber keinen Anspruch auf hohen Durchsatz bei Applikationsausgaben. Dennoch existiert eine Optimierungsvariante, die so auch implementiert wurde. `Write()` ist nicht mehr die wirklich auf den Stream schreibende Basismethode. Sie verweist mit einem einelementigen Byte-Array auf `write`

`(byte[] b, int offset, int length)`. Für den Fall, dass diese Methode direkt von der Applikation ausgeführt wird, muss nochmals abgefragt werden, ob der Systemdienst läuft. Wenn dem so ist, wird hier das Byte-Array in ein Paket gepackt und schlussendlich in einem synchronisierten Abschnitt komplett gesendet.

Für den `InputStream System.in` sieht es ein wenig anders aus. Da der Systemdienst alle Daten von diesem Stream liest, müssen die Applikationen ein anderes Eingangsmedium bekommen. Dazu wird eine `ByteArrayBlockingQueue` verwendet. Basierend auf der `java.util.concurrent.ArrayBlockingQueue`, welche Objekte in einem Ringpuffer speichert, stellt diese Klasse einen Ringpuffer für den primitiven Datentyp `byte` dar. Ein `byte` benötigt auf SHAP eine Wortgröße, also 32 Bit. Byte-Arrays sind also ebenso ineffizient. Deshalb hält die `ByteArrayBlockingQueue` ihre Daten in einem Int-Array und arbeitet mit je zwei Zeigern zum Lesen und Schreiben. Einen für das `int`-Element des Feldes und eines für das auszuwählende `byte` des `int`-Wertes. Jede Applikation erhält nun einen solchen Inputpuffer. Die Methode `System.in.read()` holt sich den Puffer der Applikation, die den gegenwärtig laufenden Thread umhüllt und ruft die Pufferfunktion `take()` auf. Sie liefert das älteste im Puffer befindliche `byte` zurück. Ist der Puffer leer, so wartet `take()`, bis er gefüllt wird. Dieses `byte` wird auf `int` gecastet und von `read()` zurückgeliefert.

3. Übertragung & Start eines neuen Programmes

Auf dem Fundament einer sicheren und fehlerfreien Paketvermittlung können nun alle Verwaltungsfunktionen implementiert werden. Um diese sinnvoll testen zu können, ist stets mindestens eine laufende Applikation notwendig. Die erste zu realisierende Funktion ist demnach das Starten einer solchen. Es wird sich zeigen, dass sie zugleich den größten Aufwand benötigte.

3.1 Initialisierung von SHAP aus einer .shap-Datei

Der initialisierende Startvorgang ist in SHAP ein komplexes Zusammenspiel zwischen ShapLinker, der die .shap-Datei erstellt und dem `Bootloader`, der diese Datei seriell verarbeitet. Am Ende dieses Prozesses steht der erste echte Thread `Startup`. Er startet die `main`-Routine des eigentlichen Programms. Es folgt eine detailliertere Betrachtung[9]. Der ShapLinker ist ein kompliziertes Gebilde und braucht zunächst nicht zu interessieren. Um die Aufgabe zu lösen, genügt eventuell schon das Wissen über den Aufbau der aus ihm resultierenden .shap-Datei, wie sie in Abbildung 12 dargestellt ist.

Im Anfangszustand kennt SHAP keine Java-Klassen. Diese werden nebst Programmklassen erst mit der .shap-Datei geladen, und auch nur diejenigen, die der Linker für die Abarbeitung des Programmes als nötig erkannt hat. Die Datei wird als Byte-Stream über den UART empfangen. Die `Bootloader`-Klasse steht am Beginn der Datei, da sie in ihrer ersten Phase keine anderen Klassen benutzt und den weiteren Ladevorgang übernimmt. Über die Werte `bias` und `size`, die in den ersten beiden Halbworten stecken, wird ein Objekt im SHAP-Speicher erstellt und der Rest der `Bootloader`-Klasse dort hinein geladen.

`Loader MP` ist eine Referenz auf die auszuführende `Bootloader`-Methode `loadup()` und hat den hexadezimalen Wert 0x00000005. Dabei ist das erste Halbwort die Klassennummer (Klasse 0 entspricht dem `Bootloader`) und das zweite der Methodenoffset (für `loadup()`) im Speicherobjekt. Erste Handlung von `loadup()` ist das Lesen und Speichern des 18 Wörter großen *Konstanten-Pools* (JVM CP) der JVM. Danach wird die Klassenanzahl gelesen und eine Klassentabelle dieser Größe angelegt, die eine Referenz auf das Klassenobjekt im Speicher mit der Klassennummer verknüpft. Für jede Klasse wird nun ein Klassenobjekt erzeugt, dessen Größe von `bias`, `size` und dem `control word` abhängt. Es wird geschrieben und am Ende in die Klassentabelle eingetragen.

[9] Ausführlich nachzulesen in dem Paper *„Java-Programmed Bootloading in Spite of Load-Time Code Patching on a Minimal Embedded Bytecode Processor"*. Thomas B. Preußer & Rainer G. Spallek, CSREA Press, 2008, S. 260-264. Verfügbar unter: http://shap.inf.tu-dresden.de/paper/esa08_preussers.pdf

Alle Klassen sind geladen. Referenzierungen zwischen den Klassen wurden vom Linker erstellt und basieren auf den Klassennummern, wie sie auch in der .shap-Datei vorkommen. Prinzipiell funktioniert das auch, wenn man bei jeder Referenz in der Klassentabelle nach der Referenz auf das tatsächlich im Speicher liegende Klassenobjekt schaut. Wegen offensichtlichen Leistungsnachteilen werden nun diese Klassennummern gepatcht. Der Block Class Patches ist dazu gedacht, die Orte solcher Referenzen dem SHAP zu übermitteln. Außerdem enthält er das sogenannte Cohen's Display. Es enthält alle Basisklassen, von der eine Klasse erbt und dient zum Beispiel zur Verifizierung eines Casts, also einer Typumwandlung.

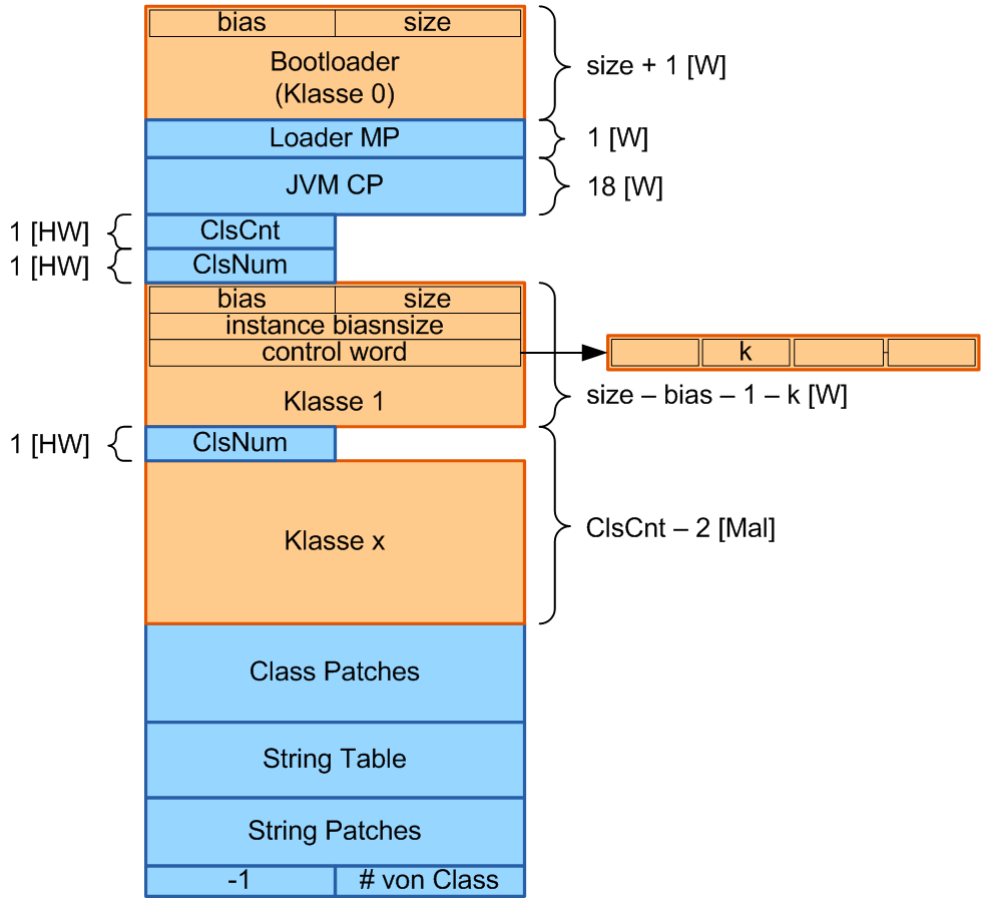

Abb. 12: Aufbau einer .shap-Datei, erstellt vom ursprünglichen ShapLinker

Damit ist die erste `Bootloader`-Phase beendet und `phase2()` beginnt. Sie kennt zum ersten Mal richtige Klassen. Davon wird auch gleich Gebrauch gemacht. Der Block String Table wird gelesen und in ein String-Array gespeichert. Anschließend werden die String Patches, analog zu den Class Patches, geladen und verarbeitet. Zuletzt wird aus dem rohen Speicherobjekt der Klassentabelle eine Hash-Tabelle, die die Klassenobjekte mit ihren Namen verknüpft.

Was weiter geschieht, zeigt Abbildung 13. Hash-Tabelle und String-Array werden einer neuen `Startup`-Instanz, dem ersten wirklichen Thread, übergeben und dieser gestartet. Das heißt, die `run()`-Methode des Threads wird aufgerufen. Die Klassentabelle

wird derweil in einer neu instanziierten, den Startup-Thread umschließenden `Application`, gespeichert.

Die Klasse `Startup` besitzt im Original keine Methode `run()`. Der ShapLinker fügt ihr allerdings diese hinzu. Für jede `main`-Routine des tatsächlichen Programms wird in `Startup` eine von Thread erbende Unterklasse erzeugt, deren `run()`-Methode die `run()`-Methode der Klasse `Thread` überschreibt. In ihr wird die jeweilige `main`-Routine gestartet. Jede dieser `main`-Routinen erhält ebenfalls eine neue `Application`-Instanz, die sie umschließt.

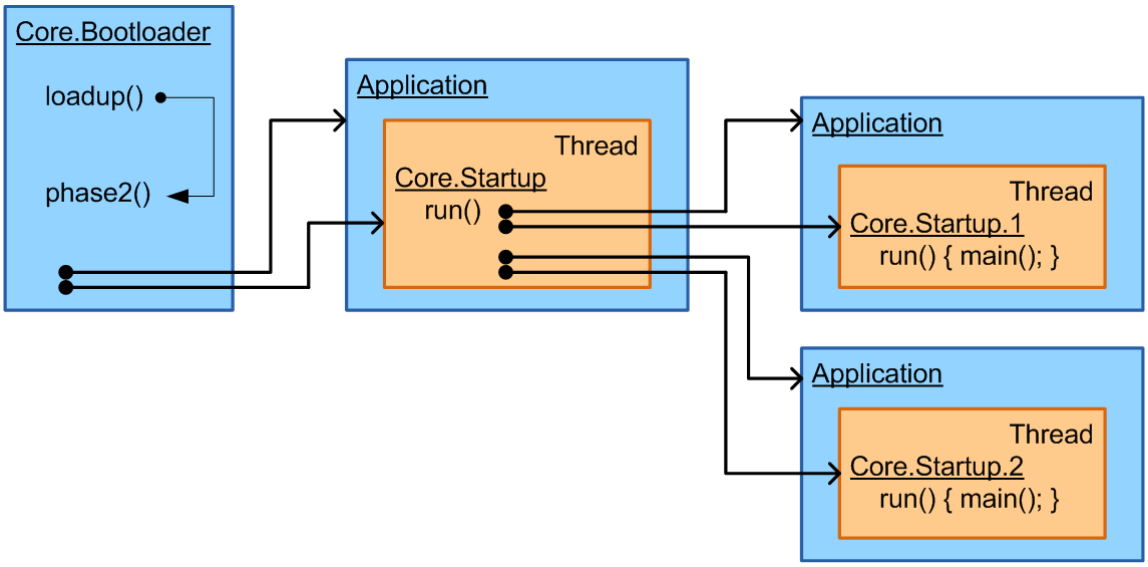

Abb. 13: Ursprüngliches Laden einer .shap-Datei

3.2 Nachladen einer .shap-Datei

3.2.1 Modifikation der .shap-Datei

Was ist auf einem initialisierten SHAP-System vorhanden, was ein zusätzliches Programm zum Ablauf benötigt? Was muss dafür mitgebracht werden und was nicht? Der `Bootloader`, der eine .shap-Datei liest, ist schon vorhanden. Er muss nicht nochmal gesendet werden, ebenso die Referenz `Loader MP` auf dessen `loadup()`-Methode, sowie der finale Verweis auf die Klasse `java.lang.Class`. Auch der JVM CP ist immer gleich und kann ignoriert werden. Das heißt, das Terminal nimmt sich aus der .shap-Datei nur alle Klassen, die Class Patches, die String Table und die String Patches.

Für einfache Programme, und bei den Ressourceneinschränkungen eines *Spartan-3* FPGAs können praktisch nur relativ simple Programme laufen, sind die meisten Klassen API-Klassen. Deshalb integriert der ShapLinker auch nur solche Klassen in die .shap-Datei, die auch tatsächlich für den Ablauf des Programmes gebraucht werden.

Hier wird augenscheinlich, dass beim Nachladen eines neuen Programmes einige API-Klassen schon vorhanden sein werden und einige aber auch fehlen können. Der Linker könnte nun alle fehlenden Klassen identifizieren und den Programmklassen hinzufügen. Aber wie soll der Linker diese ermitteln? Er müsste das aktuelle System analysieren und immer erst zur Laufzeit die neue .shap-Datei erstellen – eine inakzeptable Variante und dabei äußerst kompliziert zu implementieren. Deshalb wurde beschlossen, den Systemdienst mit sämtlichen API-Klassen, und anschließend nur noch Programmklassen hochzuladen. Dazu sind im ShapLinker zwei neue Modi erstellt worden. Erstens: der Modus „Boot“, der mit den Parameter „`-B`“ gekennzeichnet wird. Er fügt der Liste aller einzubindenden Klassen sämtliche API-Klassen hinzu, falls diese nicht schon in der Liste stehen. Der zuvor beschriebene Abschnitt im Build-File ändert sich zu der Form:

```
<target name="SystemDaemon.shap" depends="compile,ShapLinker">
    <ShapLinker main="SystemDaemon">
        <arg value="-B"/>
    </ShapLinker>
</target>
```

Der zweite Modus wird „Application“ genannt und bekommt den Parameter „`-a`“. Bei nachzuladenden Programmen müssen keine API-Klassen geladen werden. Nur für die Class Patches müssen ihre Nummern der neuen .shap-Datei mit den Referenzen der schon auf SHAP bestehenden Klassenobjekte aufgelöst werden. Das heißt, in der Klassentabelle wird an die Position ihrer Klassennummer die Referenz auf das alte Klassenobjekt eingetragen, weil für API-Klassen kein neues Objekt erstellt wird. Aber wie erhält man das alte Klassenobjekt? In der Initialisierung von SHAP wurde eine Hash-Tabelle übergeben, die Klassennamen mit Klassenobjekten verknüpft. Sie ist einzig dazu da, um im späteren Verlauf mit der Methode `Class.forName (String s)` diese Referenz zu erhalten. Dazu muss allerdings der Name der Klasse bekannt sein. Bisher hat der `Bootloader` allerdings keine Information, welche Klasse er bearbeitet. Erst wenn er am Schluss alle Strings patcht, kann er API- von nicht-API-Klasse unterscheiden. Deshalb muss bereits im ShapLinker diese Information eingetragen werden. Im „Application“-Modus werden alle zu verarbeitenden Klassen auf ihren Namen geprüft. Fängt dieser entweder mit dem String „`com/`“, „`ite/`“, „`java/`“, „`org/`“ oder „`[`“ an, so wird an erster Stelle einer API-Klasse das hexadezimale Halbwort 0x7FFF gestellt, da oxFFFF schon als Trennmarker vergeben ist. Das Halbwort entspricht der dezimalen Zahl 32767. Der `bias`-Wert bewegt sich in deutlich kleinerem Bereich, sodass es nicht zu Fehlinterpretationen kommt. Dahinter wird der Name der Klasse übergeben - für jedes Symbol des Namens-Strings ein Byte. Ein vorangestelltes Halbwort (CharCnt) gibt die Anzahl der Zeichen an. Eine Klasse im „Application“-Modus hat die in Abbildung 14 veranschaulichte Struktur.

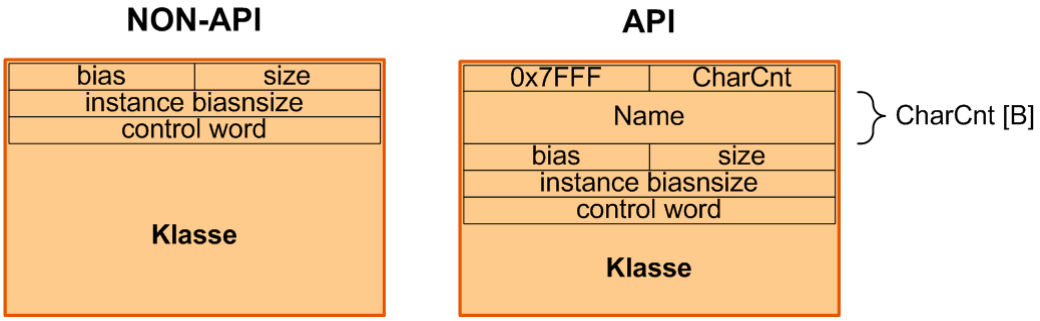

Abb. 14: Unterscheidung zwischen API- und Nicht-API-Klassen im Application-Modus

Sicherlich ist es gar nicht notwendig die .shap-Datei aufzublähen, indem man die ungenutzten Daten einer API-Klasse, also alles nach und inklusive dem `bias`, in ihr speichert. Die internen Verknüpfungen des ShapLinkers lassen allerdings keine triviale Lösung dafür zu. Der dadurch entstehende Overhead macht sich auch nur im Linking-Prozess bemerkbar und kostet im späteren Verlauf des Hochladens so gut wie keine Ressourcen mehr.

Nachgestellt sei besonderes Augenmerk auf die API-Klasse `java.lang.Core.Startup` gelegt. In ihr befindet sich die während des Linking-Prozesses erzeugte Methode `run()`. Sie ist für die Möglichkeit, Nachladen zu können, ganz entscheidend. Denn über sie startet das neue Programm. Um beim Nachladen das neue Programm zu starten, muss auch die neue Version der API-Klasse `Startup` benutzt werden. Aus diesem Grund wird `Startup` nicht als API-Klasse deklariert.

3.2.2 Segmentierung der .shap-Datei

Nachdem das ShapTerminal einen `start`-Befehl erkannt und die dazugehörige Datei gefunden hat, wird diese zunächst in ein Byte-Array umgewandelt. Nun sollen alle Klassen samt vorangestellter `ClsNum`, die Class Patches, die String Table und die String Patches eingelesen werden. Die Klassennummer der Klasse `java.lang.Class` muss nicht übertragen werden. Die Referenz auf das alte Objekt genügt hier ebenfalls. Diese Partitionierung übernimmt die Methode `ShapTerminal.fileToSegments()`. Im Abschnitt 2.1 wurde darauf hingewiesen, dass die neue .shap-Datei nicht komplett bzw. als einzelnes Paket aller benötigten Teile gesendet wird, sondern in vielen Paketen, die je ein Teil der Datei in sich tragen. Die Gründe für diese Entscheidung sind zum einen, dass ein Byte weniger für die Nutzdatengrößencodierung benötigt wird, was in Anbetracht der Vervielfältigung der zu schickenden Pakete aber zu vernachlässigen ist, aber vor allem die Ressourcenknappheit auf SHAP. Der Versuch, ein HelloWorld-Testprogramm mit ausgeschaltetem Garbage-Collector zu laden, scheitert mit dem Fehler „Out of References“. Es werden demnach kleinere Segmente benötigt, die hintereinander geladen, verarbeitet und dann gelöscht werden können, um über genügend freien Speicher für das nächste Segment zu verfügen.

Zunächst wird eine leere `ArrayList` von Byte-Array-Objekten erstellt, die im Weiteren als *Lump* (englisch für: Haufen, Batzen, Klumpen) bezeichnet wird. Anschließend wird die Größe des `Bootloaders`, also das dritte und vierte Byte der Datei, ausgelesen und damit der Offset initialisiert. Er gibt die aktuelle Position im, die .shap-Datei repräsentierenden, Byte-Array an, das nun analysiert wird. Die Klassenanzahl wird gelesen. Dabei muss immer beachtet werden, dass die Klassenanzahl die Summe sämtlicher Klassen inklusive der `Bootloader`-Klasse angibt.

Nun werden nacheinander alle Klassen samt `ClsNum` gelesen und als Byte-Array in den Lump gesteckt. Dabei müssen in jedem Fall `bias`, `size` und `k` ausgelesen werden, um mit ihnen die Klassengröße zu berechnen, für Nicht-API-Klassen um die Größe des zu kopierenden Bereiches zu ermitteln, für API-Klassen um damit den Offset bis zur nächsten Klasse zu verschieben. Für API-Klassen werden allerdings nur die Klassennummer, das API-Flag, der `CharCnt` und die Klassennamens-Chars in ein Byte-Array gepackt und dem Lump hinzugefügt.

Der Class Patches-Block trägt keine Größenangabe in sich. Für jede Klasse hat er einen Eintrag, der mit dem Markierungs-Halbwort -1 bzw. dem hexadezimalen 0xFFFF, endet. Über die Klassenanzahl kann damit das Ende des Blocks erkannt werden. Anschließend wird er zum zweiten Mal gelesen und dabei in ein Byte-Array gespeichert, dessen Initialisierungsgröße erst jetzt bekannt geworden ist, und das anschließend in den Lump gesteckt wird.

Die String Table sticht hier etwas heraus. Ihre Struktur sieht nach optimiertem[10] Linking-Prozess anders aus als ohne Optimierung. Abbildung 15 zeigt den Unterschied. Die hier eingesetzte Methode muss dies auffangen. Das erste Halbwort ist bei beiden Varianten gleich und gibt die Anzahl der enthaltenen Strings an. Im nichtoptimierten Fall folgt nun für jeden String ein Block, der aus der Anzahl der den String bildenden Zeichen besteht, den Zeichen selbst, einer 0, nochmals der Zeichenanzahl und einer abschließenden -1. `FileToSegments()` sucht nach der letzten -1 der String Table, die nach $StringCnt + 1$ Schleifendurchläufen gefunden ist, und erstellt danach wieder ein Byte-Array. Nach optimiertem Linken folgt nach der String-Anzahl die Größe des Char-Feldes. Darin sind alle Strings nacheinander enthalten. Anschließend folgt für jeden String ein Block, der dessen Anfangsposition im Char-Feld und dessen Zeichenanzahl angibt. Mit der Formel

$$x = 4 + Feldgröße * 2 + StringCnt * 4 + 2$$

ergibt sich die Größe x der String Table in Bytes. Die Unterscheidung, welcher Fall vorliegt, wird über das zweite Halbwort getroffen. Es ist also entweder die Char-Feldgröße oder der `CharCnt` des ersten Strings. Für ein HelloWorld-Testprogramm ergibt sich in der optimierten Variante eine Feldgröße von rund 2600. Andere Programme sollten demnach wenigstens ebenso große Felder generieren. Für einen Puffer

[10] Der ShapLinker arbeitet standardmäßig im Optimierungsmodus - eingestellt in der built.xml seines Apache-Ant-Projektes mit Parameter: „-o“.

bei möglichen Änderungen in der API wird hier ein Schwellwert von 2048 festgelegt. Das bedeutet, dass der erste eingetragene String nicht mehr als 2047 Zeichen besitzen darf. Ein so großer String sollte allerdings generell nie auftreten. Zumal die ersten drei Strings nach Stichprobenkontrolle stets `null`, `true` und `false` sind, auch wenn diese Art der Verifikation mittels Stichprobe eigentlich keine ist.

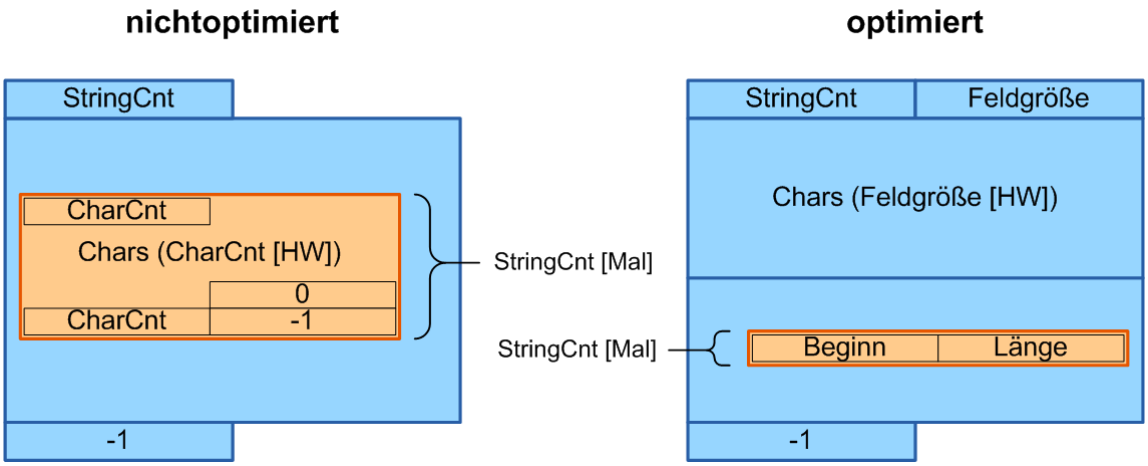

Abb. 15: Struktur der String Table nach nichtoptimiertem bzw. optimiertem Linken

Nun werden die String Patches isoliert. Es gibt hier keine codierte Information über die Größe des Blocks. Die einzelnen Einträge werden mit -1 getrennt. Es gibt allerdings auch keine Information, wie viele Einträge insgesamt vorhanden sind. Eine weitere -1 markiert den Schluss des Blocks. Das Byte-Array wird nach einer -1 suchend mit Halbwortschritten durchwandert. Das ist möglich, da alle Symboleinheiten und Chars 16 Bit groß sind. War die Suche erfolgreich, wird das darauffolgende Halbwort untersucht. Entspricht es ebenfalls einer -1, sind wir am Schluss der String Table angekommen.

Schlussendlich wird auch der Name der .shap-Datei als Byte-Array in den Lump gespeichert, um so der neuen Applikation auf SHAP-Seite einen Namen geben zu können. Der Lump enthält nun $ClsCnt - 1 + 4$ Byte-Array-Elemente.

3.2.3 Vermittlung & Verarbeitung der Segmente

Die Übertragung der Segmente und deren Verarbeitung durch den Systemdienst erfolgt alternierend und vor allem synchron. Dabei ist im ShapTerminal die Methode `sendFileSegments()` für das Senden der Elemente des Lumps und im Systemdienst die Methode `cmdStart()` für deren Empfang und Verarbeitung verantwortlich.

Folgende Erläuterungen korrespondieren mit Abbildung 16. Bei Empfang des ersten Paketes, welches die Anzahl der Lump-Elemente und die Anzahl der API-Klassen beinhaltet, wird eine neue Instanz der Klasse `Loader` erstellt und deren Methode `load1()` ausgeführt. Die Klasse `Loader` basiert auf dem `Bootloader`, arbeitet aber auf höherem Abstraktionsniveau, da das System bereits initialisiert wurde und

andere Klassen zur Verfügung stehen. Außerdem ist sie auf das Nachladen spezialisiert. Das heißt, bestimmte Abläufe, die noch im `Bootloader` abgearbeitet wurden, werden hier übersprungen. So zum Beispiel das Erstellen des Konstanten-Pools der JVM oder die andersartige Handhabung von API-Klassen. Der `Loader` ist in fünf Stufen strukturiert, die aber nicht aus der gleichen Motivation heraus entstanden sind, wie sie die Phasen des `Bootloaders` haben. Stufe 1 (`load1()`) initialisiert die Klassentabelle und eine Liste aller API-Klassen. Stufe 2 (`load2()`) lädt je eine Klasse, weshalb diese Stufe mehrfach durchlaufen wird. Stufe 3 (`load3()`) patcht alle nicht-API-Klassen. Stufe 4 (`load4()`) lädt die String Table und Stufe 5 (`load5()`) patcht diese, bevor sie schließlich eine Instanz der neu geladenen `Startup`-Klasse erstellt und zurückliefert.

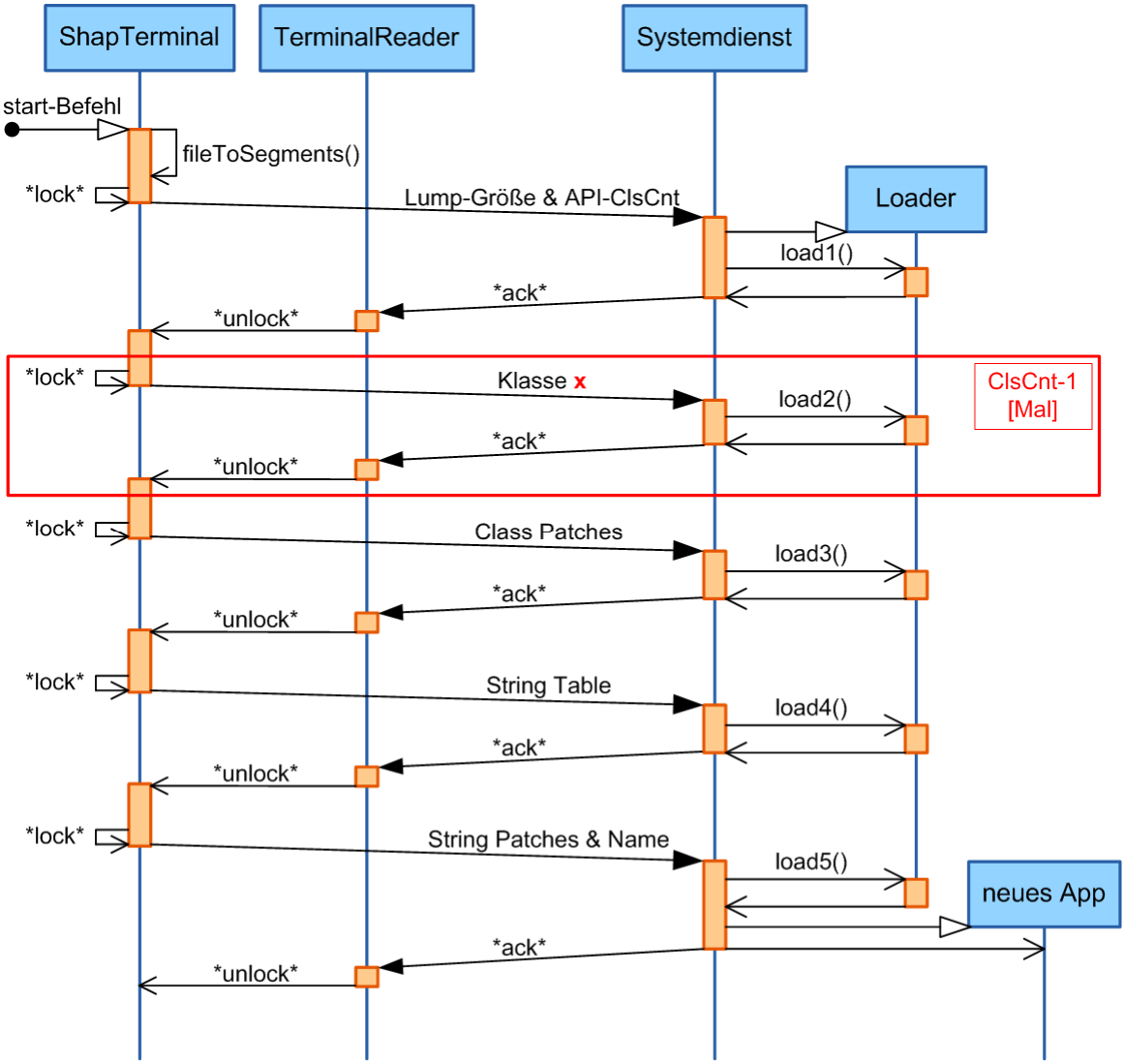

Abb. 16: Sequenzdiagramm für das Übertragen und parallele Starten eines Programmes

Jedes ankommende Paket wird also zunächst komplett gelesen und anschließend der dafür vorgesehenen Methode bzw. Stufe übergeben. Kehrt sie zurück, wird ein Acknowledgement-Paket gesendet – ein START-Paket ohne Payload und einem

Pseudo-`length`-Wert von -1, wie es Abbildung 17 darstellt. Dies ist möglich, da ein normales START-Paket nur in Richtung vom ShapTerminal zum SHAP gesendet wird, analog zur Verwendung des DATA-Paketes als Input für und Output von SHAP.

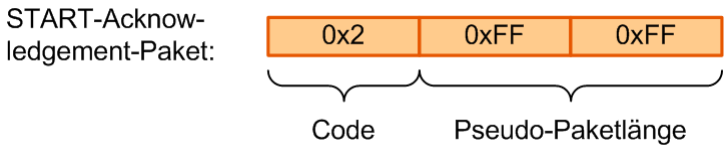

Abb. 17: Aufbau eines Bestätigungspaketes nach Erhalt und Verarbeitung eines START-Paketes

Die Synchronisierung wird über die primitive boolesche Variable `locked` realisiert, was in Java in direkter Weise nicht möglich ist. Deshalb wird das umhüllende Objekt `lock` erzeugt, das einzig dazu dient, mit Hilfe der Objekt-Klassenmethoden `wait()` und `notify()` den Wert von `locked` synchronisiert auszulesen und zu setzen.

3.2.4 Start der neuen Applikation

Ausgangssituation: SHAP wurde mit dem Systemdienst initialisiert und läuft. Alle Klassen zur Abarbeitung des neuen Programmes wurden geladen. Die Frage ist, wie die Abarbeitung des neuen Codes angestoßen werden kann. Zunächst einmal kann man nicht irgendwo anfangen. Man muss, analog zum `Bootloader` und zur Erhaltung der Applikationsstruktur, eine Instanz der neu erhaltenen Klasse `java.lang.Core.Startup` erzeugen. In ihr befindet sich auch die während des Linkens neu erstellte Methode `run()`, die alles Weitere übernimmt. Problematisch ist, dass der bereits vorhandene Code den neuen nicht kennt, und ihn deshalb auch nicht referenzieren und aufrufen kann. Das eigentliche Problem ist allerdings die Übergabe der Parameter für den `Startup`-Konstruktor, denn die Instanziierung über einen `public`-Konstruktor ohne Argumente ist über die Methode `Class.newInstance()` möglich. Nur ist es zwingend erforderlich die String- und Klassentabelle zu übergeben.

Es gibt die Möglichkeit mit dem unterliegenden Bytecode auf niedrigerem Abstraktionsniveau die Position des Konstruktors zu bestimmen und ihn aufzurufen. Dieser Weg kann allerdings leicht zu Fehlern führen, da im Bytecode keine Sicherungsmaßnahmen vorhanden sind. Außerdem wird eine Lösung auf möglichst hohem Abstraktionsgrad angestrebt, wie es auch in der Aufgabenstellung verlangt wird. Die einzige Alternative auf Hochsprachenbasis ist Polymorphie. Dabei kann ein Objekt eines bestimmten Typs sich so verhalten, als wäre es von einem anderen Typ. In Java geschieht diese Typumwandlung über Casts, Interfaces und abstrakte Klassen. Beim Cast wird das Cohen's Display durchsucht, das auch für die neuen Klassen schon erstellt wurde. Darin befinden sich Referenzen auf alle Klassen, von denen das Objekt erbt.

Abbildung 18 versinnbildlicht die nachstehenden Ausführungen. Das im Speicher liegende Klassenobjekt der neuen Klasse `Startup` lässt sich über die Hash-Tabelle, die Klassennamen auf Speicherreferenzen abbildet, erhalten. Dieses Objekt erbt von der

Klasse `Class` und kann daher mit der nativen Methode `newInstance()` instanziiert werden (siehe 1.). An dieser Stelle kommt die Typumwandlung ins Spiel. Es wird die abstrakte Klasse `AbstractStartup` definiert, von der `Startup` erben muss. Beide erhalten einen parameterlosen `public`-Konstruktor. `AbstractStartup` muss allerdings schon bekannt sein und wird deshalb als API-Klasse mit dem Systemdienst geladen. In ihr wird die abstrakte Methode `startForReal()` definiert, die später die Argumente übergeben soll. Die zuvor erzeugte Instanz der neuen und unbekannten `Startup`-Klasse wird nun auf die bekannte Klasse `AbstractStartup` gecastet. Ab diesem Zeitpunkt kann zum ersten Mal neuer Code ausgeführt werden. Da Interfaces nicht instanziiert werden können, wird die Notwendigkeit einer abstrakten Klasse klar. Die Methode `startForReal()` wird abstrakt aufgerufen, sodass deren Implementierung aus der neuen Klasse `Startup` ausgeführt wird (siehe 2.). Dabei wird die String-Tabelle gespeichert und als eine neue, den `Startup`-Thread umhüllende `Application` mit der Klassentabelle erzeugt. Die erstellte Instanz wird dem Systemdienst als `AbstractStartup`-Objekt von `load5()` zurückgegeben. Dieses Objekt erbt von `Thread` und wird nun über dessen Methode `start()` gestartet (siehe 3.). Sie verweist über Zwischenstationen auf die `Thread`-Methode `run()`, die allerdings in der Klasse `Startup` überschrieben wird (siehe 4.). Somit also wird die vom Linker generierte Methode ausgeführt. Sie startet analog zu Abbildung 13 nun jede `main`-Routine in eigenem Thread und eigener `Application` (siehe 5.).

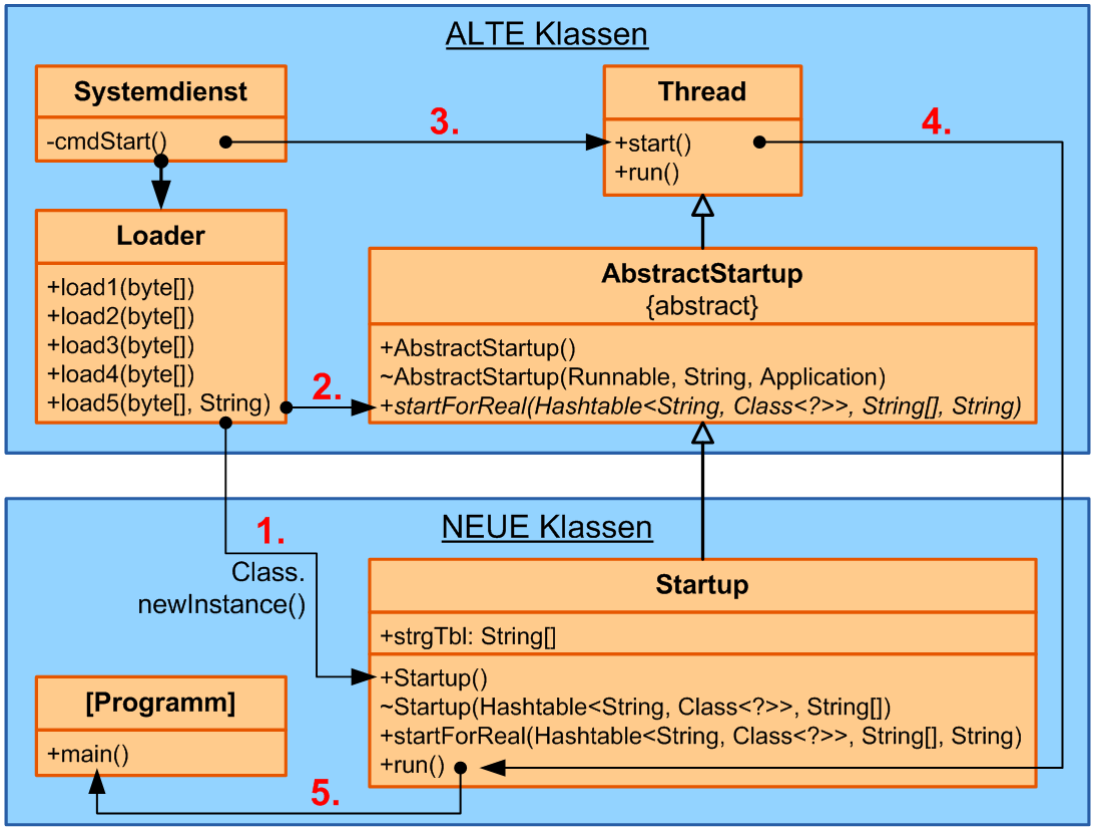

Abb. 18: Start einer neuen Klasse mit Polymorphie über `AbstractStartup` und `newInstance()`

4. Implementierung aller Aufgaben zur Applikationsverwaltung

4.1 Der Befehl LIST

Wie schon in Abschnitt 2.1 zum Übertragungsprotokoll beschrieben, benötigt der Benutzer des Terminals, um diverse Kommandos zu schreiben, die ID der jeweiligen Applikation. Dazu wird mit dem Kommando `list` ein LIST-Paket gesendet und vom Systemdienst eine Zeichenkette über ein DATA-Paket zurückgeschickt. In ihr sind Applikationsnamen mit Applikations-IDs verknüpft. Der Systemdienst soll als Basisapplikation und als erstes geladenes Programm stets die Identifikationsnummer 0 bekommen. Um nicht jedes Mal auf die Applikations-Instanzen zugreifen zu müssen, wird die Liste explizit gespeichert.

Welchen Datentyp hat die Liste? Die SHAP-API kann als Teilmenge der Java-API mit einigen nötigen Erweiterungen angesehen werden. Da sie in Relation nicht sehr mächtig ist, gibt es zur Beantwortung der Frage nur zwei Möglichkeiten:

- ein `Vector`
- oder eine `Hashtable`.

Der Vektor scheidet aus, da keine expliziten IDs vergeben werden können, sondern nur die Position im Vektor der ID entspricht. Das bedeutet, dass beim Löschen eines Elementes alle Folgenden nachrücken und ihre bisherige ID verlieren. In einer Hash-Tabelle lassen sich Objekte mit Schlüssel-Objekten verknüpfen und alle Einträge zusammengefasst als String ausgeben. Weil der primitive Datentyp `int` kein Objekt ist, muss als Schlüssel-Typ Integer benutzt werden. Der Hauptbenutzer dieser Liste ist der Systemdienst. Sie wird allerdings auch zur Sicherung von einigen `public`-Methoden der API genutzt und muss daher auch dort gespeichert werden. Die Entscheidung fiel auf die Klasse `java.lang.System`, da hier auch die Variable `isSystemDaemonRunning` definiert ist und beide bei der Sicherung zusammenarbeiten. Der Systemdienst trägt sich in seiner `main`-Routine als erstes Element mit der expliziten ID 0 selbst ein. Alle weiteren Applikationen werden am Ende des Nachladevorgangs aufgenommen, falls die maximale Anzahl von IDs, und damit parallel laufender Applikationen, noch nicht erreicht ist. Wird eine Applikation beendet, steht dessen ID bei späterem Nachladen wieder zur Verfügung. Dafür sorgt die Systemdienst-Methode `getNewID()`. Ist ein Thread mit seiner `run()`-Methode fertig, wird er gelöscht und die Anzahl aktiver Threads seiner Applikation aktualisiert. Wird die Anzahl Null, soll auch die Applikation gelöscht werden, namentlich aus der Applikationsliste `appList`. Eine Ausnahme soll implementiert werden. Um die gespeicherte Ausgabe eines Programmes nicht zu verlieren, wird geprüft, ob der Ausgabepuffer nicht leer ist. In dem diesem Fall wird die Applikation erst dann gelöscht, wenn die Restausgabe mit dem `listen`-Befehl ausgegeben wurde. Mit `kill` können Programme zu jeder Zeit terminiert werden.

4.2 Die Befehle LISTEN, UNLISTEN, MUTE und CLEAR

Eine Vielzahl von Applikationen kann nun parallel auf SHAP laufen und angesprochen werden. Einige von ihnen produzieren Ausgaben, die, wie in Abschnitt 2.3.2 schon erwähnt, verpackt und zum Terminal geschickt werden. Das Terminal gibt diese Ausgaben samt Angabe der betreffenden Applikations-ID auf der Konsole aus. Überschneiden sich viele Ausgaben wird es für den Benutzer schnell unübersichtlich und manch ein User möchte eventuelle Ausgaben vielleicht gar nicht erst sehen oder erst zu einem späteren Zeitpunkt. Die Terminal-Kommandos `listen`, `unlisten` und `mute` sind dazu gedacht, einer Applikation die direkte Ausgabe zu erlauben, nur indirekt zu erlauben oder sie komplett zu ignorieren. Dazu muss zunächst für jede Applikation der Ausgabestatus `out_state` definiert werden, der die Werte `LISTEN = 1`, `UNLISTEN = 2` und `MUTE = 3` annehmen kann. Der Systemdienst soll immer auf `LISTEN` stehen. Etwaige Änderungswünsche auf die Applikation mit der ID 0 werden verweigert. Über einen Parameter im `start`-Kommando kann für nachladende Programme der initialisierende Ausgabestatus angegeben werden. Dazu wird dem Lump ein weiteres Byte-Array hinzugefügt, das standardmäßig den Wert `UNLISTEN` enthält. Stehen nach dem Dateipfad mit Leerzeichen getrennt die Zeichenketten „`-l`“, „`-u`“ oder „`-m`“, wird dieser Standardwert explizit geändert. Der Systemdienst erhält beim Nachladen als letztes Paket diesen codierten Wert und ändert den `out_state`, bevor er schließlich die Applikation startet.

Ignoriert das System bei `MUTE` alle Ausgaben und sendet sie für `LISTEN` sofort, so wird bei `UNLISTEN` die Ausgabe in den Ausgabepuffer der Applikation zwischengespeichert. Diese Verzweigung geschieht im OutputStream `System.out`. Der Puffer ist ein Char-Array und als Ring organisiert. Das heißt, der `outBufWritePointer` zeigt auf das nächste zu schreibende Element und das Flag `fullOutBuf` gibt an, ob der Puffer voll ist. Wird weiterhin Ausgabe produziert, obwohl kein freier Platz im Puffer existiert, so wird der älteste Eintrag überschrieben.

Während der Abarbeitung eines Programmes soll eine Änderung des Ausgabeverhaltens durch die zuvor genannten Terminal-Kommandos nebst ID der betreffenden Applikation angegeben werden. Die LISTEN-, UNLISTEN- und MUTE-Pakete werden im Systemdienst erkannt und die analog arbeitenden Methoden `cmdListen()`, `cmdUnlisten()` und `cmdMute()` werden aufgerufen. Zunächst wird überprüft, ob die Applikation existiert und nicht schon im gewünschten Modus arbeitet. Falls nicht, wird mit der Methode `setState()` der Puffer zurückgesetzt und der neue Wert von `out_state` gesetzt. Bei `LISTEN` wird vorher der Inhalt des Puffers gesendet. Die Ausgabe wird also immer maximal ein Mal übertragen. Schreib-, Lese- und Rücksetzoperationen auf dem Puffer können bei Überlagern zu Fehlern führen und sind daher synchronisiert.

Mit Hilfe des `clear`-Kommandos soll der Ausgabepuffer zurückgesetzt werden. Dazu wird `setState()` einfach mit dem alten Ausgabestatus ausgeführt.

4.3 Der Befehl INPUT

Das Herzstück zur Realisierung des Inputs, der Inputpuffer als `ByteArrayBlockingQueue`, und dessen Leseverfahren wurden bereits im Abschnitt 2.3.2 vorgestellt. Bevor Applikationen daraus lesen können, muss dieser zunächst gefüllt werden. Mit dem `input`-Kommando ist das möglich. Ihm folgen die Applikations-ID und die Eingabezeichenkette. Voran- und nachgestellte Leerzeichen werden ignoriert. Im Systemdienst wird die Existenz der angegebenen ID überprüft und die Nutzdaten isoliert. Falls nun der Input größer ist, als der Puffer freien Platz besitzt, wird der ganze Input verworfen und dem Terminal eine Nachricht übermittelt. Den Puffer bis zum letzten freien Platz zu füllen bringt hier deutlich mehr Schaden durch Verwirrung, als Nutzen durch Effizienz. Denn kann der Nutzer nur schwerlich feststellen, welche Zeichen seiner Eingabe nun im Puffer gespeichert wurden und welche ignoriert werden mussten. Hier ist es sinnvoller zu warten, bis die Applikation die bisherige Eingabe verarbeitet hat, um dann mit Hilfe der Konsole schnell und benutzerfreundlich das letzte Kommando, in der Hoffnung nun Akzeptanz zu finden, wieder aufzurufen.

4.4 Der Befehl STATUS

Ähnlich wie `list` arbeitet auch der Befehl `status`. Auch er erzeugt einen String und beinhaltet dort die Applikationsnamen und –ID. Zusätzlich werden allerdings noch andere System- und Applikationsinformationen aufgeführt. Die Palette der Möglichkeiten ist lang. Die aktuelle Implementierung unterstützt: für jede Elternapplikation die ID, den Namen, den Ausgabestatus, den Füllstand des Ein- und Ausgabepuffers samt Puffergröße und die Namen der Kinderapplikationen nebst Anzahl aktiver Threads. Die Zahl der freien Referenzen und Segmente von SHAP wird zum Schluss integriert.

4.5 Der Befehl KILL

Durch `kill` soll eine noch laufende Applikation inklusive aller Kinder und Threads terminiert werden. In der SHAP-API ist eine solche Funktion nicht enthalten. Die Klasse Thread besitzt die nichtstatische native Methode `finish()`, die den aktuell ausführenden Thread beendet. Es ist aber nicht möglich, von außen einem Thread den Aufruf dieser Funktion zu befehlen. Deshalb ist eine neue native Methode `kill()`[11] nötig, der eine Referenz auf ein zu beendenden Thread übergeben wird. Sie wird mit Mikrocode implementiert. Dabei ist zu beachten, dass der beendete Thread seine Terminierung seinen Ressourcen nicht mitteilt. Wird damit in einer interagierenden Gruppe ein Thread von ihnen geschlossen, kann das zu Problemen führen. Da hier

[11] Dem Bytecode wird der neu erstellte Befehl `thread_kill` mit dem Opcode 0xF3 hinzugefügt. Dies wurde von Herrn Preußer, dem Betreuer des Beleges, und nicht vom Autor getan.

ganze Applikationen, also semantisch abgeschlossene Gruppen, beendet werden, tritt kein Problem auf. Der gesamte Löschdurchlauf über die Kinder und Threads wird synchronisiert über die Applikationsliste `appList` der Klasse `System` durchgeführt. Am Ende wird auch der Eintrag der Zielapplikation aus `appList` gelöscht.

4.6 Der Befehl RESET

Ein wesentlicher Vorteil der ferngesteuerten Verwaltung ist es, nicht mehr immer genötigt zu sein, den Soft-Reset-Knopf des FPGAs zu drücken, wenn SHAP wieder initialisierbar gemacht werden soll. Auch zuvor konnte man eine Applikation vom Terminalrechner aus steuern, aber nicht das SHAP-System selbst. Darum war es dennoch notwendig das FPGA in räumlicher Nähe zu halten. Trotzdem kann auch der Systemdienst oder eine auf ihm laufende Applikation das System immer noch zum Absturz bringen. Der Weg über den Reset-Knopf des FPGA wäre dann immer noch der einzige.

Die SHAP-API bietet eine vorgefertigte Möglichkeit zum Zurücksetzen und nutzt diese bereits, wenn die Abarbeitung des initialisierenden Programmes erfolgreich beendet wird. Damit macht sich SHAP selbst wieder initialisierbar. Es wird die native Methode `Thread.ioWrite()` verwendet und ihr der Wert 0 und die Adresse 0x2000000 übergeben. Damit wird ein spezielles Reset-Gerät des FPGAs angesprochen.

Zunächst erkennt das ShapTerminal das `reset`-Kommando und schickt ein RESET-Paket. Anschließend wartet es bis eine Bestätigung eintrifft. Nach Erhalt des Paketes im Systemdienst wird in der Methode `cmdReset()` als erstes ein RESET-Paket als Bestätigung zurückgeschickt. Anschließend wird die Funktion `System.exit()` ausgeführt, die auf `Runtime.getRuntime().exit()` verweist. Der Rumpf dieser Methode war bisher, außer dem Werfen einer `SecurityException`, leer. Hinzugefügt wurde nun eine Abfrage, ob die gegenwärtig laufende Applikation dem Systemdienst entspricht und damit die Erlaubnis zum Zurücksetzen hat. Falls Ja, wird `ioWrite()` wie oben beschrieben aufgerufen, ansonsten weiterhin die `Exception` geworfen.

Während sich das FPGA zurücksetzt, ist im `TerminalReader` die Bestätigung eingegangen. Daraufhin wird die Synchronisierungssperre aufgehoben, der bisherige `TerminalReader` wird nach einem vollständigen Schleifendurchlauf geschlossen und das `ShapTerminal` wartet erneut, diesmal die definierte Zeit `RESET_WAIT_TIME`, bis das FPGA sich zurückgesetzt haben muss. Daraufhin wird die Datei `SystemDaemon.shap` zur erneuten Initialisierung gesendet und eine neue `TerminalReader`-Instanz erzeugt und gestartet. Wieder beginnt die Ausgabe erst

nach Eingang des ersten validen INIT-Paketes. Standardwert der RESET_WAIT_TIME sind 100 Millisekunden. In der config.txt sind es aktuell 30 Millisekunden.[12]

Als Synchronisierungsobjekt wird wieder `lock` und indirekt die primitive Variable `locked` benutzt. Das ist möglich, da Kommandos im Terminal immer nacheinander abgearbeitet werden und sich so zum Beispiel das Starten und Zurücksetzten mit der Benutzung von `lock` nie gegenseitig behindern können.

4.7 Wiederverbindbarkeit

SHAP und Terminal arbeiten auf getrennten Plattformen. Fällt SHAP aus, ist die Arbeit des Terminals hinfällig. Fällt das Terminal aus, sei es, weil der Rechner abstürzt oder versehentlich das Terminalprogramm geschlossen wird, arbeitet der SHAP dennoch weiter und wartet auf Eingabe. Um auf einen schon initialisierten SHAP wieder zu verbinden, muss das Terminal zunächst erkennen, ob dies der Fall ist. Ansonsten würde er die initialisierende .shap-Datei senden, mit dem der Systemdienst nichts anfangen kann, außer dass er sie fehlinterpretiert und diverse Aufgaben anstößt. Das Lebenszeichen muss also vom Systemdienst selbst ausgehen und kann auch nicht vom Terminal angefragt werden. Dazu wird neben dem `SystemDaemon` ein weiterer Thread erstellt, der periodisch das Lebenszeichen sendet. Dazu wurde die Klasse `HeartBeat` und eine neue Paketform mit dem Namen INIT definiert. Die Periode ist aktuell eine Sekunde und das INIT-Paket hat den Opcode 0x0 und darüber hinaus zwei Pseudo-`length`-Bytes die das Halbwort 0xFFFF bilden. Damit wird gleichzeitig eine periodische Überprüfung der Paketvermittlung durchgeführt. Wird nach dem erkannten Opcode 0x0 das erwartete Halbwort nicht erkannt, wird der Benutzer informiert, da eine Störung vorliegen muss.

Das erste INIT-Paket wird noch vor Beginn der eigentlichen Arbeit des Systemdienstes übertragen und erfüllt damit eine weitere Funktion. Bei der Initialisierung von SHAP wird auch der Garbage-Collector gestartet. Als einzige Ausgabe des SHAP-Systems selbst werden dabei, und noch vor dem Start des eigentlichen Programmes, die Zeichen „G“ und „C“ gesendet, wodurch es zu einer Kollision mit unserer Paketvermittlung kommt. Es wird fälschlicherweise ein RESET-Bestätigungs-Paket erkannt und damit die Variable `locked` gesetzt. Dies zu verhindern wartet der `TerminalReader` zunächst auf ein INIT-Paket vom Systemdienst, bevor er mit seiner Arbeit beginnt.

[12] Tests für 20 Millisekunden waren stets erfolgreich, für 10 Millisekunden nicht erfolgreich. Der worst case wurde aber nicht analysiert.

5. Schlussbetrachtung

5.1 Ausbaumöglichkeiten

Ein Byte codiert den Opcode eines Paketes. Die bis hierher beschriebenen Befehle nehmen 10 von 256 möglichen Paketarten in Anspruch. Das Terminal hat dahingehend sogar überhaupt keine Beschränkung. Und auch Pakete lassen sich zusammenfassen, wenn ein weiteres Byte als Unteropcode fungiert. Will heißen, es ist genügend Raum für weitere Funktionalitäten. Welche können das sein und wie können sie umgesetzt werden?

Die meisten Programme können in irgendeiner Weise konfiguriert werden. So ist es auch vorstellbar, das Terminal oder den Systemdienst einzustellen. Das Terminal liest bereits beim Verbinden einige Werte aus der Datei `config.txt` und justiert damit die serielle Verbindung und stellt die `RESET_WAITTIME`. Es gibt nicht viel mehr wesentliche Variablen. Die Änderung der Werte in der Konfigurationsdatei muss bisher allerdings extern erledigt werden. Dies kann über ein neues Kommando mit Parameter und Wert auch über das Terminal realisiert werden. Der Systemdienst könnte einige seiner Variablen global, und teils notwendiger Weise in der API, definieren. Anschließend kann ein `Konfiguriere-SHAP`-Befehl wieder mittels Parameter und Wert, nach Prüfung der Erlaubnis, eine solche Variable ändern. Dabei muss stets darauf geachtet werden, ob eine Änderung zur Laufzeit sinnvoll ist oder zu Problemen führt.

Fast jede, und besonders neu entwickelte, Software birgt Verbesserungs- und Erweiterungsmöglichkeiten. Der Software-Entwickler muss dabei abwägen inwieweit er Erweiterungsmöglichkeiten für eine höhere Leistung verringert. Einige wesentliche und interessante Bestrebungen sollen nun angesprochen werden.

Einen optimierten Umgang mit Ressourcen, also Speicherplatz oder Rechenzeit, wird stets angestrebt. Viele der implementierten Funktionen können sicherlich noch effizienter und schneller arbeiten, doch muss man bei Neuentwicklungen auch auf Lesbarkeit und Nachvollziehbarkeit im Code achten. Kommentare können nicht alles erklären. Besonders bei der Nutzung von Datenstrukturen ist viel Speichersparpotenzial enthalten. Die Verwendung einer `ByteArrayBlockingQueue` für den Eingangspuffer einer Applikation zeigt, was gemeint ist. Bei langlebigen Daten wirkt sich der Overhead am deutlichsten aus. Die String-Tabelle einer Applikation ist ein solches Beispiel. Viele ihrer Einträge werden von nachgeladenen Programmen nicht genutzt, da sie der API entspringen und alle API-Klassen auf ihrer Tabelle arbeiten. Diese API-Strings zu finden und zu löschen würde die String-Tabelle um schätzungsweise 5 KByte[13] verkleinern. Dieser Gewinn wird für jedes nachgeladene Programm erreicht.

[13] Dazu wurden die Werte des in Abschnitt 3.2.2 erwähnten HelloWorld-Testprogrammes verwendet: rund 2500 Zeichen für alle API-Strings und ein Verbrauch von zwei Bytes pro Zeichen.

Für eine spätere Erweiterung der Applikationsstruktur auf SHAP muss die aktuell vorhandene Hierarchie umgestaltet werden. Da der `Loader` auf Basis des `Bootloaders` arbeitet, wird für ein nachgeladenes Programm nicht nur eine neue Applikation erstellt, sondern, wie in Abbildung 19 skizziert, ebenfalls eine neue Elterngeneration. Das ist aktuell noch kein großes Problem, da unsere Hierarchie immer nur zwei Stufen besitzt. Später soll es aber möglich sein, Applikationsbäume zu erstellen, wie sie in Abbildung 20 angedeutet sind.

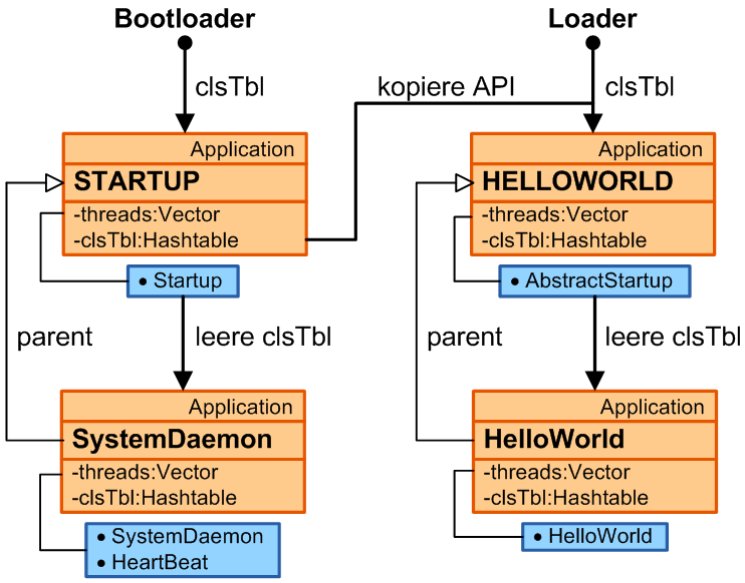

Abb. 19: Aktuelle Implementierung der Applikationsstruktur

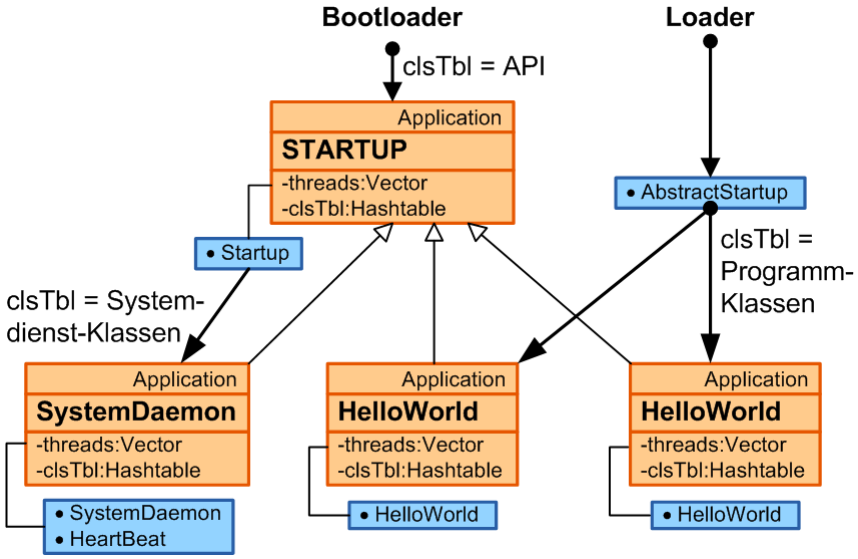

Abb. 20: Anvisierte Hierarchie mit verteilten Klassentabellen

Die Unterteilung der Klassentabellen und deren Speicherung in der jeweiligen `Application`, sollte für die angestrebte Anordnung keine trivial zu lösende Aufgabe sein. Die Zielinstanz der Klassentabelle wird erst in der `run()`-Methode von `Startup`

erstellt. An deren Schluss allerdings schon die `main`-Routine gestartet wird. Die Einbettung der `ClsTbl` setzt deshalb wahrscheinlich eine Änderung des ShapLinkers voraus. Außerdem muss die ganze Ein- und Ausgabestruktur geändert werden. Nur die ältesten `parent`-Applikationen halten momentan die Input- und Output-Puffer bereit. Applikationen zweiter Ebene führen eher ein Schattendasein.

Schließt sich das Terminal während SHAP weiter arbeitet, laufen Ausgaben auf die serielle Schnittstelle ins Leere. Das passiert aktuell auch wenn das Terminal auf mit `exit` oder `quit` geschlossen wird. Einer dieser beiden Befehle könnte nun so modifiziert werden, dass vor der Selbstzerstörung ein QUIT-Paket sendet, das dazu neu implementiert werden müsste. Der Systemdienst sollte bei Empfang dieses Paketes anschließend alle Programme auf `UNLISTEN` stellen. Eventuell ist es auch sinnvoll den Systemdienst selbst `UNLISTEN` zu machen.

In Abschnitt 2.3.2 wurde erläutert, dass die Basismethode zum Schreiben auf den UART geändert wurde, um den Paketoverhead zu verringern. Es gibt eine weitere, aber aufwendige Möglichkeit. Verschiedene Funktionen der API verweisen direkt oder indirekt auf die Methoden `write(byte[] b)` oder `write(int i)`. Falls die Ursprungsdaten hier vorher zerlegt wurden, um sie atomar zu senden, kann versucht werden, das Original direkt mit `write(byte[] b, int offset, int length)` zu senden.

Die gesamte Entwicklung der hier vorgestellten Software wurde auf einem Windows-System durchgeführt. Eine Portierung auf Linux ist wichtig und machbar. Der in Java geschriebene Code ist plattformunabhängig und braucht nicht geändert zu werden. Einzig die Benutzung von vorgefertigten Strings muss wegen sich unterscheidenden Zeichensatztabellen überarbeitet werden. Weiterhin muss RXTX in der Linux-Variante eingebunden sein.

Heutzutage werden Programme immer seltener von der Konsole aus gesteuert. Eine *grafische Benutzeroberfläche* (GUI) kann in Java relativ leicht erstellt werden. Eine GUI bietet vor allem auch für die Verwaltung von SHAP den Vorteil einer übersichtlichen Trennung der Programmausgaben. Um auf einer ShapTerminal-Backend-Version eine gleichzeitige Nutzung von Konsole- und GUI-Frontend zu erlauben, müssen Back- und Frontend eine definierte Schnittstelle erhalten. Aktuell werden Antwort-Strings komplett auf SHAP-Seite generiert und am Terminal nur angezeigt. Hier müsste optimaler Weise die Paketstruktur in der Art geändert werden, dass jedem Anfrage-Paket-Typ ein Antwort-Paket-Typ zugewiesen wird, der seine Nutzdaten in definierter Weise zusammensetzt.

5.2 Fazit

Das seit 2006 bestehende SHAP-Projekt konnte durch diesen Beleg erweitert werden. Eine parallele Abarbeitung von Programmen war bereits vorher möglich. Allerdings nur insofern, dass sie in eine einzige .shap-Datei gelinkt und gleichzeitig hochgeladen und gestartet wurden. Dabei war eine Steuerung der Applikationen im anschließenden, laufenden Betrieb nicht möglich. Diese beiden Lücken sind nun geschlossen. Programme können unabhängig voneinander und zu verschiedenen Zeiten installiert werden. Der Systemdienst ist ein wirklicher Dienst. Er kapselt, in Zusammenarbeit mit dem ShapTerminal und auf Basis des Übertragungsprotokolls, die Kommunikation aller Applikationen. Dabei kann die Handhabung der Applikationsausgabe eingestellt werden. Der Systemdienst übernimmt weiterhin grundlegende Verwaltungsaufgaben für das SHAP-System. Zum einen werden die laufenden Programme gesteuert, zum anderen kann auch in das System selbst eingegriffen werden, zum Beispiel mit einem Reset.

Um dieses Ziel zu erreichen, wurden einige Änderungen und Erweiterungen in der SHAP-API vorgenommen.[14] Außerdem hat der ShapLinker zwei neue Modi und der Bytecode einen neuen Befehl bekommen. Alle Zielstellungen konnten erfolgreich erreicht werden. Die Ressourcennutzung eines stets im Hintergrund laufenden Dienstes sollte optimiert werden, um den Zielprogrammen die besten Bedingungen zu bieten. An dieser Stelle ist die hier entwickelte Software noch verbesserbar.

Terminal, sowie Protokoll und Systemdienst sind ausbaufähig und haben einen, für spätere Erweiterungen, gut nachvollziehbaren Aufbau. Damit bieten sie die Möglichkeit neue Features, wie eine grafische Benutzeroberfläche oder weitere Verwaltungsbefehle zu integrieren. Die Erstellung einer nichtstufenbegrenzten Applikationshierarchie ist ebenfalls ein Interessanter Punkt zum weiteren Ausbau von SHAP. All das könnten Inhalte neuer Belege darstellen.

[14] Dazu zählen die neuen Klassen `AbstractStartup`, `Loader` und `ByteArrayBlockingQueue` und weitreichende Modifikationen der Klassen `Application`, `Core`, `OutputStream`, `Runtime`, `System` und `Thread`.

VI. Literaturverzeichnis

An A-Z Index of the Windows XP command line. Verfügbar unter: (Abruf: 13.12.2010 20:57) http://ss64.com/nt

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN, NAM: *Operatoren in Java*. Verfügbar unter: (Abruf: 13.12.2010 19:44) http://num.math.uni-goettingen.de/schaback/teaching/texte/informatik/skript/texte/operatoren.html

JARVI, T: *RXTX*. Verfügbar unter: (Abruf: 13.12.2010 19:12) http://www.rxtx.org

Java Modifier Summary. Verfügbar unter: (Abruf: 13.12.2010 19:29) http://www.javacamp.org/javai/modifier.html

MADHUKA: *Installing ANT for Windows 7*. Verfügbar unter: (Abruf: 13.12.2010 19:41) http://madhukaudantha.blogspot.com/2010/06/installing-ant-for-windows-7.html

ORACLE: *Java Platform, Standart Edition 6. API Specification*. Verfügbar unter: (Abruf: 13.12.2010 19:22) http://download.oracle.com/javase/6/docs/api

ORACLE: *The Java Tutorials. Synchronization.* Verfügbar unter: (Abruf: 13.12.2010 21:00) http://download.oracle.com/javase/tutorial/essential/concurrency/sync.html

PREUßER, T. B. & SPALLEK, R. G. (2008): Java-Programmed Bootloading in Spite of Load-Time Code Patching on a Minimal Embedded Bytecode Processor. *The 2008 International Conference on Embedded Systems and Applications (ESA 2008)*, CSREA Press, S. 260 - 264. Verfügbar unter: (Abruf: 13.12.2010 19:00) http://shap.inf.tu-dresden.de/paper/esa08_preussers.pdf

PROFESSUR FÜR VLSI-EDA: *SHAP (Secure Hardware Agent Platform)*. Verfügbar unter: (Abruf: 13.12.2010 19:06) http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_informatik/tei/vlsi/forschung/shap_vlsi

PROFESSUR FÜR VLSI-EDA: *SHAP Runtime Classes, API Specification*. Verfügbar unter: (Abruf: 13.12.2010 19:18) http://shap.inf.tu-dresden.de/javadoc

PROFESSUR FÜR VLSI-EDA: *The SHAP bytecode processor*. Verfügbar unter: (Abruf: 13.12.2010 19:26) http://shap.inf.tu-dresden.de

SCHILD, T.: *UTF-8-Codetabelle mit Unicode-Zeichen*. Verfügbar unter: (Abruf: 13.12.2010 21:07) http://www.utf8-zeichentabelle.de

Seven-segment display character representations. Verfügbar unter: (Abruf: 13.12.2010 21:13) http://en.wikipedia.org/wiki/Seven-segment_display_character_representations

ZABEL, M.; PREUßER, T. B.; REICHEL, P. & SPALLEK, R. G. (2007): SHAP – Secure Hardware Agent Platform. *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2007)*, TUDpress, S. 119 - 126. Verfügbar unter: (Abruf: 13.12.2010 19:00) http://shap.inf.tu-dresden.de/paper/dass07.pdf