technikum29-Entwicklungsprojekte

Punch Card Project
(Open Source)

Lochkartenverarbeitung per Computer

Anbindung eines Documation M200 Lochkartenlesers per Mikrocontroller über RS232 an zeitgenössische PC-Architekturen, Visualisierung und Verarbeitung mit einem plattformunabhängigem grafischen Editorprogramm als Hybrid zwischen Hex-Editor und Texteditor.

Sven Köppel
Februar 2010

Inhaltsverzeichnis

Einleitung

Dieses Projekt durchkreuzt in mehrfacher Hinsicht verschiedene Abstraktionsstufen: Sowohl von der Hardwarekomplexität, die von dem elektronisch-pneumatisch betriebenen Lochkartenlesegerät, über den integrierten Mikrocontroller (System-on-a-Chip) bis hin zur ausgewachsenen superskalaren x64er-Mehrkernarchitektur des 21. Jahrhunderts reicht, als auch von der Anwendungslogik, die von der rohen quasi bit-adressierten Speicherstruktur bis hin zum objektorientiertem Multi-Threading mit anspruchsvoller grafischer Benutzeroberfläche reicht.

Namentlich läuft das Projekt im Rahmen des *Punch Card Projects*, was nicht nur zufällig wie das *Paper Tape Project* von 2008 klingt, in dessen Rahmen ich Treiber für mehrere Lochstreifenstanzer und -leser für verschiedene Betriebssysteme geschrieben habe, außerdem Libraries zur Low-Level-Verarbeitung in C und Konsorten, und natürlich den Lochstreifenvisualisierer.

Letzterer ist vom ursprünglich einfachen Spaßprojekt (Lochstreifen nicht nur als ASCII-Art, sondern als richtige Bilder zu zeichnen) zu einem umfangreichen Monster mutiert, welches mehrere Entwicklungsstufen hinter sich hatte und mehrfach umgebaut wurde. Es wurde ein Erwartungshorizont gestellt, der niemals erreicht werden konnte und dermaßen oft neu entwickelt, dass in der letzten Revision nicht viel funktionsfähiges übrig geblieben ist (wenn man von den letzten funktionsfähigen Zuständen der jeweiligen Entwicklungsstadien kurz vor der Neuentwicklung absieht).

Im Vergleich zeugt das *Punch Card Project* von unheimlicher Professionalität. Das Programm für den Microcontroller entstammt einer kontinuierlichen Entwicklungslinie und hat zuguterletzt zwar einige grundlegende Designschwächen, läuft aber ansonsten völlig stabil und fehlerfrei.

Der *Punch Card Editor* wurde, bevor die erste Zeile programmiert wurde, sehr detailiert entworfen. Bereits im Vorfeld wurden Sprache und Toolkit wohlüberlegt gewählt. Ich habe mehrmals mit dem Kartenmodell angefangen, ehe sich eine ideale Konstellation ergab, und das restliche Programm ist in einer ähnlich kontinuierlichen Entwicklung entstanden. Der *Punch Card Editor* enthält nun Funktionen, von denen der *Paper Tape Editor* nur träumen konnte – nur sind sie mit wesentlich weniger Aufwand entstanden. Auch hier ist eine Professionalisierung der Arbeitsabläufe zu betrachten gewesen, wobei die Erfolge sicherlich auch Qt zuzuschreiben sind.

Allerdings birgt ein derart mächtiges Toolkit wie Qt die Gefahr, dass man immer neue Funktionen entdeckt, deren Einsatz sich dann geradezu aufdrängt – das Undo-Framework war etwas derartiges: Eine „Funktion“, die ich schon immer mal in einem grafischen Programm einbauen wollte, was dank Qt auch mit wenig Aufwand gelang, aber zur einwandfreien Umsetzung doch eigentlich noch einiges an Arbeit bedürfte.

Um aber ein gesundes Verhältnis zwischen Arbeitsaufwand und Ergebnis zu erzielen, kann es nicht erstrebenswert sein, sämtliche dieser Funktionen weiter zu verfolgen. Das vorliegende Dokument gibt eine Übersicht über den Umfang und die Funktionen, die dieses Projekt bereits erreicht hat und wird anschließend eine Diskussion liefern, welche Komponenten ausgebaut werden müssen, wo demnach noch Entwicklungsbedarf entsteht, und welche Äste abgeschnitten werden dürfen.

Sven Köppel,
im Februar 2010

1. Der Mikrocontroller

Verwendet wurde ein Mikrocontroller aus der ATmega-Reihe von AVR, namentlich Atmega 644. Er verfügt über 4kB RAM und wurde mit C sowie der avr-libc[1] programmiert. Bei letzterer handelt es sich um eine Version der „Standard C Library“, eine Programmbibliothek, die Funktionen zur Ein- und Ausgabe, Verarbeitung von Zeichenketten (Strings), Speicherverwaltung, mathematische Funktionen, uvm. zur Verfügung stellt.

Die Quelltexte bestehen aus nur wenigen Dateien, an denen man bereits die Struktur des Programmes erkennt:

Dateiname	*# Zeilen*	*Zusammenfassung*
`main.c`	140	Initialisierung aller Schnittstellen
`punchcard.h`	170	Kartenmodell, größtenteils Strukturdefinitionen und inline-Funktionen (Performance!)
`punchcard.c`	100	
`driver.c`	250	Treiber zum Lochkartenleser, größtenteils Interruptvektor-implementierungen
`driver.h`	70	
`protocol.c`	250	Protokoll und Implementierung zur Benutzerkommunikation (RS232)
`protocol.h`	70	
`wiring.h`	100	Definitionen zur Kabelbelegung
`pc-uc-protocol.h`	100	Definitionen zum Protokoll zur Benutzerkommunikation

In der Summe sind dies gerade einmal 1300 Zeilen C-Quellcode.

Offensichtlich lässt sich das Programm in einzelne Teile zerlegen, und zwar der Kartenmodellierung, dem Gerätetreiber und der Computeranbindung (Protokoll-Implementierung). Diese Teile werden im folgenden dargestellt.

1 http://www.nongnu.org/avr-libc/

1.1. Kartenmodell

Das für den Mikrocontroller programmierte Kartenmodell (ein weiteres, grundsätzlich anderes existiert auch auf der Computerseite) ist mit Hauptaugenmerk auf Geschwindigkeit und wenig RAM-Verbrauch programmiert. Damit sind die Rahmenbedingungen bereits festgelegt: Nutzung nativer Speicherstrukturen und Inlinefunktionen statt etwa objektorientierter Datenkapselung mit schwergewichtigen Klassen (Strukturen).

In diesem Kartenmodell wird eine Lochkartenspalte (12 Löcher = 12 Binärziffern) in 16 Bit kodiert. Dies trifft sich hervorragend, da der verwendete Mikrocontroller eine 16bit-Architektur aufweist und die vier ungenutzten Bit damit keinen Verlust in der Verarbeitungsgeschwindigkeit darstellen. Die Definition einer Spalte ist damit nichts anderes als ein typedef von einem vorzeichenlosen 16bit-Ganzzahlentyp (uint16_t). Um eine eindeutige Abbildung der Lochkartenspaltenpositionen auf die Bitpositionen in dem 16Bit-Speicher zu gewährleisten, existieren die Makros COL0 bis COL12 (dabei ohne COL10, welches auf Lochkarten nicht existiert), die jeweils Indexpositionen im 16Bit-Speicher entsprechen.

Auf der Spaltenstruktur aufbauend wird eine Lochkarte als Struktur modelliert, die ein Array der besagten Spaltenstrukturen speichert (Länge: 80 Spalten) sowie einen schmalen 8Bit-Integer, der das aktuelle Schreib-Offset speichert und gleichzeitig als Indikator über den Zustand dieser Lochkarte dient (wurde sie bereits komplett eingelesen?). Zur eindeutigen Interpretation spezieller Zahlenwerte wurden die „Magic Values“ `CARD_EMPTY`, `CARD_READY` und `CARD_DONE` definiert, die aussagen, ob die Karte leer ist (Offset bei Position 0, Standard), also noch keine Spalte beschrieben wurde, oder ob die Karte voll ist (Offset bei Position 80), oder ob die Karte bereits an den Benutzer ausgeschrieben wurde und die komplette Struktur gelöscht bzw. überschrieben werden kann.

Exzessive Verwendung dieser Zustandsvariablen wird von dem globalen Kartenpuffer gemacht, welcher im Folgenden im globalen Adressraum definiert wird. Es handelt sich dabei prinzipiell um einen Ringpuffer von Lochkarten, der dafür gedacht ist, die vom Lochkartenleser reinkommenden Lochkarten gegenüber der Ausgabe über RS232 an den Benutzer abzupuffern. Entsprechend besteht die Struktur des Kartenpuffers klassischerweise aus einem Array von Karten sowie einem Schreib- und Leseindex. Die Länge des Ringpuffers ist prinzipiell beliebig und nur durch den zur Verfügung stehenden RAM begrenzt. Eine einfache Überschlagsrechnung (eine Lochkarte benötigt etwa etwa 160 Bytes, wir brauchen noch Speicher für die Ein/Ausgaben sowie den Funktionsstack, ein Heap wird nicht benutzt) führt zu einer Länge von etwa 4 Karten.

In der Headerfile `punchcard.h` folgen zahlreiche Makros, um mit dem Kartenpuffer umzugehen, seinen Zustand zu überprüfen, usw.. Ohne näher ins Detail zu gehen, möchte ich seine Funktion darstellen, da ihm beim Gesamtprogramm eine zentrale Rolle zukommt.

1.1.1. Globaler Kartenringpuffer

Ringpuffer arbeiten immer nach der gleichen Weise, der englischsprachige Wikipedia-Artikel „Circular buffer“ stellt dies hervorragend dar, inklusive einer Referenzimplementierung in C, an der sich meine Implementierung durchaus orientiert hat.

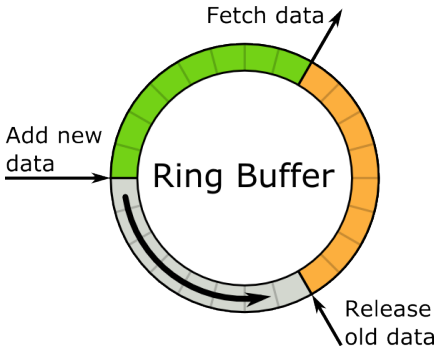


Abbildung 1.1: Funktionsweise eines Ringpuffers

Mein Ringpuffer überschreibt keine alten Daten, wenn sie noch nicht gelesen wurden, da sonst freilich Informationen

verloren gingen. Da es aber essentiell ist, alle Lochkarten in ihrer Reihenfolge lückenlos zu erfassen, wird der Mikrocontroller den Leseprozess im Falle eines vollen Puffers unverzüglich anhalten. (Dass dies oft nicht mehr möglich ist, da das Lesegerät bereits seinen nächsten Zyklus gestartet hat, steht wortwörtlich auf einem anderen Blatt).

Wieso benötigt man einen Puffer? Eigentlich könnte man eine Lochkarte doch instantan beim Bekommen der Daten vom Lesegerät an den Benutzer ausgeben. Das kleine Problem dabei ist nur, wenn die Ausgabe an den Benutzer länger dauert, als die neuen Daten reinkommen – dann tritt nämlich Datenverlust auf. Mithilfe eines Puffers wird die „kleinste Übertragungseinheit“ von der Lochkartenspalte zur gesamten Lochkarte angehoben (wie wir im späteren Kapitel lesen werden) und damit die Integrität einer Lochkarte als solche garantiert.

1.2. Kommunikation zum Lochkartenleser

Die eigentlich interessante Aufgabe des Mikrocontrollers besteht in der Kommunikation mit dem Lochkartengerät. Diese Aufgabe wird interrupt-gesteuert und damit asynchron zum eigentlichen Programmablauf erledigt.

1.2.1. Pinbelegung und -Verdrahtung

Essentiell zur Diskussion über die Funktionsweise ist dabei eine Übersicht der Anschlüsse, mit denen wir arbeiten. Zunächst sind da natürlich die 12 Datenleitungen, auf denen eine Spalte parallel reinkommt. Der Mikrocontroller verfügt über 32 frei wählbare Digitalein/ausgänge, also vier 8-Bit-Registern. Die 12 Datenbits wurden dementsprechend auf eineinhalb Register verteilt, wobei die genaue Pinbelegung in *wiring.h* definiert ist.

Als Statussignale (Ausgänge aus dem Lesegerät, Eingänge zum Mikrocontroller) bietet der Lochkartenleser des weiteren vier Fehlerleitungen an, außerdem `BUSY`, `READY` sowie eine sogenannte `INDEX MARK` (`IM`).

Als einziges Steuersignal (Eingänge zum Lesegerät, Ausgänge vom Mikrocontroller) gibt es das `PICK COMMAND` (`PC`).

Die Funktionsweise ist der Kommunikation ist denkbar einfach: In dem der Mikrocontroller das `PC` auf `HIGH` setzt, weist er den Leser an, das Lesen zu starten. Die interessante Eingangsgröße ist nun die `IM`, welche beim Anliegen der 12 Datenbits auf HIGH geht und demnach über ein flankengesteuertes Interrupt abgefangen werden kann.

1.2.2. Gerätetreiber

driver.c besteht demnach nur aus einigen Funktionsrümpfen zur Belegung von Interruptvektoren, d.h. letztlich nur aus Funktionen, die bei bestimmten Ereignissen aufgerufen werden. Insbesondere macht sich das Programm großen Nutzen einer besonderen Form von Interrupts, die der Mikrocontroller zur Verfügung stellt: Register Pin toggle Interrupts, also Interrupts, die abgefeuert werden können, wenn sich irgendein Bit eines Registers ändert.

Das Problem bei dieser Technik ist, dass man nicht weiß, welches der acht Bit sich verändert hat und wie es sich verändert hat (steigende oder fallende Flanke). Dementsprechend muss jeweils einen Zwischenspeicher für einen „vorherigen“ Zustand des Registers gepflegt werden, den man bei Aufruf der Interrupt-Service-Routine (`ISR`) mit dem aktuellen Registerzustand XORt. Der Nachteil dieser Methode ist, dass es sehr lange dauert, bis man denn rausgefunden hat, was sich wie verändert hat. Das ist insbesondere auf die Verwendung der Hochsprache C zurückzuführen, da beim Sprung in die ISR natürlich – wie bei jedem Funktionsaufruf – die Register gerettet werden.

Da aber eben einige Arithmetik notwendig ist, müssen besonders viele Register gerettet werden (40 Stück), und jeder Push auf den Stack benötigt zwei Zyklen. Legt man nun eine langsame Betriebszeit von 8Mhz zugrunde, kann man sich ausrechnen, wie lang die Impulse mindestens sein müssen, damit man sie mit diesen lahmen ISRs noch erwischen kann (es geht in den Millisekundenbereich).

Kurzum: Es musste tunlichst darauf geachtet werden, kurzlebige Signale (`IM`) auf die ausgezeichneten Interruptfähigen Pins zu legen, die hardwaremäßig auf bestimmte Ereignisse (steigende Flanke) getriggert werden können.

Lange Rede, kurzer Sinn, was macht letztlich der Gerätetreiber? Viel Aufwand wird betrieben, möglichst komplett imstande zu sein, was im Lesegerät vor sich geht. Daher werden sämtliche Fehler und `BUSY`/`READY`-Flanken betrachtet und im Debug-Modus auch an den Benutzer ausgegeben (was aufgrund der Nebenläufigkeit einige Komplikationen birgt). Wirklich interessant für die Funktion ist aber nur die Behandlung der IM. Sie ist wirklich sehr einfach: Es wird kurz überprüft, ob im Puffer noch Platz ist (wenn nicht, ist der kritische Zustand eingetreten, dass die nun reinkommenden Daten nicht mehr abgespeichert werden können), anschließend werden die anliegenden Daten an die passende Stelle im Ringpuffer abgespeichert.

1.3. Kommunikation zum Computer

Ein Problem, was man bei der Verwendung von Mikrocontrollern zur Ansteuerung von Hardware durch einen Computer leicht vergessen kann, ist, dass man durch den Mikrocontroller eine weitere Abstraktionsschicht einbaut, die untereinander kommunizieren muss, d.h. wir benötigen eine eindeutige Kommunikation zwischen Computer und Mikrocontroller.

1.3.1. Das PC Documation M200 µC Serial Communication Protocol

Um eine solche PC-µC-Kommunikation eindeutig zu definieren, schreibt man in der Informatik und Telekommunikation ein Protokoll, welches „eine Vereinbarung, nach der die Verbindung, Kommunikation und Datenübertragung zwischen zwei Parteien abläuft“[2], ist. Ein solches Protokoll habe ich (der Allgemeinheit wegen) in englischer Sprache verfasst, es ist knapp fünf Seiten lang und liegt diesem Dokument im Anhang bei.

Da das Protokoll nicht sonderlich technisch in Bezug auf die Kommunikation über RS232 ist, aber hingegen einige wichtige Begriffe definiert und die drei essentiellen Datenübertragungsformate sowie das von Douglas W. Jones vorgeschlagene Kodierungsformat für Lochkarten vorstellt, empfehle ich, das Protokoll zum Verständnis des folgenden Textes zu lesen.

(Unter anderem werden in ihm einige Begriffe eingeführt, die ich an dieser Stelle auch gerne verwenden möchte: „Server“ als Synonym für den Microcontroller und „Client“ als Synonym für den Benutzer an einer seriellen Konsole oder den Computer generell bzw. ein Anwendungsprogramm, welches auf ihm läuft und über die RS232-Schnittstelle mit dem Microcontroller kommuniziert.)

1.3.2. Implementierung des Protokolls

Auf dem Mikrocontroller spielt die Implementierung des Protokolls im Programmablauf die zentrale Rolle. Von der Eingangsroutine main() in der gleichnamigen Datei `main.c` wird ganz am Ende die `user_input_loop();` aufgerufen, und damit (naheliegenderweise) eine Endlosschleife, die auf Benutzereingaben über den RS232-Port wartet. Es war zwar eine Hardwareflusssteuerung

2 Wikipedia: „Protokoll (Informatik)“

geplant, mit deren Hilfe die Benutzerkommunikation auch asynchron ablaufen könnte, aber letztlich bringt sie hier keine großen Vorteile, da es keine Rolle spielt, ob der Mikrocontroller standardmäßig nops durchführt oder eben in der Benutzerkommunikationsschleife verharrt.

Die Implementierung bedient sich exzessiv den `stdio.h`-Komponenten der avr-libc, d.h. insbesondere der formatierten Ausgabe. Sie ist zwar sehr RAM-intensiv, das stört aber nicht weiter, vor allem, wenn man die praktischen Vorteile betrachtet.

Die Funktionsweise im Detail ist nicht sonderlich interessant, da ja vor allem die vielen Ausgaben viel Libraryaufrufe und Makroaufrufe in anderen Subsystemen erzeugen. Auf Performance wurde in diesem Abschnitt nicht besonders viel Wert gelegt, da ja nicht die Gefahr von Datenverlust besteht und es keine herausragende Rolle spielt, wie schnell die gepufferten Daten an den Client ausgegeben werden. Den meisten Einfluss auf die Geschwindigkeit hat ohnehin die eigentliche Funktion, die Lochkarten ausgibt. Freilich schlägt das Debugformat dabei so viel zu Buche, dass der Leser sogar einige Sekunden angehalten werden muss, weil der Puffer voll ist. Anschließend macht sich der Puffer derart bemerkbar, dass man am Client noch Ausgaben erhält, lange nachdem das Lesegerät nicht mehr arbeitet.

Hingegen ist die Binärkodierung nach Jones derart effizient, dass der Puffer im Grunde genommen unnötig ist, wenn die Übertragungsrate (in Baud) ausreichend hoch eingestellt ist.

Die Implementierung des Protokolls weist noch einige programmatorische Feinheiten auf, auf die hingewiesen werden soll: Zum einen sind alle Client-Anfrage-Codes sowie Server-Antwort-Codes in der C-Headerdatei `pc-uc-protocol.h` definiert, die von allen Implementierungen eingebunden werden soll und die durch einige Makros recht komfortabel einen Versionsvergleich zwischen Server und Client durchführen kann. Zum anderen wurde in der Implementierung des Protokolls wegen der vielen Strings exzessiver Gebrauch von *Program Memory* (`PROGMEM`) gemacht. Dies verlangsamt zwar die Ladezeit der Strings nochmals, allerdings zugunsten des RAM-Verbrauchs, der etwa bei der langen `help_message` enorm wäre (über 1.5kB).

1.4. Status des Teilbereiches

Für den Teilbereich „Mikrocontroller-Programmierung“ ist abschließend zu sagen, dass eigentlich kein Programmieraufwand mehr nötig ist. Das Programm ist zwar keineswegs perfekt, aber eine Optimierung wäre den Aufwand nicht wert. Das Programm weist etwa bei der Synchronisierung der Ausgabe an den Client erhebliche Schwächen auf (Aus der Interruptroutine wird etwa einfach in die Ausgaben geschrieben). Das kommt bei Debuggingausgaben, aber vor allem auch bei der Information, dass etwa `BUSY` oder `READY` sich verändert haben, vor. Dem wurde bereits mit einer Art Semaphore (`set_main_loop_printing`) entgegengewirkt, allerdings beginnt das gleiche Problem wieder, wenn die Interruptroutine von einer weiteren Interruptroutine unterbrochen wird, was durchaus vorkommt. Eine saubere Lösung würde mit einer Event Queue arbeiten, in die alle Nachrichten eingestellt werden und ihrer Priorität nach abgearbeitet werden würden. Dies würde aber eine Neuimplementierung bedürfen (man könnte übrigens Glib-Funktionen dafür verwenden), die sich aber wie gesagt nicht lohnt.

2. Das Computerprogramm

In der Programmiersprache C++ wurde mit der bekannten C++-Klassenbibliothek *Qt* ein einzigartiger Lochkarteneditor programmiert, der seinesgleichen sucht. Der Gesamtumfang beträgt ohne Zusatzbibliotheken derzeit etwa 4000 Zeilen (Stand Februar 2010).

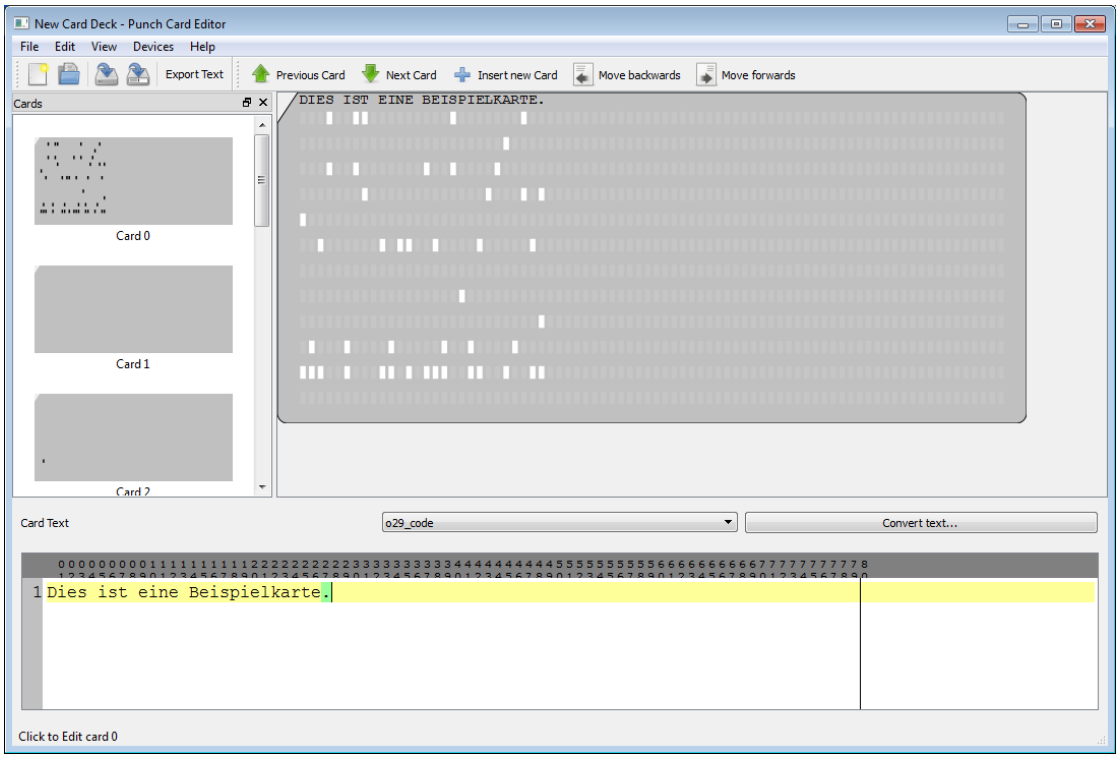


Abbildung 2.1: Exemplarischer Screenshot einer typischen Sitzung mit dem Punch Card Editor

2.1. Zielsetzung und Paradigmen

Es sollte eine grafische Anwendung geschrieben werden, mit der Lochkarten behandelt werden können, d.h. Ein- und Ausgabe von Lochkarten und eben die Anbindung von Gerätetreibern, um Lochkarten einzulesen und zu stanzen.

Aufbauend darauf, und in Anlehnung an die Ziele des *Paper Tape Projects*[3], hatte ich einen Lochkarteneditor vor Augen, der in erster Linie die Grundmetapher der grafischen Benutzeroberflächen seit ihrer Existenz verinnerlicht, und zwar die Schreibtischmetapher: Gib dem Benutzer des Programmes die Möglichkeiten, mit Lochkarten zu arbeiten, als seien sie echt, als lägen sie vor ihm auf seinem Schreibtisch.

Dementsprechend bedarf es primär der Visualisierung einer einzelnen Lochkarte, und dann vor allem dem Lochkartenstapel (*Deck*). Das Verschieben, Hinzufügen und Entfernen von Karten im Stapel soll nativ möglich sein, also per *Drag & Drop*. Wie bearbeitet man denn nun eine Lochkarte?

3 *The Paper Tape Project*, processing paper tapes with today's computers on common platforms like GNU/Linux and Win32: http://dev.technikum29.de/projects/paper-tape-project/

Für gewöhnlich war auf Lochkarten Text in einem speziellen Zeichensatz gestanzt. Das muss aber nicht der Fall sein. Da der Anspruch war, einen universellen Lochkarteneditor zu schreiben, war demnach das Ziel, mit jeglichen binären Lochkarten arbeiten zu können, also kann eine Lochkarte nur als binärer Datenspeicher behandelt werden. Binäre Daten werden allerdings heutzutage in der Regel mit Hex-Editoren bearbeitet. Damit haben wir bereits eine Kategorisierung des Programmes erreicht: Ein Hex-Editor mit graphischer Repräsentation der Dateien. Es war des öfteren in der Diskussion, mit einem Ressourcenmanager oder vergleichbarem die Manipulation von Lochkarten zu ermöglichen, dies wurde aber zugunsten einer komplett grafischen Bearbeitung per Maus auf der Lochkarte verworfen. So war es also angedacht, Lochkarten zu verändern, in dem man mit der Maus auf die Löcher in den Spalten klickt und diese damit umschaltet (*togglen*).

Damit ist im großen und ganzen das Leistungsspektrum abgedeckt, was das Paper-Tape-Project sich als Anspruch gestellt hat. Verworfen wurden dabei die völlig überzogenen Forderungen, Lochkarten beliebig drehen und verschieben zu können (Affine Abbildungstransformation in der Darstellung), außerdem erschienen auch die vielfältigen Konfigurationsmöglichkeiten in der Farbdarstellung den Aufwand nicht Wert. Hingegen ist ein Export als Bilddateien dank des Qt-Zeichenframeworks sehr einfach möglich und wurde auch vollständig implementiert.

Um der grafischen Benutzeroberfläche mehr Reiz zu verleihen und einen intuitiveren Umgang zu fördern, war zudem geplant, die Möglichkeiten des Qt-Toolkits auszunutzen. Dazu gehörte die Umsetzung eines Undo-Frameworks sowie ein ansprechendes Fensterlayout mit Docks[4].

2.2. Namensräume und Libraries

Für eine schnelle Übersicht über den doch recht umfangreichen Code lohnt sich ein kurzer Blick auf die Dateiordnung und die damit einhergehenden C++-Namensräume, von denen exzessiver Gebrauch gemacht wurde. Basisnamensraum ist `QPunchCard`.

Im Groben hat das `src`-Verzeichnis folgende Struktur:

- `app`, Namensraum: `QPunchCard::App`
- `deckviewer`: verschiedene Views auf das Deck
  - Klasse `QPunchCard::CardEditor` (`cardeditor.h`)
  - Namensraum `QPunchCard::Navigator` (`navigatordock.h`, `navigatormodel.h`, `navigatorview.h`)
- `driver`, Namensraum: `QPunchCard::Device`
  - Device-Driver-Definitionen
  - Die eigentlichen Gerätetreiber in Unterordnen (etwa Documation-M200-Client)
- `qpunchcard`, Namensraum `QPunchCard`
  - Kartenmodell: `card.h`, `deck.h`
  - Ein/Ausgabe: `format.h`, `jones.h`
  - Primitive Darstellung: `widget.h`
- `text`: Textdarstellungen
  - Klasse `QPunchCard::Codec`
  - Namensraum `QPunchCard::Text` (`editor.h`, `editordock.h`)

Abseits dieser Struktur sind die externen Libraries (QExtSerialPort) eingebunden.

4 Vorsicht: Dock meint abkürzend DockWidget, also ein Andockfenster, wohingegen Deck abkürzend für *Card Deck* verwendet wird, zu deutsch Kartenstapel.

2.2.1. Qt und QExtSerialPort

Bevor ich auf das eigentliche Programm zu sprechen komme, möchte ich noch mal ein Wort zu den verwendeten Libraries auslassen.

Im Paper-Tape-Project hab ich die grafische Benutzeroberfläche zunächst mit C und Gtk+-2.0, später dann mit C++ und der Kapselung Gtkmm programmiert. Klassischer Konterpart zu Gtk+ in der Open-Source-Szene ist Qt, daher war bei diesem Projekt meine Motivation derart gelegen, mit Qt zu arbeiten.

Für den Open-Source-Einsatz ist Qt kostenlos verwendbar, dieses Projekt ist ausdrücklich unter der GNU Public License (GPL) lizensiert und versteht sich somit als Freie Software.

Zwar ist Gtk+ auch „plattformunabhängig“ (was ich im folgenden dafür verwende, dass es mindestens auf GNU/Linux als auch Windows läuft), aber erst mit Qt fühlen sich meines Empfindens nach viele Anwendung auch unter Windows wirklich nativ an. Des weiteren beinhaltet Qt viele weitere Libraries, vor allem in Hinsicht Datenstrukturen und Frameworks, womit es in einer ganz anderen Liga als Gtk+ (mit Glib) spielt. Spätestens beim Zugriff auf Hardware stößt man mit Qt allerdings auch auf Grenzen.

Während ich im Paper-Tape-Project noch separater Treiber für Linux und Windows (NT) entwickelt habe, habe ich an dieser Stelle die Open-Source-Library *QExtSerialPort* gefunden, die sich selbst als „cross-plattform serial port class“ beschreibt, die einen serieleln Port auf POSIX und Windows-Systemen kapselt[5]. Auf diese Weise brauch ich nur eine Client-Implementierung des PC Documation M200 µC Serial Communication Protocol schreiben, die sich Qt und QExtSerialPort bedient und damit auf Linux (i.A. POSIX-konformen Betriebssystemen) und Windows gleichermaßen läuft.

2.3. Das Kartenmodell

Lochkarten zu modellieren ist ein leidiges Thema, welches weiter oben bereits für den Mikrocontroller behandelt wurde. Auf dem Computer zählt nun weniger die Performance und mehr, dass es möglichst praktisch für den Programmierer ist. Außerdem wird zur Implementierung selbstverständlich auf die Klassen der Qt-Bibliothek zurückgegriffen.

Das Kartenmodell ist hier am PC um einiges umfangreicher als auf dem Mikrocontroller (zur Veranschaulichung: Der Gesamtumfang an Zeilen des PC-Kartenmodells entspricht dem Gesamtumfang des kompletten Mikrocontroller-Projektes). Alle Klassen sind in dem QPunchCard-Namensraum gesammelt und gliedern sich in mehrere Dateien. Im Einzelnen sind das:

Dateiname	*# Zeilen*	*Zusammenfassung*
`card.{c,h}`	150	Column, Card, CardIndex: Simple Klassen
`deck.{c,h}`	400	Deck, DeckIndex: Komplexe Klassen (Dokumentenmodell), außedem Undo-Klassen für das Undo-Framework
`format.{c,h}`	100	FileFormat-Klassen zur gekapselten Ein/Ausgabe von Decks
`jones.{c,h}`	280	Konkretes FileFormat nach Douglas Jones
`widget.{c,h}`	280	CardWidget: Grundlegende Zeichenfunktionen

5 http://qextserialport.sourceforge.net/

2.3.1. Lochkartenspalten und Kartendeck

Die 12 Bit breiten Lochkartenspalten werden naheliegenderweise in einem `QBitArray` gespeichert. Ähnlich wie auf dem Mikrocontroller auch wird das letztlich intern zu einer 16Bit-Speicherung führen. Der Column-Datentyp erweitert nur die besagte Qt-Klasse, dass damit allerdings auch Methoden zum Verändern der Länge der Column offengelegt werden, ist ein Manko am Design, den man durch eine aufwändige Kapselung freilich beheben könnte.

Ein klassisches Problem bei der Lochkartenspaltenspeicherung ist die Iterierung über die Positionen. Welche Position soll welche reale Lochkartenreihe einnehmen? Im Prinzip sind beliebige Permutationen möglich, wenn auch nur einige sinnvoll sind. In diesem Modell hab ich das QBitArray auf 13 Bit Breite gesetzt, um eine „natürliche Indizierung“ zu ermöglichen, wobei die Daten an Position 10 dann natürlich keiner realen Reihe entsprechen.

Die Card-Klasse speichert eine Lochkarte. Intern arbeitet sie freilich auch mit einem Array der fixen Breite 80, in dem Columns gespeichert werden. Es ist geplant, den Datentyp implizit geteilt zu implementieren, um einen perfomanteren Umgang zu ermöglichen. Die Implementierung als `QObject` scheidet aus, da ein QObject als „Identität“, und nicht als „Wert“ gehandhabt werden soll[6].

Bei der CardIndex-Klasse handelt es sich um eine Hilfsklasse, die nur von ihrer Header-Definition lebt (also quasi ohne Performance-Verluste benutzt werden kann) und die ein Bound-Checking einer Integerwertigen Indexvariable anbietet. Auf diese Weise ist gesichert, dass man in Karten nicht auf Spalten zuzugreifen versucht, die nicht gültig sind.

Wie man merkt, sind also alle diese Datentypen keine QObjects (aus zitierten Gründen).

2.3.2. Deck und Dokumentenmodell

Das `Deck` hingegen ist ein QObject, da es eben all die genannten Eigenschaften eines QObjects erfüllen soll. Es speichert eine Liste von Karten, außerdem den Dateinamen, unter dem es zuletzt gespeichert wurde, zudem ein FileFormat und auch noch einen UndoStack. Es bietet konstante Gettermethoden zum transparenten Zugriff auf die Karten an, außerdem Settermethoden, die letztlich das Undo-Framework kapseln.

Bei dem Undo-Framework handelt es sich, wie der Name sagt, um den objektorientierten Ansatz, die Funktion „Rückgängig“ und „Wiederholen“ in einer grafischen Benutzeroberfläche zu implementieren. Qt bietet dazu (anders als Gtk+-2.0 dies tat) ein Rahmengerüst an, in dem man seine eigenen Command-Klassen implementieren kann. Dies wurde mit DeckUndoCommand als Basisklasse, und davon abgeleitet exemplarisch DeckAppendCard und DeckModifyCard, implementiert. Um das System ernsthaft zu nutzen, müssten aber noch einige

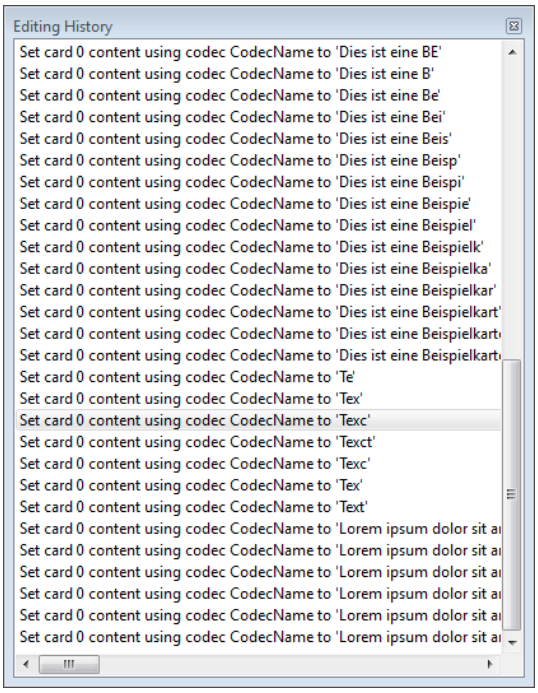


Abbildung 2.2: Qts Undo Framework erzeugt diesen schicken Dialog (QUndoView), in dem man die chronologischen Bearbeitungen des Dokumentes verfolgen kann (unter Bearbeiten *aktivierbar)*

6 „For these reasons, Qt Objects should be treated as identities, not as values. Identities are cloned, not copied or assigned, and cloning an identity is a more complex operation than copying or assigning a value.“, http://doc.trolltech.com/4.6/object.html

Commands mehr geschrieben werden.

Mit der DeckIndex-Klasse existiert eine zum CardIndex völlig analoge Klasse mit ähnlichem Funktionsumfang.

Zusammenfassend handelt es sich bei der Deck-Klasse um eine umfassende Modellierung des „Dokumentes“, wenn man vom Model-View-Controller-Pradigma ausgeht. Mit annähernd 100 Methoden ist es auch eine der umfangreichsten Klassen im kompletten Programm.

2.3.3. FileFormat und JonesFileFormat

Ich habe lange überlegt, wie sich im objektorientierten Ansatz die verschiedenen Dateiformate unter einen Hut bringen lassen. Die Lektüre von *Design Patterns - Elements of Reusable Object-Oriented Software* der Viererbande hat mich schließlich vom Strategie-Pattern überzeugt, welches mit dem `FileFormat` implementiert wurde. Gekapselt sind mit dieser abstrakten Klasse zwei Methoden `write(QFile, Deck)` und `read(QFile, Deck)`, mit denen jeweils ein Deck in eine Datei geschrieben bzw. von einer Datei gelesen werden kann. Welches Format dabei benutzt wird, entscheidet die konkrete Klasse bzw. deren Objekt, welches man zur Ein/Ausgabe benutzt.

Ich habe mir verschiedene Dateiformate überlegt, um binäre Lochkarten auf dem Computer in Form von Dateien zu speichern. Neben dem Formatvorschlag von Douglas W. Jones, zu dem er selbst eine Implementierung in C geschrieben hat, die ich auch in diesem Zusammenhang umgesetzt habe (JonesFileFormat), habe ich mit einem XML-Format geliebäugelt, welches zudem die Möglichkeit bietet, konkrete Textinterpretation, Zusatzkommentare zu einzelnen Lochkarten, Codecs, uvm. zu speichern. Allerdings würde dessen Implementierung wohl doch noch einige Zeit kosten und steht in keinem nennenswerten Verhältnis. Auch einen Namen (und eine DTD ;-)) hab ich mir bereits ausgedacht: Punch Card Markup Language, PCML.

2.3.4. CardWidget

Zwar gehört dieser Abschnitt strenggenommen nicht zum Kartenmodell, sondern selbstverständlich ganz eindeutig zur View-Komponente, allerdings gehört die Klasse wegen ihrem engen inhaltlichen Zusammenhang zum Namensraum `QPunchCard` und dementsprechend in dieses Kapitel.

Das `CardWidget` ist ein typisches QWidget nach allen Regeln der Kunst. Dementsprechend besitzt es ein paar Properties (Zeichnungsqualität, Editierbarkeitsflag) und natürlich einen Link zur Karte, die gezeichnet werden soll. Außerdem ist sie in der Lage, den typischen Text oberhalb der obersten Zeile einer Lochkarte zu zeichnen, wenn ihr ein Codec übergeben wird.

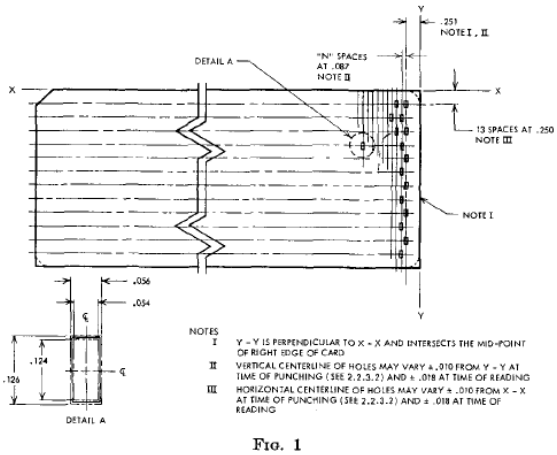


FIG. 1

Abbildung 2.3: Standardmaße einer Lochkarte nach dem Standard, vom CardWidget implementiert

Besonderes Augenmerk wurde beim CardWidget auf Exaktheit gelegt. Die Zeichenfunktionen implementieren vollständig den P*roposed American Standard – Rectangular Holes in Twelve-Row Punched Cards*[7] aus dem Jahre 1966. Je nach eingestellter Zeichnungsqualität kann man auf diese Weise eine originalgetreue Lochkarte sogar mit entsprechendem Bildmotiv gezeichnet bekommen. Das ist in der Qualität und Geschwindigkeit einmalig.

7 Siehe Anhang III

2.4. Textinterpretation

Die Interpretation von Lochkarten als Textdatenträger ist sehr wichtig. Ein Lochkarteneditor wäre kein ernstzunehmendes Werkzeug, wenn es diese Interpretation außen vor gelassen hätte. Daher umfasst das Programm eine Textkomponente. Essentiell ist dabei die Metapher vom *Codec*, also dem Zeichensatz. Aber auch die Darstellung und Weiterverarbeitung von Text ist elementar.

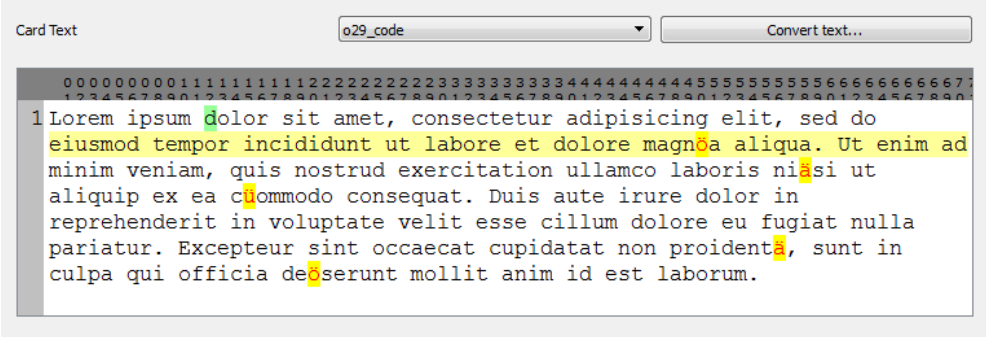


Abbildung 2.4: Ein exemplarisches Texteditor-Dock, hervorgehoben sind illegale Zeichen, die aktuelle Karte (Reihe), angezeigt ist ferner der aktuelle Codec (029 IBM Keypunch)

2.4.1. Der Codec

Die Klasse `Codec` im Namensraum `QPunchCard` definiert mathematisch gesehen eine bijektive Abbildung zwischen einem Zeichen (`char`) und einer Lochkartenspalte (`Column`). Zusätzlich sammelt die abstrakte Klasse Methoden zum Überprüfen der Gültigkeitsbereiche (Definitionsbereich und Wertebereiche der Abbildung). Das Klassenlayout entspricht dabei allen Regeln der Kunst, mit Überladung kann man Codec-Objekte nativ benutzen.

Als konkrete Implementierung des Codec-Konzepts existiert die `CharArrayCodec`-Klasse. Sie baut auf hardgecodeten Integerarrays auf, also Tabellen, die irgendwo im Quelltext untergebracht werden. Auf diese Weise wurden einige der Codectabellen von Douglas Jones in das Programm eingebaut, welcher einige Codes niedergeschrieben hat. CharArrayCodec-Klassen sind implizit geteilt und können daher beliebig als Objekte verwendet werden, ohne sich um Speicherverbrauch zu kümmern. Ohnehin gibt es keine Möglichkeit, den Codec nach Konstruktion zu verändern.

Als weitere Implementierungen habe ich ursprünglich eine Codecspeicherung in XML geplant, zu der ich aber nie gekommen bin. Eine weitere schöne Speichermöglichkeit, die die Bearbeitung eines Codecs direkt implizit im Karteneditor ermöglichen würde, wäre, einen Codec in einer Karte zu speichern, in dem man eine feste Zeichenreihenfolge vorgibt: So muss dann etwa die erste Spalte einem „a“ entsprechen, die zweite einem „b“, usw.

2.4.2. Der Editor

Die graphischen Bereiche des Editors bestehen aus dem Controllerwidget (`EditorDock` in dem Namensraum `QPunchCard::Text`), welches selbst eine Buttonleiste sowie den eigentlichen Text::Editor beinhaltet. In der Buttonleiste kann man auswählen, mit welchem Codec das angezeigte Kartendeck dekodiert werden soll, also in Text umgewandelt werden soll. Außerdem gibt es einen Button, um eine Codectransformation zu starten, den Inhalt des Decks also in einen komplett anderen Codec zu übersetzen. Eine solche Funktion ist noch nicht geschrieben, hat aber eine sehr einfache Funktionsweise: Man nimmt das momentane Deck, übersetzt es mit dem

aktuellen Codec in einen String und erzeugt daraus mit dem neuen Codec ein neues Deck.

Das Editorwidget selbst (`Text::Editor`) ist ein `QPlainTextEdit`-Widget, also ein Eingabefeld für Plain Text, welches mit eigenen Zeichenfunktionen um einen Bereich zum Ablesen der aktuellen Zeile sowie deren Hervorhebung und einiges mehr erweitert wurde.

Anzumerken ist, dass ein Editor selbst sein `QDocument` speichert und somit über ein eigenes Datenmodell verfügt, welches mit dem „offiziellen“ Modell (*Deck*) bei jeder Änderung abgeglichen werden muss. Auf diese Weise ist es aber auch möglich, dass man mehrere Texteditordocks gleichzeitig offen hat und demnach ein Deck in mehreren Codecs betrachten kann.

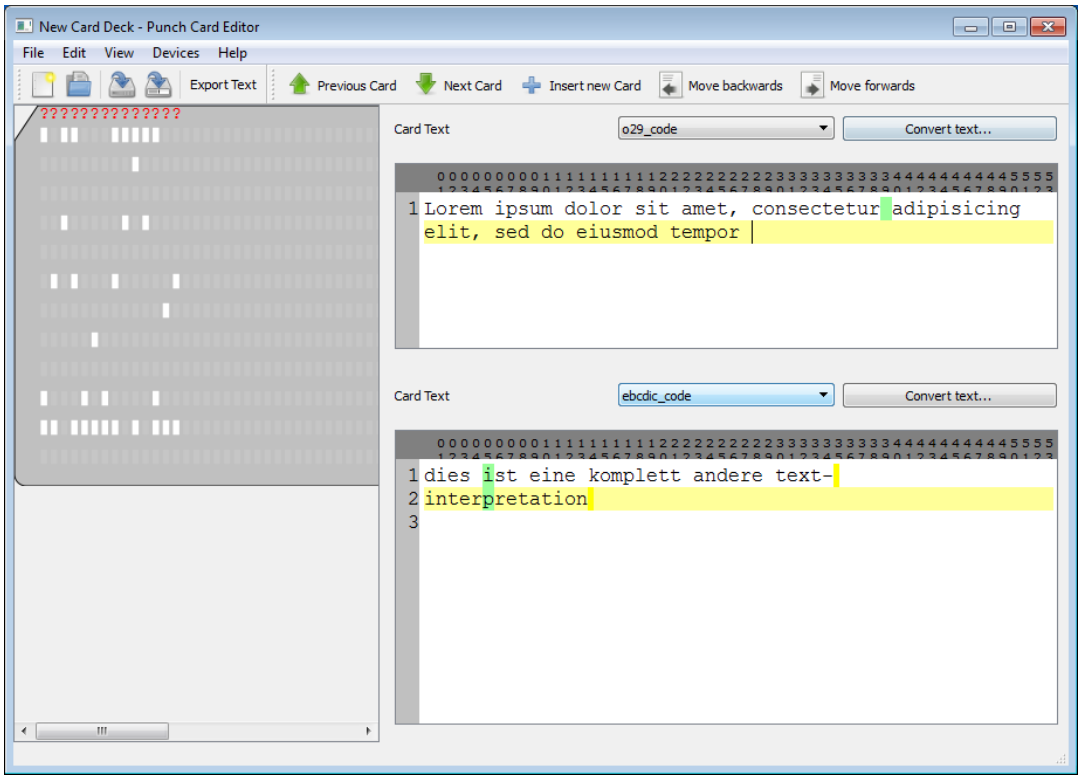


Abbildung 2.5: Zwei Texteditordocks bearbeiten ein Deck mit verschiedenen Codecs. Die Hauptansicht (links) kann damit noch nichts anfangen und zeigt lauter illegale Zeichen an

2.5. Weitere graphische Komponenten

Des öfteren fragt man sich beim Anschauen des Quelltextes eines Programmes, welches eine graphische Benutzeroberfläche erzeugt (von mir „graphisches Programm“ abgekürzt, entsprechend erklärt sich auch ein „graphischer Komponent“, kurzum etwa ein Widget), wo dort eigentlich vorne und hinten ist.

Anfangen tut der Programmfluss beim Punch Card Editor in der Datei `app/main.cc`. Dort steht die ominöse `int main(int argc, char**argv)`, die dann aber doch nicht mehr macht, als eine Instanz von `QPunchCard::App::MainWindow` zu erzeugen (und anschließend in die Endlosschleife (Event-Loop) von Qt fährt, womit sich auch die Frage klärt, wo das Programm aufhört).

MainWindow ist die zentrale Controller-Instanz im kompletten Programm. Dort wird der Pointer auf ein Deck gespeichert, außerdem werden dort alle Docks verwaltet.

2.5.1. Die Docks

Was hat es mit den Docks auf sich? Im Grunde genommen handelt es sich nur um ein gar nicht so junges Bedienkonzept für grafische Benutzeroberflächen, was sich vermutlich mit Einflüssen aus dem „Multi Document Interface“-Paragmatismus (MDI) entwickelt hat, aber (in diesem Fall zumindest) für eine SDI-Anwendung gebraucht wird. Ein Dockfenster ist etwa das standardmäßig geöffnete Navigatorfenster auf der linken Seite im Hauptfenster (siehe Abb. 2.1. auf Seite 9), aber auch das manuell geöffnete Texteditorfenster im unteren Bildbereich. Klickt man mit der Maus auf den Titelbereich oben im Dockfenster, dann kann man es an beliebige Positionen im Hauptfenster ziehen oder gar komplett als eigenständiges Fenster behandeln. Das ist bereits alles um die Theorie der Dockfenster; an sich nicht sonderlich spektakulär und dank Qt auch kein erwähnenswerter Programmieraufwand.

In dieser Anwendung hab ich den Docks allerdings eine Doppelfunktion zukommen lassen: Jedes Dock stellt eine Ansicht auf das Deck dar. Das ist zwar programmatorisch nicht festgelegt, sondern eher als Festlegung oder gar Beobachtung einzuwerten, aber es ist ein Bedienkonzept der Anwendung: Mit jedem neuen Dock kriegt man eine neue Ansicht des Decks. Freilich zerstört man damit auch keine Daten, wenn man ein Dock schließt – die eigentlichen Daten sind ja im Deck gespeichert, und das – so die Metapher – sitzt letztlich in der Mitte des Fensters, wo der CardEditor sein Dasein fristet und stets die global aktuelle Karte (`MainWindow::current_index`) anzeigt.

Damit klärt sich auch, wieso es möglich und sinnvoll ist, mehrere TextEditorDocks zu ermöglichen. Außerdem ist bereits von vorherein klar, wie sich ein Treiber bzw. dessen grafisches Frontend in die grafische Benutzeroberfläche einklinken muss: Per Dock.

2.5.2. Der Navigator

Das `Navigator::Dock` ist standardmäßig auf der linken Seite aktiviert. Es war ursprünglich so gedacht, dass es analog zu den Vorschaufenstern, wie man sie in Präsentationsprogrammen wie Microsoft Office PowerPoint, OpenOffice Impress oder Betrachtungsprogrammen wie Adobe Acrobat Reader kennt. Entsprechend vergrößert/verkleinert sich die Ansicht der Miniaturvorschauen der Karten je nach Größe des Docks.

Im Dock dargestellt wird die `Navigator::View`, welches eine `QListView` ist. Dabei findet das Model/View-Framework von Qt exzessiv verwendung, in dem es `Navigator::Model` als `QAbstractListModel` verwendet, welches als Items Pixmaps zurückgibt, die von `QPunchCard::Widget` gerendert werden. Als Qualitätsstufe wird dabei „Thumbnail“ gewählt, womit die kleinen, qualitativ nicht so hochwertigen Vorschaubildchen der Lochkarten gezeichnet werden.

2.6. Gerätetreiber

Mittlerweile dürfte man einen Überblick über die Modularität und Erweiterbarkeit des Programmes erhalten haben. Entsprechend benötigt man nicht viel Vorstellungskraft, um sich ein weiteres Dock, mit dem man neue Lochkarten erzeugen oder bestehende exportieren kann, vorzustellen, mit anderen Worten also ein Dock, mit dem man neue Lochkarten einlesen oder bestehende stanzen kann, also letztlich ein Frontend für die Ansteuerung eines Lochkartengerätes, also ein Frontend für einen Gerätetreiber.

2.6.1. Documation M200 Client

Der Documation M200 Client implementiert das PC-µC-Protokoll (Kapitel 1.3.1. sowie Anhang) auf Clientseite und kann somit mit dem Server (Mikrocontroller) über den seriellen Anschluss eines PCs kommunizieren.

Herzstück ist der Controller (`QpunchCard::Device::DocumationM200::Controller`), welcher das Dock darstellt, also das Frontend. Er startet auf Benutzerinteraktion dann einen ClientThread, welcher die eigentliche (offensichtlich asynchrone) Kommunikation über RS232 vornimmt.

Die Implementierung als eigener Thread ist möglicherweise nicht das Mittel der Wahl. Die eigentliche Motivation war, in einem eigenen Thread zu arbeiten, um einen möglichen Datenverlust durch die ohne Flusssteuerung betriebene serielle Schnittstelle bereits im Vorfeld zu vermeiden, allerdings scheint sich die QExtSerialPort-Library ohnehin eher für Event-gesteuerte Programmierung zu eigenen. Defacto war der Plan, den Thread in einer Endlosschleife auf Daten warten zu lassen (was auf Single-CPUs keine sonderlich gute Idee ist, weil dann die GUI nicht mehr reaktiv ist, also muss man doch wieder Schlafbefehle einfügen, womit die Idee mit dem Thread hinüber ist, außer man macht soetwas wie einen Poll/Select).

Die Kommunikation mit dem Haupthread funktioniert dann über Qts hauseigenes Signal/Slot-System, was Thread-sicher ist bzw. dafür sorgt, dass ein Slot in dem Thread aufgerufen wird, in dem sein Objekt „lebt“ - womit man bereits eine vollständige Interprozesskommunikation (IPC) implementiert hat.

Mit Semaphoren hingegen muss noch abgesichert werden, wann der Thread sich killt (Deadlock beim Beenden des Programmes...) und wie die neu eingelesenen Karten in das Deck eingepflegt werden. Das Einlesen von Karten in das lokale PC-Kartenmodell funkioniert bereits (wie man durch eine Debugging-Ausgabe des Datenmodelles nachprüfen kann), meines Wissens sogar zeitkritisch (d.h. ohne Datenverlust).

2.7. Status des Teilbereiches, TODO-Liste

Das Programm leidet vor allem unter zahlreichen Bugs, die an irgendwelchen komischen Stellen auftreten und teilweise Programmabstürze verursachen. Leider macht Debugging von C++-Anwendungen auch nicht sonderlich viel Spaß, wobei es mit einer richtigen IDE (QtCreator), die tatsächlich Debugging bis auf Symbolebene ermöglicht, sogar möglich ist.

Um den Bugs entgegenzuwirken, müssten in erster Linie Komponenten deaktiviert/entfernt werden (die blödsinnige (allerdings extern gewünschte) Spaltenhervorhebung ist ein typischer Fall von Fehlentwicklung, die nicht benötigt wird).

Natürlich muss das Programm noch an vielen Stellen vervollständigt werden. Viele Buttons sind ohne Funktion, der Documation M200-Treiber funktioniert noch nicht (in dem Sinne, dass er autonom das Deck füllt). Die Speicherroutinen stürzen ständig ab (Jones-Umwandlung), außerdem braucht man einen Konfigurationsdialog für die serielle Schnittstelle (wäre eigentlich im Kompetenzbereich des QextSerialPorts).

Gerade das Finden von Bugs lässt sich schwer deterministisch im Zeitaufwand quantifizieren, aber größtenteils, wie man dem Text entnehmen kann, ist das Projekt fertig.

Anhang I: PC Documation M200 µC Serial Communication Protocol

Punched Paper Project – December 2009
Author: Sven Köppel – *$Id: protocol.htm 59 2010-01-22 18:44:34Z sven-win $*

Abstract

This file describes the communication design via RS232 to the microcontroller driver software that was written for controlling the *Documation M200* punch card reader.

This protocol is mainly ASCII based and designed with the target for a lightweight microcontroller implementation (using the C programming language) and a client application, running on a computer. Alternatively, it is supposed to be human readable at a terminal, so an human shall be able to replace the client program on a PC.

Revision

Version 1.0.0
: First write down from some scratch paper

Version 1.0.1
: Extended the example (4)

Contents

1. Motivation

In 2009 I've programed a microcontroller control and read out the data from a punch card reader. It was supposed that the microcontroller software translates the data and prints them out to the connected computer.

Since punch cards, having 80 columns with 12 bit each, are a bit weird to handle on modern computer platforms (all talking in 8 bit wide bytes), the question of how to represent such a punch card on a computer arised soon. Furthermore, this Documation M200 punch card reader device isn't complex at all, but there are nevertheless some status and error signals that would be nice to be transfered to the computer, too. After all, it's supposed that humans could use that device with their computers without being freaks. So there must be some running some client program on the computer that actually accepts the data from the microcontroller. For processing this workflow, there's a need of a standarized protocol.

Well, we're in 2009, today there are many, many protocols outside to use. But there's another point: Users should be able to use the microcontroller *without* the client program, because I don't want to bind the microcontroller to one special client program.

Since we are using RS232 as transportation layer between computer and microcontroller, we already have some sort of flow control and handshaking, so we need only some data model. Using ASCII is quite the only choice if humans should be able to handle it directly from terminals.

Summing up, we need a protocol with these features:

- ASCII protocol
- Efficient in implementation (small memory footprint on microcontroller)
- Some kind of request code set, some kind of server response set
- Compact transmission of data (since RS232 throughput is quite low)

The proposed protocol in the present document is intended to cover all these issues.

2. Definitions

At first I want to introduce some simple abbreviations:

Reader
: refers commonly to the punch card reader *Documation M200*

Server

µC
: This is the Microcontroller (with the software running on it)

Client

Application

Frontend
: This is the PC in general or some application running on it, like a terminal or a graphical frontend.

3. Hardware wiring and RS232 settings

This is a rather unimportant chapter concerning how to get a RS232 communication line to the microcontroller.

Discussion about RS232 male and female sockets are too specific at this point, since they only cover our needs. Our microcontroller board is equipped with a **RS232 female socket**. Since computers are using typically RS232 male sockets, you will need an ordinary RS232 connection cable. The connection is **not** crossed.

We will use **hardware flow control** with 8bit bytes and no parity bits. Currently we are using 38400 baud.

4. General protocol layout

This is a typical communication. The client inputs the **bold** digits (the lines with the *italic* comments, which are of course not intended to be printed), and the server returns every other text.

```
200 v1.0 Welcome to the M200 Punch Card Reader Microcontroller
110 Type 'h' for help
0   (lets ping the server)
100 pong
```

```
4   (set to debug output format)
204 input format set okay
1   (say the microcontroller that I'm ready)
201 Strobing start signal...
400 This is any debug output about some
401 ping changements, as an example
424 for a debug statement
470 /123456789 012|
700 0001000000 000
701 0000100001 100
702 0010010000 010
...
778 1000000100 000
779 0001000000 000
453 hmmm, i'm finished somehow :-)
2   (okay, stop it)
202 Stopped reader.
```

5. Client request codes

List of codes:

char	signification
h	Order µC to print out a help of client request codes
c	Connect: Will print out version and internally reset
0	Ping (microcontroller will print out pong if not hung up)
1	Start punching
2	Stop punching
3	Set output to *hexadecimal output*
4	Set output to *debug output*
5	Set output to *binary jones output*
6	Reset internal buffers and stop punching out
7	Toggle very verbose output (experts only)

6. Server response codes

Range	Typical numbers and comments/data	signification
100	100 pong	The answer to a ping request
	101 help output	Commenting and help output on any other [101; 199] line
200	200 v1.0 Welcome to M200 Card Reader	Answer to Connect request: Print out version (in v[Major].[Minor] format) and welcome message
	201 Going to start soon...	Acknowledgements to request code, code [x] will produce 20[x] as answer (See table above)
	202 Stopping as soon as possible...	
	203 Set output to HEX	

	204 Set output to DEBUG	
	205 Set output to JONES	
	206 Performing a soft RESET...	
	207 Verbose output ON	
300	301 ERROR rising	Scheme: 3[x][y], where [x] = Signal line, [y] = 1 or 0, depending on state
	300 ERROR falling	
	31- READY	
	32- BUSY	
	33- HCK (Hopper Check)	
	34- MOCK (Motion Check)	
400	4-- Any debug output	Specific debug output, can be completely ignored
500	500 illegal character at input: 'd'	Bad input, see table above for correct request codes
600	600 0x12 0x85	Hexadecimal output. Scheme: 6[xy] where xy is the column number, going from 00 to 79
	601 0x00 0xA5	Data scheme: low octett, high octett...
	602 0x72 0x42	
	...	
	679 0xBE 0xEF	
700	470 /123456789 012|	Debug output. Displays the complete punch card as ASCII art. The first line
	700 000000010 000	(470) may occur to make some nice column heading
	700 000000010 000	
	...	
	779 111011011 101	
800	800 BINARY	Binary Jones output (123 bytes) starting after newline. See next chapter for format
	808 FINISHED	Binary Jones output finished (maybe... we'll look)
900	900 Bad card! Stopping reader	Bad device behaviour, possibility of data loss! Stop everything immediately.
	905 Printer Buffer Full	Ugly code design caused status output information loss (no data loss)

7. Punch card representations

As you can already extract from the response code table, there are three punch card formats:

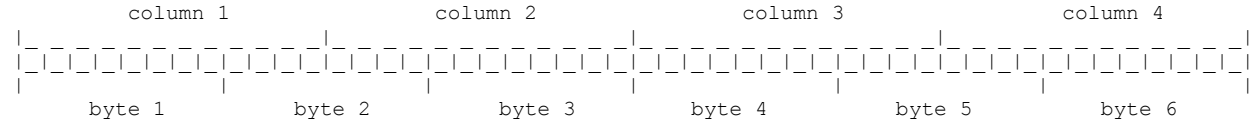
- *700 range:* Debug output. Ideal for actually viewing the card on the screen without any frontend program, but scales very bad because there are about 1600 bytes to transfer for each card, and that takes much time, so card reading cannot be performed in real time any more

- *800 range:* Binary output. Ideal for high speed application processing. One card expands to exactly 123 bytes (almost "compressed") data.
- *600 range:* Hexadecimal output. Maybe a good tradeoff for high speed "real time" debugging: With consuming about 900 bytes, its almost 2 times faster than debugging output, but still about 8 times slower than binary output

The only real world data format is the binary format, named after the inventor, Douglas Jones. He proposed a format for complete card decks, as excepted in APPENDIX A. We do **not** use the fully featured syntax (which defines some bytes as meta data) but only the basic idea of encoding an 80 columns punch card to exactly 120 octetts. So a typical card data transportation following this protocol will look like:

```
800 BINARY DATA\r\n.....................................808 FINISHED\r\n
                        ^         120 bytes payload            |
          start of data exactly at caret, end after 120 bytes
```

This is an example for the first six bytes in the payload:



For the exact decoding of one column (three nibbles) see APPENDIX II.

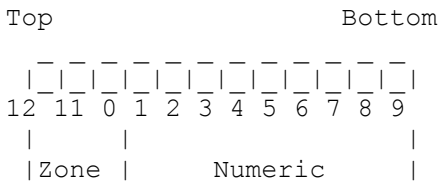
Anhang II: Binary Punch Card Representation (by Douglas Jones)

This appendix is a copy of Emulated Punch Card Decks *by Douglas W. Jones, Part of the Punched Card Collection. I have attached this copy, since the original document could become unavailable at any time or its identifier could be changed. Original source:* http://www.cs.uiowa.edu/~jones/cards/format.html

The purpose of this file is to document the card formats I hope will be used by developers of emulators for antique computers, so that card "decks" punched on one emulator will be readable by others, and so that a standard suite of utility software can be used to support "card" processing on all such emulators.

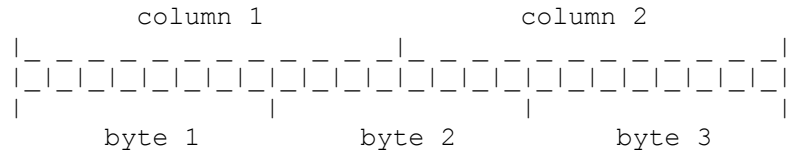
The following file format is proposed for use on such emulators:

One column of a card holds 12 bits; in the file, we lay them out as follows, with ones representing punched holes:



Note that there are multiple mappings from ASCII to card codes, reflecting different keypunches and reflecting different interpretations of non-ASCII graphics such as cent-sign and logical-not.

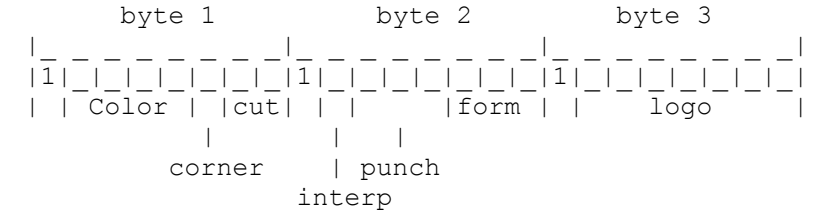
A design dilemma presents itself: There are two rational ways to map card columns to byte sequences, highbyter (or bigendian) and lowbyter (or littleendian). The choice is arbitrary. Here, we will be bigendian as follows:



Card files need a distinguished magic number or prefix to prevent accidental interpretation of random files as virtual card decks. Here, we will use the ASCII prefix "H80", in honor of the FORTRAN Hollerith format used to read one card image as uninterpreted text.

This prefix allows extension to H82, where both columns 0 and 81 are represented -- note that some verifiers punched in column 0, and that IBM 026 keypunches could generally punch column 81. Emulators should generally accept H82 files and ignore the extra leading and trailing columns! Similarly H51 can be used to represent the 51 column cards used for some business forms, and H53 can be used to represent these cards with data punched in columns 0 or 52. Mixing multiple card sizes in one deck was never practical, so emulators need not support it, and most emulators need only support H80 (and H82, but ignoring columns 0 and 81).

To support keypunch emulators and card deck display and editing programs, a 3 byte prefix is needed on each card. Emulators that read the holes on a card should ignore this prefix (other than using it to verify that a card-file is indeed being read). The prefix format is as follows:



Cards came in many colors, and cream cards (the default) came with pale colored stripes coarsely printed across the top margin. The following color selection should be more than sufficient:

```
Color:  cream (unbleached)      0000 default!
        white                   0001
        yellow                  0010
        pink                    0011
        pale blue               0100
        pale green              0101
        pale orange             0110
        pale brown              0111 rare
        yellow stripe           1010 very common!
        pink stripe             1011
        pale-blue stripe        1100
        pale-green stripe       1101
        pale-orange stripe      1110
        pale-brown stripe       1111
```

Most cards had rounded corners to prevent fraying, but you could save a bit of money by ordering square cornered cards. One of the top corners of each card was usually cut off diagonally. Cards with no cut and with both top corners cut were made!

```
Corner: round   0  default
        square  1

Cut:    neither 00 rare
        right   01 common
        left    10 default
        both    11 rarest
```

Keypunches usually printed on the top edge of the card as they punched. Each model of keypunch printed its own interpretation of the character codes used, and if a card was punched by a high speed computer-driven punch, it was not usually printed. Any deck of cards could be run through an interpreter which overprinted the card with a somewhat eccentric interpretation of the data on the card. Accurate emulation of the particular character sets used by different interpreters is probably not necessary (most programmers got used to the fact that interpreters hardly ever printed in the character set that they wanted!)

```
Interp: no      0  default
        yes     1

Punch:  none            000 punch didn't print
        026 Commercial  001 older, with "&-#@.¤$*,%"
        026 FORTRAN     010 older, with "+-='.)$*,("
        029             100 default
```

Cards could be printed with a number of forms and logos. Most corporate logo cards were based on a standard form, with the logo added in light grey in the center of the card. The set of available forms was open ended, and of course, the set of logos was open ended. This virtual card format allows only a few of them (and far too many logos).

```
Forms:  no printing  000
        IBM 5081     001 all numeric rows marked (default)
        IBM 507536   010 only colmns 1, 80 and row 0 marked
        IBM 5280     011 8 fields, 3-3-3-1 subfields
        DSI 327      100 8 fields, 5-5 subfields
        IBM 733727   101 20 fields, 4 chars each
        IBM 888157   110 FORTRAN column layout
                     111


Logo:   none                                    0000000 \ default may
        (your institution's logo here)          0000001 / vary!
        IBM 821924 (701 binary)
        IBM 821162 (701 assembly)
        IBM 874266 (7090 assembly)
        Rechenzentrum RWTH Aachen
        University of Alaska
        University of Arizona Computer Center
        Battelle Laboratories
        Bell Labs (old style)
        Bell Labs (GE 600, new style)
```

A matter of philosophy: The forms listed in the forms category above are those which served as the basis for large numbers of institutional overprints! There were huge numbers of other standard forms, some of which should be probably be assigned as logos and overprinted on the blank form. As a result, these forms cannot have custom logos, but this should not cause great problems with users of the full-blown emulator package envisioned. Please, if you ever come up with keypunch emulators that support custom logos, inform me of the logos you use and reserve their numbers!

Anhang III: Proposed American Standard: Rectangular Holes in Twelve-Row Punched Cards

Dieser Anhang ist eine Kopie der Publikation S. Gorn: Rectangular holes in twelve-row punched cards, erschienen in Communications of the ACM, Volume 9, Issue 10 (October 1966), Page 763. *Der Volltext stand zum Entstehungszeitpunkt unter der folgenden Adresse zur Verfügung:* http://doi.acm.org/10.1145/365844.365872

Die Publikation wird in der Druckversion an dieser Stelle im Anhang geführt, da das Originaldokument im Internet nur noch sehr rar verfügbar ist und auch an keiner anderen Stelle zitiert wurde.