Paper Tape Project —Manual

Sven Köppel
koeppel@technikum29.de

27. Juli 2013, 08:43

Zusammenfassung

Das vorliegende Dokument soll den Leser befähigen, komplexe Manipulationen an Lochstreifen und Dateien auf Binärlevel zu verstehen und selber vorzunehmen. Komplex bedeutet, dass der sichere Umgang mit Dateitypen und Binärstreams gewährleistet ist und Tools, die fortgeschrittene Betriebssysteme wie Linux mitbringen, eingesetzt werden können, um beliebige Byteketten zu erzeugen.

Dazu wird zunächst ein Einblick in Philosophie und Benutzung unixoider Betriebssysteme geboten, anschließend werden die im Rahmen des Paper Tape Projects von 2008 bis 2013 entwickelten Tools beleuchtet sowie gegangene Irrwege in der Entwicklung einer Software, die alle denkbaren Arbeitsschritte grafisch bewerkstelligen können sollte.

Inhaltsverzeichnis

1 Einleitung

Die Verarbeitung von Lochstreifen mithilfe von Personalcomputern begründete die technikum29-Entwicklungsprojekte [1]. Die erste Software entstand 2008, es folgten viele Jahre intensive Entwicklung an diversen Unterprojekten, und diverse neue Geräte oder Ansätze und Softwareplattformen sorgten für einen Zuwachs an Bestand und gleichzeitig Unübersichtlichkeit.

Verschiedene Dokumentationsansätze sollten dies bändigen; zu Beginn reine Text-/HTML-Dateien im Stil einer programmnahen Dokumentation, welche dann zu einer kreativen Insellösungs-Website erweitert wurde, die heute noch erreichbar ist [2]. Spätestens mit der Erweiterung um Lochkartenprojekte erwies sich dies nicht mehr als praktikabel, mit der Installation eines Softwaremanagamentwerkzeugs mit integrierter Wiki, Bugtracker und Codeviewer, namentlich *Trac* [3, 4], wurde eine Lösung geschaffen, die das Potential hat, zu skalieren; defakto hat sie sich bis heute bewährt, sodass letztes Jahr diverse andere Wikis und Dokumentationsbereiche auf der technikum29-Homepage in dieses integriert wurden.

Die Online-Dokumentation hat aber einen wesentlichen anderen Berechtigungsgrund: Papierzettel gehen in ihrer schieren Anzahl, unterschiedlichen Qualität und Organisation verloren. Sie sind nicht durchsuchbar, verblassen mit der Zeit, und sind stets nur einmal an einem Ort vorhanden. Ein digitales Dokument hingegen ist leicht vervielfältigt, korrigiert, kann leicht ausgetauscht werden und ist idealerweise noch im Netz frei verfügbar und sorgt somit für eine Sichtbarkeit der Projekte weit über die Mauern des Museums hinaus. Gerade im Umfeld eines Technikmuseums ist eine derartige Präsenz meines Erachtens nicht außer Acht zu lassen, und ich lege stets großen Wert auf größtmögliche Transparenz an jeder Stelle.

So sind die technikum29-Entwicklungsprojekte von Anfang an mit dem Hintergedanken an *Open Source* entstanden. Zur Entwicklung trugen größtenteils andere Open-Source-Software bei, nicht zuletzt aus Kostengründen. Es macht meines Erachtens aber auch überhaupt keinen Sinn, sich bei unentgeltlichen (ehrenamtlichen) Projekten auf proprietäre Software zu einigen, denn es droht der *Vendor Lockin*, also die Abhängigkeit vom Hersteller. Auch macht es wenig Sinn, den Quelltext seiner Programme geheim zu halten, will man doch nichts dran verdienen. Closed Software läuft auch gegen jeden Community-Gedanken, den ich gerade mit der transparenten Entwicklung fördern will. Auch wenn sich nach 5 Jahren kaum jemand (in der Tat nicht niemand) gemeldet hat, halte ich daran dennoch fest.

Sven Köppel, Frankfurt im Juli 2013

1.1 Über dieses Dokument

Dieses Dokument versteht sich als eine Einführung in einige Themen, die allesamt exzellent dokumentiert wurden, und zu denen es eine Unmenge an Texten gibt, von den kürzesten Tutorials bis hin zu Büchern. Da ich im Leser einen mündigen Bürger des 21. Jahrhunderts sehe, erwarte ich von ihm, Stichwörter bei der Suchmaschine seines Vertrauens einzugeben, um sich bei Interesse in die Materie zu vertiefen.

Dieses »Paper« soll das Paper-Tape-Project dokumentieren und einen in Linux unerfahrenen Benutzer ermächtigen, fortgeschrittene Operationen mit Lochstreifen durchführen zu können. Im Gegensatz zu kurzen Stichpunktanleitungen ziehe ich es vor, ein wirkliches Verständnis zu generieren, weil ich der Meinung bin, dass nur so ein Wissenszuwachs entsteht. Getreu der Binsenweisheit

> Give a man a fish and you feed him for a day.
> Teach a man to fish and you feed him for a lifetime.

1.2 Listing-Konvention

In diesem Dokument wird im Wesentlichen die Benutzung der *Bourne Again Shell* (Bash) erlernt. Ein Terminalprompt in einer solchen Shell sieht in der Regel etwa so aus:

```
name@computername:/home/name $
```

Diesen Text nennt man *Prompt*, dahinter blinkt typischerweise der Cursor und erwartet eine Eingabe. Alles vor dem zu Illustrationszwecken grün gezeichneten Dollarzeichen beschreibt, als welcher Benutzer man sich auf welchem Computer eingeloggt hat und in welchem Arbeitsverzeichnis man sich gerade befindet. Für ein Codebeispiel ist dies freilich untinteressant, daher steht stets bloß das Dollar in der ersten Spalte:

```
$ einzugebendes kommando
```

Das Dollar gibt man natürlich nicht mit ein, es steht bereits da und zeigt an, dass diese Zeile im vorliegenden Dokument andeuten soll, sie stände in einem Terminal. Ansonsten gilt meist: Groß- und Kleinschreibung ist unter Unix höchst relevant (auch bei Dateinamen); Leerzeichen sind in der Shell wie bei vielen Programmiersprachen egal. Kommentare werden mit der Raute # eingeleitet und gelten bis Zeilenende:

```
$ kommando   # ein Kommentar dazu
$ # auskommentiertes kommando
```

Insbesondere kann man solche Shellkommandos in Dateien packen, sogenannte Scripte, wo Kommentare Sinn machen. Hier dienen sie der knappen Inline-Dokumentation.

Zuletzt sei das Wort *Pseudocode* erläutert: Es taucht immer dann auf, wenn ein scheinbarer Code nur illustrative Bedeutung hat.

2 Unix-Grundlagen

Die Tatsache, dass Lochstreifen eine beliebig lange Folge von 8bit-Wörtern speichern, und Massenspeicher auf Computern seit geraumer Zeit in Ordner bzw. Verzeichnisse und darin enthaltenen Dateien strukturiert sind, erlaubt die exakte Identifikation eines Lochstreifens mit einer Datei; kurzum: Den Inhalt eines Lochstreifens kann man ohne Einigung auf irgendwelche Repräsentationsformate in einer Binärdatei speichern und andersrum.

2.1 Dateien

In der Entwicklung von Personalcomputern kam es bei vielen Betriebssystemen zu Klassifikationen von Dateien: Etwa Textdateien gegen jede Art von Binärdateien, gern auch formuliert in der Abgrenzung von Zeichen- zu Byteorientierten Streams.

Interessanterweise gibt es diese Abgrenzung auf dem 1977 auf einer PDP-11 entwickelten *Unix* traditionell nicht. Der natürliche Umgang mit Streams bei Bedienung von unixoiden Betriebssystemen macht die Ein- und Ausgabe mit Hardware sehr einfach. Das war schon früher so, wenn man sein Dateisystem mit dem *Tape Archiver* aufs Bandlaufwerk gestreamt hat, und merkt noch heute jeder fortgeschrittene Benutzer eines *Linux*-Betriebssystems (Linux kann hier als Unix-Derivat betrachtet werden), der Festplattenpartitionen verschieben muss bzw. Laufwerkimages über das Netzwerk streamt. Dateien verhalten sich auf unixoiden Rechner equivalent zu *Geräten*, etwa Festplatten, CD-Laufwerken, RS232-Kommunikationsgeräten oder unidirektionalen TCP-Streams. Was macht diese unterschiedlichen Dinge identisch?

Es ist die Abstraktionsart, mit der sie betrachtet werden. Informatiker klassifizieren Datenstrukturen gerne danach, wie man auf sie zugreifen kann: wahlfrei etwa oder in dem man Zeichen für Zeichen liest. Als zweites Kriterium soll in dieser kurzen Betrachtung die Uni- und Bidirektionalität von Informationskanälen gelten. Der Bytestream, also das Byte für Byte-Lesen einer Informationsentität, mag als eine den eben genannten »Geräten« gemeinsame Zugriffsart gelten, man kann sich vorstellen, dass man eine Festplatte Byte für Byte liest, ebenso wie ein CD-Laufwerk oder eine Textdatei. Das eine serielle Schnittstelle geradezu nach diesem Prinzip funktioniert, steckt sogar in ihrem Namen, und das *Transport Control Protocol* ist das tiefste Protokoll im Stapel der Internetprotokolle, welches zwischen zwei Kommunikationspartnern, etwa zwei Computern, nichts anderes als einen solchen Binärstream aufbaut, welches genau diese Eigenschaft erfüllt.

2.2 Die Shell

Unix bietet nun sehr effiziente und eingängige Werkzeuge, mit solchen Streams umzugehen. Während andere Betriebssysteme ihren primären Bedienzugang, die sogenannte

Betriebssystem-Shell, die spätestens ab den 1970ern bei jedem Computer ein Textterminal war, so früh wie möglich (in den 1980/90er Jahren) gegen grafische Benutzeroberflächen eingetauscht haben, zeichnet ein unixoides Betriebssystem eben diese textbasierte Shell aus. Dies erfordert einerseits eine hohe Lernkurve, um »auf der Shell« (umgangssprachlich für »mithilfe des Terminal-Prompts«) sinnvolle Arbeiten zu erledigen (eine Lernkurve, die in der zweiten Hälfte des 20. Jahrhunderts selbstverständlich war), ermöglicht aber dann das wirkliche Beherrschen des Computers auf einer vergleichsweise hardwarenahen Ebene. Damit meine ich, dass man die Peripherie des Computers auf einer Weise repräsentiert bekommt, die weit über ein Symbol auf dem Desktop, welches man anklicken kann, hinausgeht. Die Repräsentation von Geräten durch spezielle Dateien ermöglicht mit den Bordmitteln der unixoiden Betriebssystem-Shell bereits eine effiziente Steuerung derselbigen.

So ist es in einer einzigen Eingabezeile möglich, den Inhalt einer Festplatte auszulesen, zu komprimieren und über Netzwerk an einen anderen Computer zu schicken, wo der empfangene Stream auf gleiche Weise dekomprimiert werden kann, um dann dort auf eine Festplatte geschrieben zu werden:

```
$ cat /dev/hda | gzip | nc 192.168.10.5 1234
```

An diesem Beispiel sieht man den besten Unterschied zu Windows: Während man dort für diese Aufgabe ein neues Programm installieren müsste, kann man hier Bordmittel des Linux-Rechners verwenden, in dem man sie geschickt kombiniert. Zum Einsatz kommt dabei Stream-Redirection, das zentrale Feature von Unix-Shells, um welches es in dieser knappen Unix-Einführung auf diesen Seiten geht.

2.3 Unix-Standardstreams und deren Verkettung

Ein Unix-Programm wird vom Betriebssystem beim Start mit drei offenen Streams ausgerüstet, die einfach da sind. Jeder, der mal etwas programmiert hat, weiß, dass er in der Regel sofort sowas wie (Pseudocode)

```
print "Hallo Welt"
```

schreiben kann, was dann auf magische Weise in einem geeigneten Terminal angezeigt wird. Während so ein Kommando, von der Namensgebung herleitend, früher einen großen Drucker angeschmissen haben mag, wird es seit geraumer Zeit an die richtige Stelle des Bildschirms gezaubert. Dafür sorgt unter anderem eine Weiterleitung des Streams.

Die drei Streams, die ein Unix-Programm a priori besitzt, lauten `STDIN`, `STDOUT` und `STDERR`. Diese gängigen Abkürzungen stehen für die Standardeingabe, Standardausgabe und Standard-Fehlerausgabe. Über die Eingabe kann man ein Programm mit Daten füttern, über die Ausgabe gibt es verarbeitete Daten aus (vergleiche *print*-Statement).

Die Shell bietet nun Werkzeuge, um diese Streams miteinander zu verbinden. Eigentlich braucht man sich fürs erste nur zwei Symbole merken:

```
$ erstes-kommando | zweites-kommando
$ kommando > neue-datei.txt
```

Der senkrechte Strich verbindet die Standardausgabe vom ersten Kommando mit der Standardeingabe des zweiten. Die eckige Klammer schreibt die Standardausgabe des Kommandos in eine Datei.

Es gibt noch diverse weitere solcher Symbole, die sehr nützlich sind. Es sei dafür etwa auf die exzellente Bash-Hackers-Wiki verwiesen [5].

Um sinnvoll mit diesen Werkzeugen umgehen zu können, gibt es einen großen Umfang an Programmen (Größenordnung mehrere Tausend, von denen man aber meist nur wenige regelmäßig benutzt), die sehr generisch sind und sich zur Verarbeitung solcher Streams eignen. Ein paar seien an dieser Stelle mit kurzem Kommentar aufgelistet:

```
$ cat file.txt               # Gibt Datei aus
$ cat file.txt | programm    # Dateiinhalt in Standardeingabe...
$ programm | tee ausgabe.txt # Ausgabe in File und Terminal
$ programm | wc              # Zaehle Laenge der Ausgabe
$ cat file.txt | wc          # Zaehle Laenge der Datei
```

Jedes Programm kann in einer solchen Weise verkettet werden. Das Pradigma dahinter ist: Kleine kurze möglichst universelle Programme, und Verarbeitung universeller Binärströme. In sehr vielen Fällen sind dies Textströme, daher existieren sehr viele Kommandos, die besonders auf die Verarbeitung von Textströmen spezialisiert sind. Zeilenweise Textdateien sind auf unixoiden Betriebssystemen der kleinste gemeinsame Nenner für strukturierte Informationen.

Meist erwähnt man in dem Zusammenhang die Unix-Philosophie [6]

- Schreibe Computerprogramme so, dass sie nur eine Aufgabe erledigen und diese gut machen.
- Schreibe Programme so, dass sie zusammenarbeiten.
- Schreibe Programme so, dass sie Textströme verarbeiten, denn das ist eine universelle Schnittstelle.

Es ist erstaunlich, dass man mit diesem minimalen Ansatz ein Maximum an Flexibilität und gleichzeitig Mächtigkeit erlangt. Für die allermeisten komplex anmutenden Aufgaben, die einem im täglichen Datenleben vorkommen, wie etwa der Umbenennung vieler Dateien, Neusortierung von PDF-Seiten, Erzeugung von Bilddateien, usw. kann man nur wenig Zeichen lange Befehlsketten auf der Linux-Shell konstruieren, die sie mit Bordmitteln lösen.

Was man dabei tut, mutet freilich auch schon sehr schnell als Programmieren an. Defakto ist die Shell ein Bindeglied zwischen in »echten« Programmiersprachen geschriebenen Programmen und deren Interaktion auf dem Computer. Ein Bindeglied, welches in graphisch orientierten Betriebssystemen nur noch über Umwege zu erreichen ist (oder etwa mit der *PowerShell* bei Microsoft ein Retro-Comeback mit ähnlichen Pradigmen erlebt). Ausführbare Programme auf dem Computer werden in der Shell zum »Kommando«.

2.4 Navigation im Unix-Dateisystem

Statt nun in aller Tiefe die verbreiteste Linux-Shell, und zwar die *Bourne Again Shell*, vorzustellen, will ich bloß noch ein paar wenige Stichworte zur Navigation verlieren.

Unter Unix gibt es ein Wurzelverzeichnis, von dem alle anderen Verzeichnisse abgehen. Laufwerke, USB-Sticks, usw. werden in diese Verzeichnisstruktur *gemountet* und nach Benutzung entfernt. Dies sind typische Verzeichnisnamen:

```
/                       - das Wurzelverzeichnis
/home/benutzername      - das Heimverzeichnis
/usr/bin                - ein Programmverzeichnis
/media/USB-STICK        - ein moegliches Mountverzeichnis
/mnt/cdrom/             - Das Slash am Ende ist wahlweise
```

Als traditionelles Mehrbenutzersystem ist man unter Unix typischerweise als einfacher Benutzer angemeldet und kann mit Kommandos wie `su` oder `sudo` zum Root-User wechseln. Es gibt Benutzergruppen, die regeln, was man darf. In seinem Heimverzeichnis hat man normalerweise alle Rechte.

In der Shell hat man immer ein Arbeitsverzeichnis, in dem man sich aktuell befindet, das *Private Working Directory*. Mit `pwd` findet man raus, wie es heißt, meist steht das aber auch im Prompt. Mit *Change Directory* (`cd`) wechselt man in andere Verzeichnisse per relativer oder absoluter Angabe. Mit `ls` gibt man ein Verzeichnislisting aus.

```
user@computer:/home/user $ pwd
/home/user
user@computer:/home/user $ cd /mnt/cdrom
user@computer:/mnt/cdrom $ ls
datei1_auf_cdrom  datei2_auf_cdrom ...
user@computer:/mnt/cdrom $ ls -l | wc -l  # Anzahl der Files
31
user@computer:/mnt/cdrom $ cd ..  # relativer Pfad: ".."=hoch
user@computer:/mnt $ cd ../home/user
user@computer:/home/user $
```

3 Umgang mit Binärdateien

Binärdateien, wie sie Lochstreifen a priori representieren, sind schwer visualisierbar, da sie ohne weitere Information über deren Inhalt keinen Blick auf ihre Struktur zulassen. Die einzige Möglichkeit, sie anzuschauen, besteht darin, jedes Byte seinem Wert nach anzuzeigen.

Da ein Byte in unserer Konvention die Zuordnungseinheit von 8 Bit entspricht, kann ein Byte im Binärsystem mit 8 Ziffern dargestellt werden, im Oktalsystem mit 4 Ziffern, im Hexadezimalsystem mit 2 Ziffern und im Dezimalsystem gebrochen mit maximal 3 Ziffern. Konventionell versieht man Zahlen mit Präfixen, um ihre Basis festzulegen:

```
eine Binaerzahl:  0b01010101  (Prefix 0b)
in Oktal:         0125        (Prefix 0)
in Hexadezimal:   0x55        (Prefix 0x)
in Dezimal:       85          (kein Prefix)
```

Für Details zur Zahlendarstellung verweise ich auf Literatur der technischen Informatik oder Internetquellen [7]. In diesem Paper behandeln wir Bytes stets als *unsigned Integers*, also vorzeichenlose Zahlen zwischen 0_{dec} und 255_{dec}.

3.1 Der Hexeditor

Eine Datei besteht nun aus einer geordneten Folge solcher Zahlen. Wir können also jede Binärdatei im Prinzip darstellen als Liste solcher Zahlen. Dies sei eine Datei, ihr Inhalt könnte auf einem Lochstreifen mit 10 Reihen stehen:

```
0x68, 0x61, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x65, 0x6c, 0x74
```

Eine solche Darstellung einer Datei erlaubt ein *Hex-Editor*. Fortgeschritten Hexeditoren ermöglichen die Darstellung in verschiedenen Basen, übersetzen parallel mittels einem *Zeichensatz* (ungeachtet des Erfolges) in Text und ermöglichen vor allem die Manipulation einer solchen Datei, in dem man Text oder Zahlen in gewählter Basis eingeben, löschen, usw. kann. Ein Hexeditor eignet sich also in letzter Konsequenz zum Bearbeiten von Lochstreifen, wenn auch nicht besonders gut, vgl. Kapitel 6.1.

Gute Hexeditoren, mit denen ich Erfahrung gemacht habe sind, unter Linux Okteta (KDE) [8], auf dem Terminal `od` sowie `hexdump`. Für Windows gibt es Hunderte.

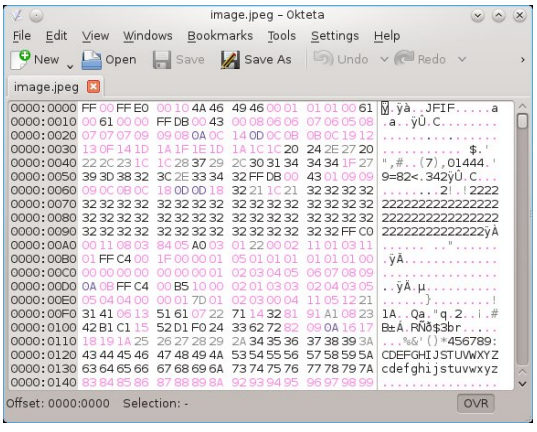

Abbildung 1: Screenshot von Okteta, Quelle: [8]

3.2 Textdateien

Ein Zeichensatz ist eine Funktion, die jedem Byte ein Zeichen zuordnet. Oft meint man mit solchen Zeichen etwa Buchstaben aus dem lateinischen Alphabet. *ASCII* ist der verbreiteste Zeichensatz, er gewann in den 60er/70ern gegen seinen Konkurrenten *EBCDIC*. Während letzterer Zeichensatz für Lochkarten mit 12 Bit breiten Wörtern konzipiert war, wurde ASCII für 7 Bit breite serielle Terminals entwickelt. Später wurden die Wörter konventionell zum 8 Bit-Nibble, sodass ASCII ein Bit undefiniert lässt. Dieses Bit verwendete man später zur Erweiterung dieses Zeichensatzes um nochmal so viele Zeichen, berühmt im deutschländigen Raum ist *Latin1* oder *ISO 8859-1*, welcher die westeuropäischen Sonderzeichen darstellen kann.

Mithilfe von ASCII oder der Erweiterung Latin1 kann man obiges Listing umrechnen in den Text

```
hallo welt
```

Insbesondere hat ASCII auch Raum für Steuerzeichen. Man hielt sich dabei an die Funktionen der Schreibmaschine, daher gibt es unter anderem den Wagenrücklauf (`CR`, Carrier Return) und den Zeilenumbruch (`NL`, Newline). Verschiedene Betriebssysteme verwenden konventionell verschiedene Darstellungen des Zeilenendes, DOS/Windows etwa `CR NL`, Unix `NL`, Mac OS verwendete `CR` [9]. Die Repräsentation des Zeilenumbruchs wird wichtig, sobald man den Computer verlässt und Peripherie oder andere Computer übers Netzwerk anspricht. Das Linux-Kommando `dos2unix` kann diese Umwandlung etwa im Stream vornehmen:

```
cat dosfile | dos2unix | tee unixfile | unix2dos > dosfile-gleich
```

Unicode ist das Bestreben, alle Zeichen der Menschheit in einem Zeichensatz aufzunehmen. Zur Repräsentation der abstrakten Tabellen, die jedem Zeichen eine Zahl bis 2^32 zuweisen, verwendet man *Transformation Formats*. `UTF-32` ist eine solche Unicode Transformation Format, in der stets vier Byte ein Zeichen darstellen (*Multbyte*). Aus dem 10 Buchstaben langen (10 Byte in ASCII/Latin1) »hallo welt« wird damit eine 40 Byte lange Datei. Da man in Westeuropa selten exotische thailändische Symbole benötigt, gibt es Übersetzungstabellen, die sich wie Latin1 verhält, also ASCII erweitert, dies ist das sehr verbreitete `UTF-8`. Dies ist im Übrigen eine Zeichenkodierung mit variabler Kodierungsbreite, der String »köppel« ist darin 7 Byte lang, da das »ö« als einziger Nicht-ASCII-Wert mit zwei Byte kodiert wird.

Die Kodierung von Lochstreifen ist für das Behandeln von Lochstreifen im Prinzip egal, erwähnen tue ich Unicode an dieser Stelle nur wegen der *Unicode Byte Order Mark* (`BOM`), eine 3-Byte-Kennung am Anfang einer Datei, die das Byte-Ordering der Datei signalisiert. Sie ist optional, doch viele Texteditoren packen sie an den Anfang [10] und man muss drauf achten, sie zu deaktivieren, wenn man die Textdatei z.B. unverändert auf einen Lochstreifen stanzen will, welcher höchstwahrscheinlich von einem

nicht-Unicode-fähigen Antikcomputer verarbeitet werden soll.

3.3 Formatierte Textdateien

Textdateien sind dahingehend universell, als dass Zeichensätze bereits sehr früh über Architekturgrenzen hinweg standarisiert waren. Textdateien sind extrem einfach zu bearbeiten, jedem Betriebssytem liegen einfache Editoren bei. Textdateien bezeichnet man oft als *Plain Text*, um sie vom *Rich Text* oder auch *Formatierten Text* abzugrenzen, das ist Text mit Formatierungen/Auszeichnungen wie »Dickschrift« oder ähnliches. Textprozessoren wie *Microsoft Word* oder *OpenOffice* erzeugen solche formatierten Dateien, Worddateien etwa oder Open Document-Dateien, die man konventionell an ihrem Dateisuffix `.doc` bzw. `.odt` oder an ihren *Magic Bytes* (Unix-Kommando `file`) erkennt.

Der eigentliche, in einer Worddatei gespeicherte Text liegt in einer Weise vor, die man nicht mit den in Abschnitt 3.2 diskutierten Transformationsmethoden lesen kann. Selbst die nach dem offenen Dokumentenstandard formatierten Open Document-Dateien sind eigentlich *Zip-Archive*, in denen einige Metadateien und zuletzt ein *XML*-formatiertes Textdokument ist, welches den eigentlichen Text enthält. Es ist nicht sinnvoll, derart formatierte Dateien automatisiert zu »entformatieren« wollen, im Sinne einer Umwandlung

```
cat worddatei.doc | word2plaintext > textdatei.txt
```

Der Grund liegt darin, dass eine formatierungsentfernende Abbildung in einem derart komplexen Dateiformat mit großer Wahrscheinlichkeit nicht eindeutig ist.

Daher ist es nur ratsam, unformatierte Textdateien auch als solche zu speichern.

3.4 Umwandlung zwischen Text- und Binärdateien

Oft ist es inpraktikabel, mit einem Hexadezimaleditor zu arbeiten. Im Laufe des Paper-Tape-Projects ist es oft dazu gekommen, dass Binärdateien ausgestanzt werden sollten, die aber in ganz anderen Formaten vorlagen. 2008 etwa hat Udo Assemblercode für eine Maschine geschrieben, und zwar mit der Tabellenanwendung *Microsoft Excel*. Ein Export in eine *CSV*-Datei und damit Plaintext-Datei ermöglichte es, einen *Compiler* darüberzujagen, der diese Textdatei, die defakto Inhalte wie im Listing

```
0x68, 0x61, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x65, 0x6c, 0x74
```

in eine Binärdatei umwandelt. Denn hier bedarf es besonderer Aufmerksamkeit: Wenn der Inhalt dieses Listings buchstabengenau Inhalt einer Datei wäre, dann wäre sie 58 Byte groß. Jedem menschlichen Leser ist aber klar, dass hier eine Repräsentation von 10 Byte stehen.

In welchem Format einem Dateien vorliegen, sollte stets klar sein. Hier eignen sich Linux-Tools wie

```
$ file dateiname
dateiname: ISO-8859 text
$ file irgendwas
irgendwas: data
$ file nochwas
nochwas: OpenDocument Text
$ ls -l dateiname
-rw-rw-r-- 1 user user 51431 Jul 24 13:49 dateiname
```

In diesem Beispiel haben die Dateien bewusst keine Dateiendung, wie unter Windows üblich. Denn unter Unix sind Dateiendungen nicht vorgeschrieben. `file` versucht, Dateien anhand ihrer ersten Bytes (*Magic Bytes*) und einer Datenbank, in der solche Magic Numbers verzeichnet sind, zu erkennen. Gut ist auch ein Plausibilitätscheck mit `ls -l`, der verrät, dass die Datei *dateiname* ganze 51kB lang ist. Für eine kurze Plaintext-Datei mit wenigen Zeilen ist das unrealistisch, da das in einer ASCII-Kodierung ja bereits 51.000 Zeichen entsprechen.

4 Lochstreifen-Hardware

Eine vollständige Beschreibung der Hardware, die im Paper Tape Project verbaut wurde, soll hier nicht geschehen. Stattdessen soll ein grober Überblick über die verwendeten Architekturen gegeben werden, zusammen mit der Nennung von Gründen dafür.

4.1 Linux Parallelport-Treiber

Die meistgenutzte Hardware sind Stanzer und Leser mit 25-poligem D-Sub-Stecker, die aber nicht nach dem *Parallelport-Standard*, auch *Centronics-Standard* genannt, exakt *IEEE 1284*, arbeiten. Personalcomputer, und insbesondere derart alte, über die ich 2008 verfügte, hatten aber in den 90ern alle einen solchen Parallelport-Anschluss für Drucker. Mit einem geeigneten selbstgelötetem Kabel, welches die Ein/Ausgänge des Gerätes mit geeigneten Ein/Ausgängen des Computers verbindet, ist eine elektrische Verbindung möglich. Bloß spricht der Gerätetreiber, der im Kernel des Computers arbeitet, nicht das Protokoll des angeschlossenen Lochstreifengerätes.

Um ein solches Gerät also am Computer zu betreiben, benötigt es einem eigenen *Kernel*-Treiber, da in modernen Betriebssystemen nur der Kernel mit der Hardware direkt kommunizieren darf. Ein solcher Treiber läuft allerdings auch immer der Gefahr, den Computer zum Absturz zu bringen, wenn er nicht richtig programmiert ist, was ein »normales« Programm im *User Space* nicht kann. Unter Linux bot sich nun die

Programmierschnittstelle `ppdev` an, ein Kernelmodul, welches es erlaubt, die Pins des Parallelports des Computers aus dem Userspace heraus zu steuern. Also konnte ich einen *Parallelport Userspace-Driver* (in der Programmiersprache C) schreiben, der mit *ioctl*-Kernelaufrufen die Kommunikation zum Gerät steuerte.

Die Vorteile dieser Lösung waren einfaches Debugging und einfaches Programmieren. Nachteilig ist der häufige *Kontextwechsel* zwischen Kernel und User Space, der sehr langsam vonstatten geht. Dass man mit diesem Modell nicht interruptbasiert programmieren konnte, war mir damals gar nicht bewusst.

Ob der Treiber richtig funktioniert, hängt damit sehr stark von der Leistung des Computers, dem was er nebenbei sonst noch macht, und den eingestellten Warte/Pollingzeiten ab. Es ist sozusagen purer Zufall, dass diese Treiber so gut gingen.

Dafür ist die Benutzung denkbar einfach: Das Leseprogramm gibt direkt die gelesenen Daten aus, das Stanzprogramm erwartet sie auf der Standardeingabe. Mit Piping, wie oben diskutiert, geht damit alles.

4.2 Pradigmenwechsel über die Jahre

Mit zunehmenden Geräten zeigte sich, dass der Parallelport vielen I/O-Pins und hohen Geschwindigkeiten nicht gewachsen ist. Für neue Projekte wurde auf AVR-ATMega-Microcontroller gesetzt, vor allem im Punch-Card-Project. Dies lagert den »Gerätetreiber« auf ein *echtzeitfähiges* im Sinne eines deterministischen System-on-a-chips aus, der dann über den wohldefinierten Industriestandard *RS232*, die serielle Schnittstelle, mit dem Computer kommuniziert. Bewusst entschieden wir uns gegen USB direkt vom Microcontroller zum PC, da RS232 als reiner bidirektionaler Datenstream eben gemäß obiger Pradigmen eine universelle Schnittstelle darstellt und völlig betriebssystemunabhängig ist, wohingegen es für einen speziellen USB-Chip womöglich wieder keine Treiber für Linux geben könnte.

Mit diesem Schritt ist die Benutzung der Geräte von Windows aus tatsächlich möglich geworden. Die Parallelport-Treiber wurden zwar auch mal unter Windows entwickelt, jedoch mit wenig Bequemlichkeit, was die Konsolentools angeht (siehe folgende Kapitel).

4.3 Richtung einer Architekturunabhängigkeit

Bidirektionale Streams können prinzipiell beliebig von RS232 etwa über TCP netzwerktransparent weitergeleitet werden. Der Clienttreiber auf dem Computer, um per RS232 zu kommunizieren, kann eventbasiert unter beliebigen Frameworks implementiert werden. Leicht denkbar werden dabei Modelle, in denen die Lochstreifengeräte vom Smartphone aus bedienbar werden, über HTML5-Schnittstellen oder gar einen Browser-Client mit Websockets.

Allerdings muss man sich dabei immer überlegen, für welchen Einsatzzweck man Anwendungen baut. Diese Frage mag beim Lesen des folgenden Kapitels im Hinterkopf behalten werden.

5 Tools und Kommandos des Paper Tape Projects

Zu Beginn des Paper Tape Projects wurden einige Hilfsscripte in der Programmiersprache *Perl* geschrieben. Sie ermöglichen das bequeme Arbeiten mit Lochstreifen als Dateien.

5.1 Repräsentation von Lochstreifen auf dem Terminal

Per Konvention stelle ich einen Lochstreifen auf dem Terminal so dar, dass jede Zeile ein Byte repräsentiert, mit gesetzten oder nicht-gesetzten Bits:

```
| **.  ***|
|***.  ***|
|   .* ***|
```

Zu sehen ist, von links: Linker Rand des Lochstreifens (angedeutet), dann Stellen 2^0, 2^1, 2^2, das Führungsloch (immer da), anschließend 2^3 bis 2^7. Die Angaben 2^i verstehen sich so, dass die Byte-Repräsentation für den Wert z dezimal ergibt:

$$z = \sum_{i=0}^{7} 2^i \cdot \begin{cases} 1, & \text{wenn } * \\ 0, & \text{sonst} \end{cases}$$

Es gibt Konverter, um diese Form aus Binärdaten zu erzeugen:

```
$ cat binfile | punch-simulator
```

Außerdem gibt es konventionell, wenn auch überflüssig, einen Konverter, der aus dieser Darstellung wieder eine Binärdatei macht

```
$ cat binfile | punch-simulator | depuncher > binfile-copy
```

5.2 Debuggen

Zur Verarbeitung braucht man immer wieder Testdaten und Auslesewerkzeug. Dazu eignen sich:

```
$ hexmeter | punch-simulator | head -6
|   .     |
|*  .     |
```

```
| * .     |
|** .     |
|  *.     |
|* *.     |
```

`hexmeter` gibt ein »Hexadezimal-Band« aus, welches alle Belochungen von 0 (kein Loch) bis 255 (alle Löcher) enthält. Im Beispiel wird das mit `punch-simulator` im Terminal dargestellt, `head` ist ein Befehl um die Ausgabe nach in dem Fall 6 Zeilen abzuschneiden.

Mittels `hexmeter 4` gibt es vier mal diese Folge, allgemein `hexmeter` N, Defaultwert $N = 1$ wenn weggelassen. Für Details sollte man sich den Sourcecode anschauen.

Für noch akribischeres Zusammenbasteln gibt es

```
$ byte-tester 4 | punch-simulator
|  *.     |
```

Dieses Kommando erzeugt einfach ein Byte mit dem entsprechenden Dezimalwert.

Um zu schauen, wie lang so ein Lochstreifen wird, eignet sich

```
$ hex-meter | how-long-is-this-papertape
stdin: 256 bytes = 65.024 cm
$ how-long-is-this-papertape /etc/beispielfile
beispielfile: 1254536 bytes = 318652.144 cm
```

Hiermit kann man schnell sehen, wie lange man warten müssen wird. Der 318 Meter lange Lochstreifen der 1MB großen Datei füllt einen ganzen Raum.

5.3 Parser

Ein Umwandler für Formate, vgl. Abschnitt 3.4, ist schnell geschrieben. Die alten Udo-Parser sind Perl-Oneliner und kompilieren obiges Hallo-Welt-Beispiel zu:

```
$ udo-parser "68 61 6c 6c 6f 20 77 65 6c 74"
hallo welt
```

Udo steht übrigens nicht für *Universal decimal octettnotation*.

5.4 Stripping und Streammanipulation

Lochstreifen benötigen vorne und hinten einige cm leeres Band (Nullbytes). Diese sind auf dem Computer aber oft unerwünscht, da sie das Vergleichen von Dateien erschweren. Daher gibt es Tools, um eingelesene Lochstreifen von Nullbytes zu entfernen oder, im Gegenteil, solche Nullbytes im Stream zu erzeugen.

```
$ cat viel-whitespace-datei | strip-null-bytes > gestrippt
```

Das Erzeugen von Nullbytes um Daten herum geht mit

```
$ cat gestrippt | padd-null-bytes > null-bytes-herum.bin
```

Der Syntax ist `padd-null-bytes +`N_s`+`N_e, wobei N_s die Anzahl Nullbytes am Anfang und N_e die Anzahl Nullbytes am Ende ist. Per Default ist $N_s = N_e = 10$. Alternativ geht auch `padd-null-bytes` N, dann ist $N = N_s = N_e$.

In dem Zusammenhang möchte ich auch ein paar fortgeschrittene Streaming-Tricks nennen. `cat` steht für *concatenate* und schreibt in die Ausgabe Dateien aneinandergehängt:

```
$ cat datei1 datei2 datei3 | punch-simulator
```

Man kann aber auch Subshells verwenden, um z.B. jede Datei einzeln mit Nullbytes zu wrappen und so viele Dateien in einem Rutsch zu stanzen (die Einrückung im Listing ist Willkür):

```
$ { cat datei1 | padd-null-bytes;
    cat datei2 | padd-null-bytes; } | punch-simulator
```

5.5 Visualisierung

In Abschnitt 5.1 wurde eine Standardform zur Darstellung binärer Lochstreifendaten im Terminal eingeführt. Diese eignet sich zwar zum ersten Debuggen, ist aber nicht besonders praktisch, da sie nur wenig Ähnlichkeit mit einem Lochstreifen hat. Gerade auf dem Bildschirm bietet es sich aber an, einen Lochstreifen in Originalgröße zu zeichnen, auf den man einen ausgestanzten Streifen dann auflegen kann.

Dafür wurden bereits 2008 Programme entwickelt, aus Binärdaten hübsche Vektorgrafiken zeichnen. Zunächst in der Programmiersprache C mit der Grafikbibliothek *Cairo*, wurde später eine grafische Benutzeroberfläche mit dem *Gtk+*-Toolkit hinzugefügt, die später durch eine Benutzeroberfläche in C++ mit *Gtkmm* ersetzt wurde.

Die Visualisierung steht in zwei Varianten zur Verfügung: Entweder erzeugt sie direkt ein Bild als Datei, oder öffnet sich als interaktives, alleinstehendes Programm. Der Bilderzeuger heißt `tape2picture`:

```
$ hex-meter | tape2picture -f PNG -i pixelgrafik.png
$ hex-meter | tape2picture -f SVG -i vektorgrafik.svg
```

Das Programm bietet einige Parameter. Im Detail sind das Skalierungsmöglichkeiten, Paddings, Farben aller Komponenten sowie Transformationen/Rotationen:

```
$ tape2picture --help
Usage: tape2picture [OPTION...] [FILE TO READ FROM]
This program uses cairo to draw a punched tape as a PNG or SVG file. Any binary
data are accepted via stdin or read in from the file given as argument. The
```

```
produced image is written into stdout or in the filename given by -o.

  -f, --format=SVG|PNG       Set desired output image format (PNG or SVG)
  -o, -i, --output=FILE, --image=FILE
                             Output to FILE (instead of standard output)
  -v, --verbose              Produce verbose output on stderr

  Dimensions                 Dimensions are integers without units
  -d, --diameter=NUM         Set dimensions for output image by punched hole
                             diameter (pixel)
  -h, --height=NUM           Set desired height for output image (unit like
                             width argument)
  -w, --width=NUM            Set desired width for output image (in px or
                             points, according to output format)

  Empty bytes                Bytes with value=0x00 which are not content of
                             files
      --empty-end=NUM        Set number of empty bytes at end (default=0)
      --empty-start=NUM      Set number of empty bytes at beginning (default=0)


  Colors                     Color format: #RGB[A], #RRGGBB[AA]
      --color-feedholes=#RGBA   Set color of feed holes (the small ones)
      --color-imagebg=#RGBA  Set image background color
      --color-notpunched=#RGBA   Set color of bits with boolean value "false"
      --color-punched=#RGBA  Set color of holes (punched bits)
      --color-tapebg=#RGBA   Set tape background color
      --hide-feedholes       Hide the feed holes (which means they wouldnt be
                             punched)
      --hide-imagebg         Make the image background (space around paper
                             tape) transparent
      --hide-notpunched      Hide the holes which arent punched on real tapes
      --hide-punched         Hide the holes (only the punched ones)
      --hide-tapebg          Hide the paper tape background

  Transformations and Rotations
      --reflection-bit-direction   Enables vertical reflection on the other
                             axis (inverts bit direction)
      --reflection-byte-direction
                             Enables horizontal reflecion on the data axis
                             (inverts data direction)
      --rotation=right|bottom|left|top
                             Rotation of punched tape (alternative short
                             notation: 0=right, 1=bottom, 2=left, 3=top)

  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

Das folgende Kommando lässt sich vom Zufallsstrom der Datei `/dev/random` die ersten 70 Byte geben und plottet sie mit einem Lochdurchmesser von 15 Pixeln. Das resultierende Bild wird direkt mittels *ImageMagick* angezeigt:

```
$ cat /dev/random | head -c770 | tape2picture -d15 | display
```

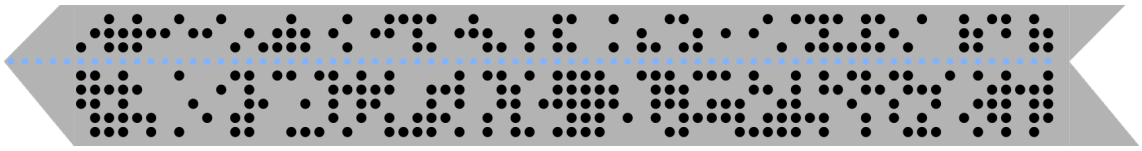

Abbildung 2: Das generierte Bild

5.6 Beschriftung

Lochstreifenbeschriftung ist eine Spielerei, die auch einen Nutzen hat. Absicht ist, in dem durch die Lochung des Lochstreifens entstehenden Bild einen Text abzubilden. Dafür hab ich verschiedene *Schriften* entwickelt, die als Perl-Programme implementiert sind.

```
label-it "hallo welt" | tape2picture -d10 > hallo-welt-alt.png
% Drehung noch mit: convert -flip foo.png foo.png
```

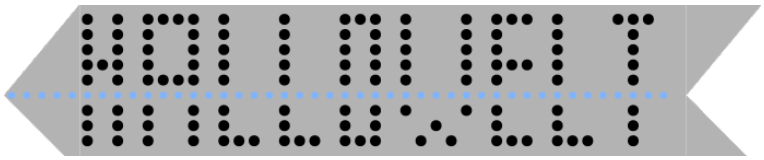

Abbildung 3: *Hallo Welt* in der ersten Lochstreifenschrift

Die »neue« bzw. zweite gemachte Lochstreifenschrift erwartet die Eingaben auf der Standardeingabe:

```
$ echo "hallo welt" | beschriftung | tape2picture -d10 > hallo-welt.png
```

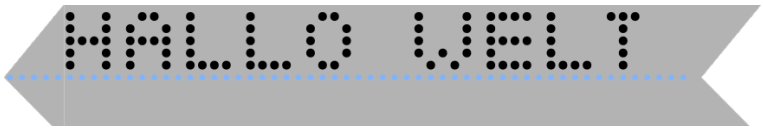

Abbildung 4: *Hallo Welt* in der zweiten Lochstreifenschrift

Mit diesen Beschriftungen lässt sich beliebig rumspielen, um sie etwa vor Nutzdaten mit ausreichend Abstand zu packen.

Desweiteren gab es eine 1000-Zeilen lange C-Code-Implementierung eines Parsers für Schriften in einem Austauschformat, was dem aus Abschnitt 5.1 nahekommt. Der kam aber nie zum Einsatz, sieh dafür auch weiter unten. Für den Einsatz brauchbar sind tatsächlich nur diese Schriften-Executables, die an dieser Stelle vorgestellt wurden.

5.7 Ein- und Auslesen von Lochstreifen

Zum letztlichen physischen Einlesen und Stanzen von Lochstreifen gibt es den `tape-reader`:

```
$ tape-reader > eingelesene-datei.txt
```

Er gibt die eingelesenen Daten auf der Standardausgabe aus und gibt gleichzeitig auf der Standardfehler-Ausgabe eine formatierte Ansicht des Lochstreifens aus, die Abschnitt 5.1 folgt.

In gleicher Weise arbeitet der Stanzer:

```
$ cat stanzdatei | tape-puncher
FACIT 4070 Tape punch-75 CPS userspace driver
time ms |123.45678|  hex=dec status: =) punch successful / :-( still busy
....
```

Hier sehen wir auch in der zweiten Zeile eine Legende der Ausgaben, die dann folgen. Der Stanzer kann auf dem Terminal auch eine Art Fortschrittsangabe machen, wenn er weiß wie viele Daten ihn erwarten. Das kann übergeben werden mit `tape-puncher -l` *n*, wobei *n* eine Ganzzahl ist.

Bequemer geht das mit dem *Punch-Frontend*, einem kleinen Perlscript, welches ich mal geschrieben habe, um Label-Generierung, Nullbytes und Punching in einem Schritt zu machen. Im Prinzip macht es soetwas wie folgender Pseudocode:

```
$ { ask_for_label; print label; print file; } | tape-puncher
```

Aufgerufen wird es mittels

```
$ puncher-frontend dateiname.txt
```

Es fragt interaktiv nach einem Label, welches man eingibt und mit Entertaste (Newline) bestätigt. Dann wird die Datei gestanzt, die als Argument übergeben wurde.

6 Einzelne Aspekte

Im Anschluss sollen nun noch einzelne Themen des Paper Tape Projects beleuchtet werden.

6.1 Die PaperTapeSuite

Nachdem sich auf Linux 2008 schnell ein gesundes Ökosystem gebildet hat, mit dem Lochstreifen gestanzt werden können, kam die Frage auf, wie man *möglichst einfach* Stanzen und Lesen könne. Das ist equivalent zu der Frage nach einer möglichst einfachen *grafischen Benutzeroberfläche*, die also per Maus bedient werden kann. Dass diese

Benutzeroberfläche unter Windows laufen muss, stellt nicht nur an die Treiberentwicklung, sondern auch die Programmierung der Oberfläche große Herausforderungen, da ein möglichst plattformunabhängiges Toolkit vonnöten ist.

Ein solches Werkzeug sollte die PaperTapeSuite werden. Angefangen als Visualisierungswerkzeug, wie in Abschnitt 5.5 dargestellt, ist die Software über Jahre gewachsen, mit dem Anspruch, ein vollwertiger Ersatz für alle Terminalanwendungszenarien zu werden, die oben beschrieben sind.

Dies umfasste neben dem Öffnen von Dateien, die als Lochstreifen dargestellt wurden, auch einen *integrierten Editor*, der einen Hexadezimaleditor, wie in Abschnitt 3.1 vorgestellt, ersetzen sollte. Augenmerk wurde dabei insbesondere darauf gelegt, dass einzelne Bits mit der Maus umgedreht werden können sowie Auswahlbereiche, Cursorverhalten, Drag & Drop, Copy & Paste möglich sind. Auch ein *Multi Document Interface* war angedacht (Öffnen von mehreren Dokumenten gleichzeitig im gleichen Programm), bzw. mehrere Views des gleichen Lochstreifen. Alle Einstellmöglichkeiten des *Lochstreifen*-Widgets sollten grafisch repräsentiert werden, etwa durch Farbwähler und ähnliches.

Darüberhinaus sollte selbstverständlich die Lese- und Stanzprozedur mit einer grafischen Benutzeroberfläche in das Programm eingebaut werden.

Quasi unlösbar wurde die Aufgabe auch durch die Wahl der Mittel; mit C und Gtk+ befindet man sich derart Lowlevel, dass bereits das Neuzeichnen des Widgets zur Geduldsprobe wird, muss man sich doch zwischen Gdk, Gtk und Cairo bewegen, und das mit Clipping-Zeichenroutinen und Export der Ansicht in PNG und SVG.

Letztlich ist das Projekt 2009 nach über 6.000 Codezeilen eingeschlafen, weil es unsäglich viel Zeit gefressen hat.

6.2 Paper Tape Projects NG

Next Generation war im Jahr 2012 das Schlagwort für ein Wiederaufleben des Lochstreifenprojektes, nachdem es scheinbar in der völligen Unbenutzbarkeit untergegangen ist. Gerade die Installation auf diversen alten Computern lief dem Produktiveinsatz entgegen. Um 2010/2011 herum war auch mal der Aufbau einer *Punch-Station* angedacht, also einem stationären Computer, um den herum alles aufgebaut ist. Der hat im Gartenhaus dann auch mal einige Jahre recht unbeachtet vom Zielpublikum gestanden.

Kernziel der NG-Projects ist die Migration auf RS232, um damit per USB an Laptops ranzukommen und sich von den Altcomputern mit LPT-Schnittstelle verabschieden zu können. Der Fortschritt dieses Vorhabens ist nicht schlecht, allerdings stellt sich auch die Frage, wie brauchbar Lochstreifen-Werkzeuge sind, wenn man nur eingeschränkte Werkzeuge zur Verfügung hat wie unter Windows.

Letztlich kam es in diesem Zusammenhang auch zum Wiederaufleben der alten Programme und einer Reinstandsetzung der Tools, die in diesem Dokument mündete.

7 Quellen und weiterführende Links

[1] technikum29-Entwicklungsprojekte
http://labs.technikum29.de/

[2] In-Code-Documentation, siehe Checkout.
http://labs.technikum29.de/checkout/paper-tape-project/trunk/documentation/

[3] Edgewall Trac
http://trac.edgewall.org/

[4] Trac-Installation vom technikum29
http://labs.technikum29.de/wiki

[5] Bash-Redirection Syntax und illustriertes Tutorial in der Bash-Hackers-Wiki
http://wiki.bash-hackers.org/syntax/redirection
http://wiki.bash-hackers.org/howto/redirection_tutorial

[6] Unix-Philosophie, aus Wikipedia
http://de.wikipedia.org/wiki/Unix-Philosophie

[7] Zahlendarstellungen in der Informatik, zu finden etwa in Wikipedia und Scripten der technischen Informatik, beispielsweise
http://de.wikipedia.org/wiki/Integer_%28Datentyp%29
http://www.es.cs.uni-frankfurt.de/index.php?id=203

[8] Okteta und Konsorten, Hexeditoren.
http://www.kde.org/applications/utilities/okteta/
Bildquelle:
http://www.dedoimedo.com/computers/linux-data-recovery.html

[9] Eine hübsche Darstellung, was unterschiedliche Zeilenenden bewirken können, beinhaltet auch der Wikipedia-Artikel zur *Shebang*
http://de.wikipedia.org/wiki/Shebang
http://de.wikipedia.org/wiki/Zeilenumbruch

[10] *Notepad++* unter Windows kann beispielsweise prima mit Zeichensätzen und der BOM umgehen
http://notepad-plus-plus.org/