Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Das „a“, Teil 2

Henry Dettmer

Matrikelnummer: 4300293

henry.dettmer@fu-berlin.de

Gutachter: Prof. Dr. Lutz Prechelt

Zweitgutachter: Prof. Dr. Carsten Schulte

Betreuer: Franz Zieris, M.Sc.

Berlin, den 25.04.2013

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 25.04.2013 _____

(Henry Dettmer)

Zusammenfassung

In dieser Bachelorarbeit werden die Software-Abläufe eines Computersystems analysiert und dargestellt, die dazu führen, dass aus einem Druck auf die „A“-Taste einer Tastatur ein sichtbarer Buchstabe auf einem angeschlossenen Bildschirm wird.

Der untersuchte Computer verwendet ein auf dem Linux-Kernel basierendes Betriebssystem und das X Window System zum Darstellen von grafischen Programmen. Als Textverarbeitungsprogramm kommt OpenOffice.org Writer zum Einsatz.

In erster Linie wird dabei die Vorgehensweise beim Dokumentieren dieses Programmablaufs und beim Untersuchen der erwähnten Komponenten und der anschließenden Aufbereitung der Daten beschrieben.

Die Daten werden schließlich als möglichst alleinstehend verständlicher Text und als Skizze einer Präsentation präsentiert.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel und Aufbau der Arbeit . . . 1

2 Vorgehensweise **3**
2.1 Vorüberlegungen . . . 3
2.2 Untersuchung der Komponenten . . . 4
2.2.1 Linux-Kernel . . . 4
2.2.2 X.Org - Eingabe . . . 5
2.2.3 OpenOffice.org . . . 7
2.2.4 Kommunikation mit X.Org . . . 7
2.2.5 X.Org - Ausgabe . . . 9
2.2.6 Weitere Komponenten . . . 9
2.3 Ordnen der Daten . . . 9

3 Ergebnis **11**
3.1 Überblick . . . 11
3.2 Eingabe . . . 12
3.3 Ausgabe . . . 13

4 Zusammenfassung **14**

A Ausführliches Ergebnis **15**
A.1 Hardware . . . 15
A.1.1 Koordination der Eingaben . . . 15
A.1.2 Hardwareseitige Behandlung von Interrupts . . . 16
A.1.3 Interrupt eines Tastendruckes . . . 18
A.2 Kernel . . . 18
A.2.1 Aufgaben des Kernels . . . 18
A.2.2 Aufbau und Strukturierung des Kernels . . . 19
A.2.3 Anforderungen an die Interrupt-Bearbeitung unter Linux . . . 19
A.2.4 Implementierung der Interruptbehandlung in Linux . . . 20
A.2.5 Behandlung der Events . . . 28
A.3 X.Org-Server - Eingabe . . . 29
A.3.1 Event-Erstellung . . . 30
A.3.2 Event-Auslieferung . . . 33

A.4 OpenOffice.org 36
A.4.1 Anforderungen an OpenOffice.org 36
A.4.2 Visual Component Layer (Eingabe) 37
A.4.3 OpenOffice.org Writer 43
A.4.4 Visual Component Layer (Ausgabe) 50
A.4.5 Kommunikation mit X.Org 51
A.5 X.Org-Server - Ausgabe 52
A.5.1 Von der Anfrage zum Bild 54
A.5.2 Zur Grafikkarte 56

B Präsentationsentwurf **58**

C Inhalt der CD **60**

Literaturverzeichnis **61**

1 Einleitung

1.1 Motivation

Die Eingabe eines Textes mit der Tastatur ist für viele Computerbenutzer eine ihrer alltäglichsten Handlungen. Doch was steckt eigentlich dahinter? Wie gelangen die einzelnen Buchstaben von der Tastatur in das Textverarbeitungsprogramm und letztendlich auf den Bildschirm? Die vorliegende Arbeit beschäftigt sich genau mit dieser vermeintlich einfachen Frage, hier konkret umgesetzt durch das Drücken der Taste „A“ in einem leeren OpenOffice.org-Writer-Dokument unter Verwendung eines Linux-Betriebssystems.

Die Dokumentation und Darstellung dieses Vorgangs soll einen Einblick in die für viele ungeahnte Komplexität alltäglicher Computeranwendungen geben und den Weg des Buchstabens in verständlicher Form darstellen.

Dies ist bereits die zweite Bachelorarbeit zu diesem Thema, die in der Arbeitsgruppe Software Engineering geschrieben wird. Die gleiche Fragestellung untersuchte 2009 bereits Franz Zieris [ZIE09], der jedoch damals nicht den gesamten Ablauf dokumentieren konnte.

Die Unvollständigkeit der Vorarbeit lässt sich wohl darauf zurückführen, dass der Detailgrad der Ausarbeitung von abstrakt bis sehr detailliert nahezu beliebig ansetzbar ist. Deshalb ist dies auch eine sehr anspruchsvolle Aufgabe. So ist der Vorgang in der gröbsten Form mit einigen Sätzen erklärt, während bei genauster Betrachtung z. B. schon die Beschreibung der Kommunikation zwischen Betriebssystem und Hardware reichen würde, um Unmengen von Seiten zu füllen.

Für mich persönlich bot dieses Projekt noch einen weiteren Anreiz: Als Benutzer von Open-Source-Programmen steht man immer vor der Möglichkeit, diese auch selbst zu verändern oder zu verbessern. Betrachtet man aber Software wie den Linux-Kernel oder X.Org, die über Jahrzehnte gewachsen ist und dazu noch nah an der Hardware agiert, ist die Hemmschwelle, diese Möglichkeiten auch wahrzunehmen, groß. Diese Arbeit stellte für mich so auch eine Gelegenheit dar, den Sprung ins kalte Wasser zu wagen und tief in die Quellcodes dieser Programme einzutauchen.

1.2 Ziel und Aufbau der Arbeit

Aufgabe dieser Arbeit ist es also, den gesamten Software-Ablauf zu untersuchen, der sich abspielen muss, bis aus dem Drücken der Taste „A“ auf der Tastatur ein sichtbarer

Buchstabe auf dem Bildschirm geworden ist.

Die gesammelten Daten sollen in zwei Formen aufbereitet werden: zum Einen als textliche Ablaufbeschreibung, die verständlich sein sollte, ohne weitere Ressourcen hinzuzuziehen, und zum Anderen in Form einer Präsentation, die als Grundlage für eine spätere Vorstellung der Ergebnisse dienen könnte.

Im Folgenden erläutere ich als Erstes meine Überlegungen und Vorgehensweise beim Untersuchen der zu betrachtenden Software in chronologischer Form (Kapitel 2). Danach folgen eine gekürzte Version der gesammelten Erkenntnisse über den Programm-Ablauf (Kapitel 3) und eine abschließende Zusammenfassung (Kapitel 4).

Der Anhang enthält eine ausführlichere Beschreibung des Software-Ablaufs (Anhang A) und den Entwurf einer Präsentation zum Darstellen der Ergebnisse (Anhang B).

2 Vorgehensweise

2.1 Vorüberlegungen

Zu Beginn der Arbeit entschied ich mich dazu, für die zu untersuchenden Programme eine virtuelle Maschine einzurichten. Ich versprach mir dadurch mehrere Vorteile.

Zum Einen würde ich so die laufenden Programme von einem unabhängigen System aus über eine Netzwerkverbindung untersuchen können. Damit kann vermieden werden, dass Eingaben zum Steuern des Debuggers auf dem untersuchten System gemacht werden, die dann wiederrum in den Untersuchten Komponenten verarbeitet werden. Außerdem würde ich mögliche Fehler, z. B. durch Veränderungen am Betriebssystem, durch Backups der virtuellen Festplatte schnell rückgängig machen können.

Als Emulationsprogramm wählte ich QEMU[QEMU], da dieses uneingeschränkt frei verfügbar ist (im Gegensatz zu z. B. VirtualBox) und durch die Unterstützung von KVM (Kernel-based Virtual Machine) unter Linux als Hostsystem sehr gute Geschwindigkeiten erzielt.

Um die gleichen Versionen der Programme untersuchen zu können, die auch in der Vorarbeit verwendet wurden, richtete ich als nächstes auf der virtuellen Machine eine ältere Version der Linux-Distribution Debian (Version 5.0)[DEB] ein. Hierfür gab es mehrere Gründe:

- Ich hatte bereits Erfahrung im Umgang mit Debian.
- Alle zu untersuchenden Programme standen in den benötigten Versionen zur Verfügung.
- Der Quellcode von Programmen lässt sich direkt über die Paketverwaltung herunterladen. Anschließend kann man diesen modifizieren und zu Paketen kompilieren, die sich wiederum mit der Paketverwaltung installieren lassen. So vermeidet man Konflikte zwischen manuell und mit der Paketverwaltung installierten Programmen.

Nachdem ich das Grundsystem so eingerichtet hatte, untersuchte ich folgende Programme und Schnittstellen (Abbildung 2.1):

1. Linux-Kernel[KERNEL],
2. Kommunikation zwischen Kernel und X.Org,
3. X.Org[XORGa],

4. Kommunikation zwischen X.Org und OpenOffice.org,
5. OpenOffice.org[OOOa, OOOb],
6. Kommunikation zwischen X.Org und der Grafikkarte.

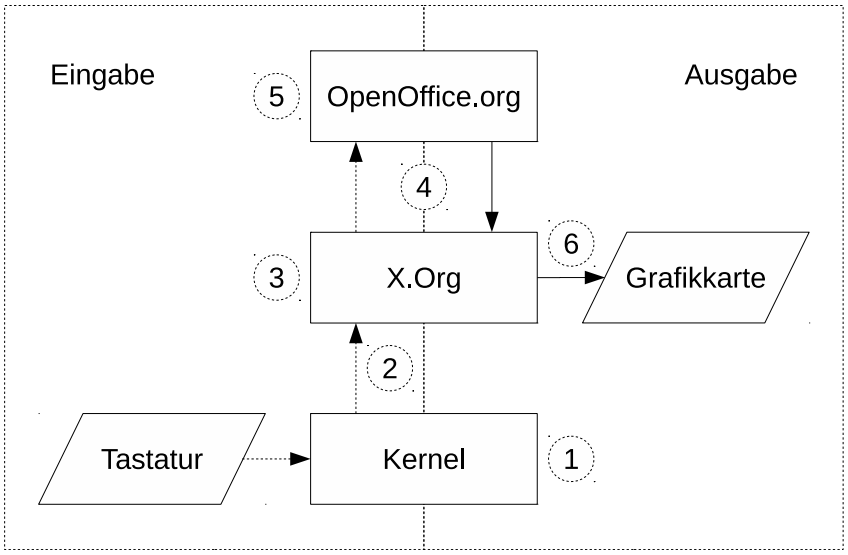

Abbildung 2.1: Schematischer Ablauf

Bei der Untersuchung von Kernel und OpenOffice.org konnte ich hierbei auf die Vorarbeit zurückgreifen; in den übrigen Punkten waren intensivere Untersuchungen notwendig.

Für mich war es von Anfang an klar, dass ich mich bei der Untersuchung des Ablaufs am Datenfluss orientieren würde. Dies schien mir am ehesten die Verfolgung des „a“ durch die verschiedenen Programme umzusetzen.

Eine reine Untersuchung des Kontrollflusses würde sicher ebenso interessante Einblicke in die Funktionsweise eines Betriebssystems geben (man denke z. B. an Scheduling), diese wären aber zeitweise weit vom eigentlichen „a“ entfernt.

2.2 Untersuchung der Komponenten

In den folgenden Abschnitten wird das Vorgehen beim Untersuchen der einzelnen Komponenten näher beschrieben. Es werden aufgetretene Probleme und mögliche Lösungen für diese sowie verwendete Techniken beschrieben.

2.2.1 Linux-Kernel

Da ich beim Kernel auf die vorherigen Ergebnisse zurückgreifen konnte, beschränkte ich meine Untersuchungen zunächst darauf, diese nachzuvollziehen.

Seit Version 2.6.26 (2008) ist es möglich, den Linux-Kernel mit einem Debugger zu untersuchen[KGDB]; für meine Zwecke erschien es mir jedoch ausreichend auf eine „klassische“ Debug-Methode zurückzugreifen[Rus05]. Durch den Befehl `printk` ist es einfach, Debug-Nachrichten in den Quellcode einzufügen, die dann im Kernel-Log ausgegeben werden. Leider ist diese Methode etwas zeitaufwendig, da nach jeder Veränderung der Kernel neu kompiliert werden muss.

Zwei Maßnahmen boten sich an, um diese Zeit zu verkürzen: Multithreading und die Verwendung eines Compilers mit Cache-Funktionalität. Multithreading ist leicht über ein Compilerargument umzusetzen. Als Compiler verwendete ich ccache[CC], welcher zum eigentlichen Kompilieren wiederrum auf die GNU Compiler Collection (GCC) zurückgreift. Deshalb kann ccache einfach als direkter Ersatz für GCC verwendet werden. Beim Kompilieren wird dann für die Ergebnisse ein Hash der relevanten Informationen des Quellcodes gespeichert. Wird der gleiche Quellcode nun erneut kompiliert, ohne dass eine Änderung aufgetreten ist, wird das bereits kompilierte Ergebnis wiederverwendet.

Da Debian ein Programm zur Verfügung stellt, um selbst neue Kernel-Pakete zu erstellen (make-kpkg), und sich so ein manuelles und fehleranfälliges Installieren des veränderten Kernels erübrigt, wollte ich dieses auch verwenden. Dabei ergaben sich jedoch einige Probleme, die darauf zurückzuführen waren, dass sich die Konfiguration des Programms (also das Einstellen der Compiler-Optionen) bei verschiedenen Programm-Versionen erheblich unterscheidet und manche dieser Versionen leider nur unzureichend dokumentiert sind.

Nachdem diese Hindernisse bewältigt waren, konnte ich die Ergebnisse der Vorarbeit durch eigene Kernel-Nachrichten und Analyse des Quellcodes nachvollziehen. Allerdings ist auch bei diesen vermeintlich harmlosen Ausgaben Vorsicht geboten. Bei zu häufigem Aufrufen kann das permanente Schreiben des Kernel-Logs das System zum Stillstand bringen.

Aus der Vorarbeit ging nicht hervor, wie die Daten vom Kernel zum X.Org-Server gelangten. Um hier einen Ansatz zu finden, untersuchte ich zunächst den X-Server und verfolgte von dort ausgehend die Eingabe-Daten zu ihrem Ursprung, einer Gerätedatei. Anhand dieser konnte ich dann die entsprechende Komponente im Kernel identifizieren und wie vorher beschrieben untersuchen.

2.2.2 X.Org - Eingabe

Der X.Org-Server diente mir als Einstiegspunkt; in dessen Entwicklerdokumentation fand ich einige Funktionen, die an der Verarbeitung von Tastatureingaben beteiligt sein sollten. Von diesen Funktionen ausgehend wollte ich mit Hilfe eines Debuggers den Programmablauf untersuchen.

Dabei ist jedoch zu bedenken, dass der X-Server, wie der Kernel, unmittelbar mit der Hardware in Verbindung steht. So verfügt X z. B. über eigene Grafik-Treiber, mit denen direkt mit der entsprechenden Hardware kommuniziert wird. So kann es beim Debugging

und dem damit einhergehenden Anhalten und Fortsetzen des Programmablaufs leicht zu Systemabstürzen kommen.

Um dies zu verhindern, wird der Debugger nicht vom gleichen System gesteuert, auf dem der X.Org-Server läuft, sondern über eine Netzwerkverbindung.

Als Debugger verwendete ich GNU Debugger (GDB). Damit dieser die Verbindung zwischen laufendem Programm und Quellcode herstellen kann, muss das Programm mit sog. Debug-Symbolen kompiliert werden. Bei Debian stehen viele Programme, darunter auch der X.Org-Server, bereits so kompiliert zur Verfügung; für den Tastaturtreiber von X.Org musste ich allerdings noch manuell ein solches Paket kompilieren.

Anschließend konnte ich also per SSH eine Verbindung zur virtuellen Machine aufbauen und dann auf dieser GDB ausführen und auf die laufende X.Org-Instanz richten. Auch wenn der X-Server jetzt angehalten wurde, konnte ich GDB weiterhin benutzen, was mir erlaubte, die üblichen Debuggingfunktionen (zeilenweises Ausführen, Haltepunkte, Untersuchen von Variablen, usw.) zu benutzen.

Ich begann also den Programmablauf zu untersuchen und dokumentierte diesen in Form von manuell erstellten Diagrammen (die eigentlich Bäume darstellen). In dem hier angeführten Ausschnitt (Abbildung 2.2) werden z. B. von der Funktion `xf86PostKeyboardEvent` nacheinander die Funktionen `GetKeyboardEvents` und zweimal `mieqEnqueue` aufgerufen, wobei `GetKeyboardEvents` wiederum `GetKeyboardValuatorEvents` aufruft. Dazu speicherte ich Stacktraces für alle Blätter des Baums. Diese enthalten die vollständige Aufrufhierarchie mit übergebenen Argumenten und Codestellen.

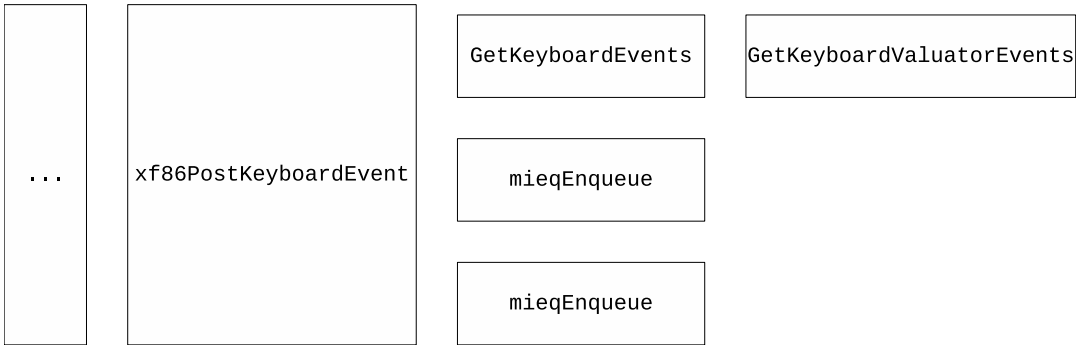

Abbildung 2.2: Beispiel Ablaufdiagramm

Mit Hilfe von Scripts ist die Automatisierung des GNU Debuggers möglich. So konnte ich automatisch zu jedem überwachten Funktionsaufruf eine Stacktrace abspeichern lassen. Nachdem die Blätter des Baums mit Hilfe der Diagramme identifiziert waren, konnten die nicht mehr benötigten Traces aussortiert werden.

An meinem Einstiegspunkt waren die Daten, welche das „a“ darstellten bereits im X-Server angekommen. Um herauszufinden, wo diese Daten herkamen und um die allgemeine Struktur (Hauptschleife) des X.Org-Servers zu verstehen ging ich zunächst den Stacktrace durch.

Aus der Untersuchung des Kernels konnte ich vermuten, dass die Eingabedaten in eine Gerätedatei geschrieben werden würden. Deshalb schrieb ich ein einfaches C-Programm, welches eine beliebige Datei auf dorthin geschriebene Events überwacht. Die Struktur der Events ging dabei aus dem Kernel-Quellcode hervor. Im Dateisystem gibt es mehrere Gerätedateien, in die Eingaben geschrieben werden können (z. B. `stdin`). Mit Hilfe des Programms konnte ich die richtige Datei für Tastaturevents und damit die entsprechende Kernel-Komponente (`evdev`) identifizieren.

Nachdem der Programmablauf in diese Richtung geklärt war, verfolgte ich die andere Richtung, also vom Einstiegspunkt aus bis zur Übergabe der Eingabe an den Client OpenOffice.org.

Um mich dabei nicht in Hilfsfunktionen, die für das Verständnis des Gesamtvorgangs nicht relevant waren, zu verlaufen, orientierte ich mich hauptsächlich am Datenfluss. Funktionen, denen das Eingabeereignis übergeben wurde, waren fast immer in irgendeiner Form für den Ablauf relevant.

2.2.3 OpenOffice.org

Da ich schon bei X.Org auf GDB als Debugger zurückgegriffen hatte, bot es sich an, diesen auch für OpenOffice.org zu verwenden. Hierzu mussten wieder beim Kompilieren die nötigen Debug-Symbole erstellt werden.

Aufgrund des Umfangs und der Komplexität des Quellcodes ist der Build-Vorgang bei OpenOffice.org nicht besonders leicht nachzuvollziehen. Hinzu kommt noch, dass sich dieser Vorgang häufig von Version zu Version ändert, wobei nicht immer alle Änderungen klar dokumentiert werden. Das Kompilieren dauerte wesentlich länger als beim Kernel, musste aber zum Glück nur einmal durchgeführt werden.

Anschließend konnte ich den in der Vorarbeit beschriebenen Programmablauf noch einmal selbst nachvollziehen.

2.2.4 Kommunikation mit X.Org

X.Org funktioniert als Server, er kommuniziert mit Clients (hier OpenOffice.org) über Netzwerk-Sockets. Für die genauere Betrachtung dieser Kommunikationsabläufe gab es mehrere Möglichkeiten[Kar09]:

- **Lesen der Daten vor dem Senden bzw. nach dem Empfangen**

  Ich entschied mich gegen diese Möglichkeit, da OpenOffice nicht direkt mit X kommuniziert sondern dazu externe Bibliotheken verwendet. Deshalb war davon auszugehen, dass die Daten vor dem tatsächlichen Versenden noch geändert werden würden.

  Um die Nachrichten von X an OpenOffice zu untersuchen wäre diese Technik allerdings völlig ausreichend gewesen.

- **Auslesen der Netzpakete**

  Da die Kommunikation zwischen Server und Client über Netzwerksockets erfolgt, wäre es möglich, die Pakete mit einem Sniffer auszulesen – es gibt jedoch eine komfortablere Möglichkeit.

- **xtrace**

  Dieses Programm lässt sich vor einen laufenden X-Server setzen, wobei es selbst einen X-Server vortäuscht. Nun führt man ein Programm auf dem vorgetäuschten X-Server aus. xtrace leitet alle Anfragen an den echten Server weiter und auch vom Server zurück an den Client; allerdings hat man dabei die Möglichkeit, alle ausgetauschten Nachrichten einzusehen.

Ich entschied mich dafür, xtrace zu verwenden. Aufgrund des geringen Dokumentationsumfangs und der Tatsache, dass beim Verwenden von OpenOffice.org immer mehrere Prozesse gestartet werden, gab es anfänglich einige Probleme damit, diese auf dem richtigen Server, nämlich den von xtrace vorgetäuschten, laufen zu lassen.

Danach konnte ich so aber die vollständige Kommunikation zwischen X.Org und OpenOffice.org einsehen. Diese stellte sich auch als sehr umfangreich heraus: So gingen allein für die Darstellung des Blinkens des Cursors ständig Anfragen beim X.Org-Server ein. Beim Drücken der Taste „a“ liefen nicht nur Anfragen zur Darstellung des Buchstabens ein, sondern es wurden auch viele weitere Anfragen ausgelöst (z. B. das Einfärben des Rückgängig-Knopfes, der aktivierbar wird). Das nächste Problem bestand also darin, die anfallenden Anfragen einzuschränken bzw. zu filtern.

Zunächst versuchte ich die Anfragen einzuschränken, indem ich den X.Org-Server mit dem Debugger anhielt. Ich wusste, dass ich die Eingabe auch im gestoppten Zustand absetzen konnte und diese dann verarbeitet werden würde, wenn ich den Ablauf fortsetzte. Jedoch kam xtrace mit dem angehaltenen X-Server nicht zurecht und reagierte nicht mehr.

Als Nächstes betrachtete ich solche Anfragen genauer, die vermutlich etwas mit grafischen Funktionen zu tun hatten, und stellte fest, dass diese immer Koordinaten enthielten. So konnte ich die X- und Y-Koordinaten an der Stelle des getippten Buchstaben verwenden, um Anfragen, die sich in diesem Bereich abspielten, herauszufiltern. So war es leicht, die für das „a“ verantwortlichen Nachrichten in dieser eingeschränkten Menge zu identifizieren.

Im Endeffekt stellte sich der Aufwand des Filterns als gerechtfertigt heraus, da ich so feststellen konnte, dass insgesamt drei Anfragen zur Übertragung und Darstellung des „a“ an den X-Server gingen.

Da es für jede Art von Anfrage im X-Server einen speziellen Handler gibt, hatte ich so auch gleich neue Einstiegspunkte für die Untersuchung der Ausgabe im X-Server.

2.2.5 X.Org - Ausgabe

Anhand des Namens der Anfrage, die von OpenOffice.org an den X-Server gerichtet wurde, konnte ich also im Code eine Funktion identifizieren, die an der Verarbeitung der grafischen Umsetzung beteiligt ist. Wie schon bei der Eingabe untersuchte ich den Ablauf von dort aus zunächst rückwärts. So gelangte ich nach kurzer Zeit wieder zu den bereits bekannten Funktionen der Hauptschleife.

Beim Untersuchen des Ablaufs in die andere Richtung stieß ich auf wesentlich mehr sich wiederholende und für mich unrelevante Funktionsaufrufe als bei der Eingabe. Durch die Diagramme ließen sich wiederholende Muster aber schon früh identifizieren und konnten dann übersprungen werden.

Ich stieß auf ein weiteres Problem: Wenn die Anfragen beim X-Server eingehen, beziehen sie sich auf spezifische Funktionen (z. B. zeichne eine Linie, zeichne ein Symbol) und lassen sich somit gut auseinanderhalten. Bei der Verarbeitung hingegen werden sie zu allgemeinen Paketen zusammengeschnürt, welche dann nur noch unspezifizierte grafische Veränderungen repräsentierten. Alle diese Pakete wurden dann intern in eine Schlange eingereiht und später abgearbeitet.

Ich musste also noch einmal das richtige Paket finden. Glücklicherweise konnte ich auch hier wieder auf die X- und Y-Koordinaten, an denen das „a“ erscheinen sollte, zurückgreifen.

2.2.6 Weitere Komponenten

Bei der Untersuchung der Programme stieß ich an vielen Stellen auf Programmbibliotheken, welche nicht Teil der untersuchten Software waren (z. B. verwendet der X-Server die Bibliothek Pixman zur Bildmanipulation).

Dies ist wenig überraschend, wenn man an die Entwicklung des X-Servers, des Linux-Kernels und der damit verbundenen Unix-Philosophie denkt. Diese betrachtet Modularisierung und Zusammenarbeit verschiedener Programme als wichtige Grundlagen der Softwareentwicklung [Ray03, I.1.]. Viele Open-Source-Programme nutzen diese Einstellung und die daraus hervorgegangenen Bibliotheken, um bestimmte Aufgaben auszulagern.

Leider war es zeitlich nicht möglich, diese Komponenten zu untersuchen. Zwar sind sie generell weniger umfangreich als die restlichen betrachteten Programm, jedoch macht die Tatsache, dass sie nur über Schnittstellen angesprochen werden, ihre Untersuchung im Kontext des Gesamtablaufs schwierig.

2.3 Ordnen der Daten

Um die Daten für die Präsentation zu strukturieren, entschied ich mich dafür, die Darstellungsformen des „a“ in den Mittelpunkt zu stellen.

Auf dem Weg vom Tastendruck zum Bildschirm werden die Daten, welche den Buchstaben repräsentieren, immer wieder unterschiedlich dargestellt und betrachtet (z. B. als Interrupt, Bild, usw.). So konnte ich den Software-Ablauf in Segmente teilen, die sich jeweils mit einem Format beschäftigten.

Diese Betrachtungsweise sollte zu jeder Darstellungsform die Fragen aufwerfen, warum gerade diese gewählt wurde und was mit den Daten in dieser Form gemacht wird. Das Beantworten dieser Fragen sollte dann als Leitfaden der Präsentation fungieren.

Die sich so ergebene Struktur lässt sich in Form eines Präsentationsentwurfs im Anhang B finden.

3 Ergebnis

Die folgende Darstellung der Ergebnisse ist stark gekürzt und vereinfacht und soll in erster Linie dem Überblick dienen. Die vollständige Fassung ist zusammen mit den Ergebnissen der Vorabeit im Anhang zu finden.

3.1 Überblick

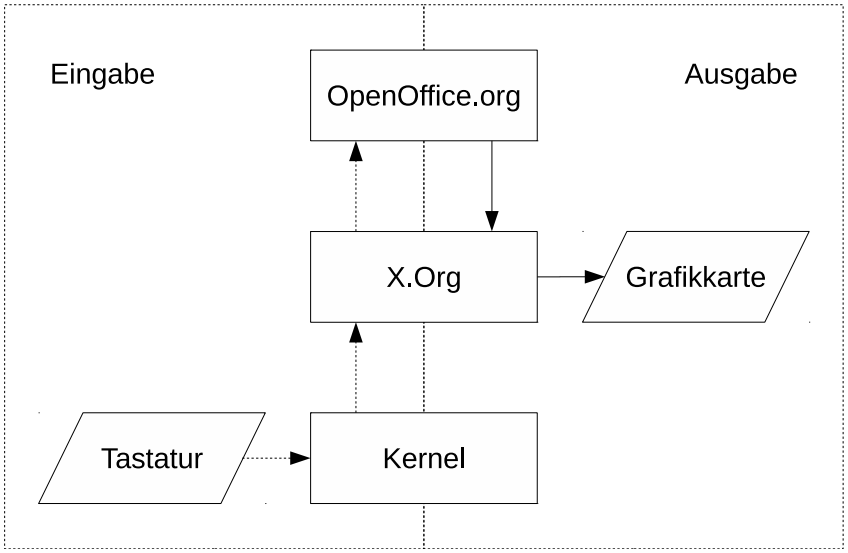

Abbildung 3.1: Schematischer Ablauf

Beschränken wir uns auf die Betrachtung der Software, so lässt sich der Ablauf folgendermaßen aufteilen:

- Eingabe: Kernel, X.Org, OpenOffice.org
- Ausgabe: OpenOffice.org, X.Org

„Eingabe“ bezieht sich dabei auf den Weg der Daten, die das „a“ repräsentieren vom Eintreffen beim Kernel bis zu dem Punkt, an dem der Buchstabe Teil des OpenOffice.org-Dokuments wird. „Ausgabe“ setzt an eben diesem Punkt an und endet mit dem Schreiben der „a“-Pixel in den Speicher der Grafikkarte.

3.2 Eingabe

Alles beginnt mit dem Drücken der „A“-Taste auf der Tastatur. Der Code der gedrückten Taste wird in der Tastatur gespeichert und ein Interrupt-Request (IRQ) wird an den Computer gesendet.

Der Kernel ordnet den verschiedenen IRQs spezifische Handler zu. Der für Tastatureingaben zuständige Handler wird aufgerufen; dieser weiß, in welchem Format der Tastendruck gespeichert wurde, und liest diese Daten aus.

Aus diesen rohen Daten wird ein Objekt (Eingabe-Event) erstellt, welches neben dem Code der gedrückten Taste z. B. auch den Zeitpunkt des Events enthält. Dieses Event wird dann wieder an die entsprechenden Handler weitergegeben.

Einer dieser Handler ist evdev: Hier wird das Eingabe-Event nochmals umgewandelt – diesmal in ein generisches Event-Format. Durch evdev wird jedem Eingabegerät eine Gerätedatei zugeordnet; das Event wird in den Puffer der Gerätedatei geschrieben. Zu einem späteren Zeitpunkt wird der Puffer geleert und in die Datei geschrieben.

X.Org arbeitet als Server, die Clients sind dabei die Anwendungen wie OpenOffice.org. Der Server stellt die Verbindung zwischen den Clients (z. B. um Drag and Drop zwischen verschiedenen Anwendungen zu realisieren) und zwischen den Clients und der Hardware her. Server und Clients müssen nicht auf dem gleichen System laufen, die Kommunikation erfolgt über Netzwerk-Sockets.

Der X-Server durchläuft ständig eine Hauptschleife in der anfallende Ereignisse abgearbeitet werden. Liegen keine Ereignisse vor, geht er in einen Ruhezustand über.

Im hier betrachteten Fall wird X.Org vom Kernel benachrichtigt, dass neue Daten in der Gerätedatei der Tastatur vorliegen. X.Org verlässt den Ruhezustand und liest das Eingabe-Event, welches von evdev in eine verständliche Form gebracht wurde, ein.

Bevor das Event an einen Client weitergeleitet wird, müssen zunächst folgende Umstände überprüft werden:

- Handelt es sich um einen Sonderfall, der gar nicht zu einem Client weitergeleitet werden muss (z. B. Sondertasten wie Caps Lock)?
- Welcher Client war zu dem im Event hinterlegten Zeitpunkt aktiv?
- Hat der Client sich für das Empfangen der vorliegenden Art von Event angemeldet?

Das „a“ ist kein Sonderfall; OpenOffice.org-Writer war zum Zeitpunkt des Events aktiv und möchte auch Tastendruck-Events empfangen. Das Event wird also dem Client zugewiesen, in eine interne Schlange für ausgehende Events eingereiht und zu einem späteren Zeitpunkt verschickt. Dazu wird das Event in das zum Client gehörende Netzwerk-Socket geschrieben.

Wie der X-Server durchläuft auch OpenOffice.org eine Hauptschleife, in der überprüft wird, ob neue Ereignisse vorliegen. Da OpenOffice.org auf einer Vielzahl von Betriebssystemen verwendet werden kann, gibt es viele verschiedene Möglichkeiten, diese Ereignisse

zu empfangen. Unter Linux wird dafür auf Xlib zurückgegriffen, eine Bibliothek, welche die Kommunikation mit dem X-Server ermöglicht.

OpenOffice.org wandelt das Event jetzt noch einmal in ein internes Format um und überprüft, wie es zu behandeln ist (z. B. ob es eine Tastenkombination ist, die eine Funktion wie Speichern auslöst). Hier soll das „a“ nun Teil des Dokuments werden: Es wird also dementsprechend in dessen interne Repräsentation eingefügt und mögliche Änderungen werden berechnet.

3.3 Ausgabe

Zum Darstellen von Text verwendet OpenOffice.org die freie Grafikbibliothek Cairo. Cairo ist unabhängig vom verwendeten grafischen System und kann z. B. auch unter Windows verwendet werden.

Es werden also die von Cairo bereitgestellten Funktionen zum Darstellen von Symbolen aufgerufen. Cairo stellt dann die entsprechenden Anfragen an den X-Server.

Dem X-Server stehen alle darstellbaren Symbole als Bilder im Dateisystem zur Verfügung. Das Bild für den Buchstaben „a“ wird geladen, auf die benötigte Größe skaliert und mit der richtigen Farbe versehen. Nun wird das „a“ über das Bild des aktuellen Bildschirms gelegt. In diesem überlagerten Bereich werden die Unterschiede der beiden Bilder berechnet. Das Bild des Buchstabens ist zwar rechteckig, enthält jedoch viele transparente Pixel – an diesen Stellen wird der Bildschirminhalt also nicht verändert.

Als Nächstes werden die berechneten Bildänderungen Pixel für Pixel in eine interne Kopie des Bildschirminhalts geschrieben. Wenn mehrere Anfragen für grafische Operationen vorliegen, werden diese also alle zunächst auf die Kopie des Bildschirminhalts angewandt. Erst zu einem späteren Zeitpunkt wird der wirkliche Bildschirminhalt an die Kopie angepasst. Bei sich überlappenden grafischen Vorgängen muss so weniger auf den Speicher der Grafikkarte zugegriffen werden. Wie überhaupt mit der Grafikkarte zu kommunizieren ist, wird vorher beim Starten des X-Servers durch den Grafikkartentreiber festgelegt.

Bevor der X-Server wieder in den Ruhezustand übergeht, um auf neue Ereignisse, Anfragen, usw. zu warten, wird der Bildschirminhalt, der im Speicher der Grafikkarte liegt, an die lokal veränderte Version angepasst. Vom Speicher der Grafikkarte wird der Bildschirminhalt je nach eingestellter Bildwiederholfrequenz (z. B. 50 Hz) an den Bildschirm gesendet und ist damit für den Benutzer sichtbar.

4 Zusammenfassung

Wenngleich es mir gelungen ist, die Lücken in der Ablaufbeschreibung der Vorarbeit zu füllen, gibt es immer noch einige Bereiche, in denen genauere Untersuchungen für den Gesamtablauf sehr interessant sein könnten – zum Beispiel die folgenden Fragestellungen:

- Wie verwaltet OpenOffice Schriftarten bzw. wo kommen diese her?
- Was würde sich beim Einsatz anderer Grafikkarten-Treiber ändern?
- Wie würde ein Fenstermanager den Ablauf beeinflussen?
- Was genau passiert in externen Bibliotheken wie Cairo oder Pixman?
- Was geschieht im Bildschirm?

Außerdem habe ich es leider nicht geschafft, den Ablauf als Präsentation aufzubereiten. Die nun vorhandenen Daten sind meines Erachtens aber ausreichend, um eine solche zu erstellen.

Während der Bearbeitung habe ich auch an andere Arten der Datenaufbereitung gedacht, die zu interessanten Ergebnissen führen könnten. Es wäre z. B. eine Visualisierung als interaktives Baumdiagramm auf Grundlage des Kontrollflusses möglich. Durch Auswahl der einzelnen Komponenten könnte der Betrachter den Detailgrad der angezeigten Abläufe anpassen.

Anhang A

Ausführliches Ergebnis

Dieser Anhang enthält die ausführliche Fassung der von mir erarbeiten Ergebnisse sowie die relevanten Kapitel der Vorarbeit. Diese Abschnitte (A.1, A.2, A.4) wurden der Bachelorarbeit zum gleichen Thema von Franz Zieris aus dem Jahr 2009 entnommen.

A.1 Hardware

Der gesamte Abschnitt A.1 wurde [ZIE09, S.9 ff.] entnommen.

A.1.1 Koordination der Eingaben

Generell gibt es zwei Möglichkeiten für Computer Eingaben zu koordinieren. Die erste besteht darin, dass die entsprechenden Geräte in kurzen Zeitabständen zyklisch angefragt werden, ob sie neue Eingaben zu liefern haben (das sog. *Polling*[1]). Dies ist allerdings sehr aufwändig, da der Computer ja nicht wissen kann, welche Eingabegeräte tatsächlich neue Daten hinterlegt haben und somit alle Geräte immer und immer wieder anfragen muss, was sich letztlich negativ auf die Leistungsfähigkeit des Gesamtsystems auswirken kann. [Güb04, S. 44ff]

Die zweite Möglichkeit ist wesentlich eleganter. Sie besteht darin, sich von dem betreffenden Gerät einfach Bescheid geben zu lassen. Das ist das Konzept der Interrupts[2], mit dem in fast allen Fällen auch Tastatureingaben bearbeitet werden. Der Name erklärt sich dadurch, dass der Computer (genauer: der Prozessor) nicht wissen kann, zu welchem Zeitpunkt ein solcher Interrupt auftritt und dadurch gewissermaßen in seiner eigentlichen Arbeit unterbrochen wird. [Güb04, S. 46f]

Die Geräte kommunizieren dabei nicht unmittelbar mit der CPU. Jedes Gerät wird in einer bestimmten Art und Weise an einen *Hardware Controller* angeschlossen, der die Kommunikation intern regelt. Unabhängig von der Verfahrensweise der Statusabfrage müssen für eine Interaktion auch Daten fließen. Dafür sind den einzelnen Controllern

[1] engl. Abfrage
[2] engl. Unterbrechnung

systemintern sog. *Ports* zugewiesen, über die der Prozessor lesend und/oder schreibend auf bestimmte *Register*, also Speicherplätze zugreifen kann.

A.1.2 Hardwareseitige Behandlung von Interrupts

A.1.2.1 Auslösen des Interrupts

Ein Interrupt kann z.B. von einem Hardware-Controller ausgelöst werden, indem er einen *Interupt ReQuest* (IRQ) sendet. Dafür hat jeder interruptfähige Controller eine Verbindung zu einem *Programmable Interrupt Controller* (PIC), der die IRQs verschiedener Geräte bündelt. Registriert der PIC an einer der IRQ-Lines einen Signalanstieg, so wird die Nummer dieses IRQs in einen *Interrupt Vector* (einer Art Kennnummer für Interrupts) umgewandelt. Standardmäßig wird dem IRQ n der Vektor $n + 32$ zugewiesen. Diese Zuordnung ist aber über entsprechende Anweisungen an den PIC änderbar.

Dieser Vektor wird in einem I/O-Port des PIC gespeichert, auf den der Prozessor des Computers zugreifen kann. Wenn mehrere IRQ-Lines einen höheren Pegel aufweisen, wird diejenige mit der niedrigeren Nummer gewählt. Danach erhöht der PIC die Spannung am `INTR`-Pin der CPU, wodurch diese über den Interrupt informiert wird. Diese Spannung wird solange gehalten, bis die CPU in einen I/O-Port des PIC schreibt - danach wird der `INTR`-Pin wieder „gecleart“. [BC06, S. 134f]

Jede der IRQ-Lines kann auch einzeln deaktiviert und auch wieder aktiviert werden. Während IRQs deaktiviert sind, werden diese dennoch im PIC gespeichert und bei Wiederaktivierung an die CPU geschickt. Die CPU kann dem PIC somit mitteilen, dass sie erstmal eine Weile Ruhe haben möchte, ohne dass dabei Interrupts verloren gehen. [BC06, S. 135]

Ergänzend zum programmierten Deaktivieren einzelner IRQs im PIC, kann die CPU auch sog. maskierbare Interrupts[3] ignorieren, wenn ein entsprechendes Flag [4] in der CPU nicht gesetzt ist. [INTEL99, S. 3-68f], [INTEL99, S. 3-664f], [INTEL09, S. 3-21ff]

Hardwareseitig wird der PIC üblicherweise durch 8259A- oder kompatible Chips realisiert, die hintereinander geschaltet werden (*Cascade Mode*). Jeder dieser Chips kann acht IRQs behandeln, sodass bis zu 64 IRQ-Lines verwaltet werden können, wenn ein Master und acht Slaves zusammenschaltet werden. Dabei belegt der Ausgang eines Slaves einen Eingang des Masters [5]. [INTEL88, S. 19]

[3] I/O-Interrupts sind auch maskierbar

[4] `IF` - Interrupt Enable Flag, Bit 9 im Register `EFLAGS`. Zum Setzen und Löschen dieses Flags gibt es entprechende Assembler-Befehle: `cli` - clear interrupt flag und `sti` - set interrupt flag.

[5] Das ist auch der Grund dafür, warum die IRQs üblicherweise die Reihenfolge 0, 1, **8, 9, . . . , 15**, 3, 4, . . . , 7 haben. Die acht IRQs vom Slave werden an IRQ 2 des Master gemultiplext, wodurch diese scheinbar nicht-lineare Hierarchie entsteht.

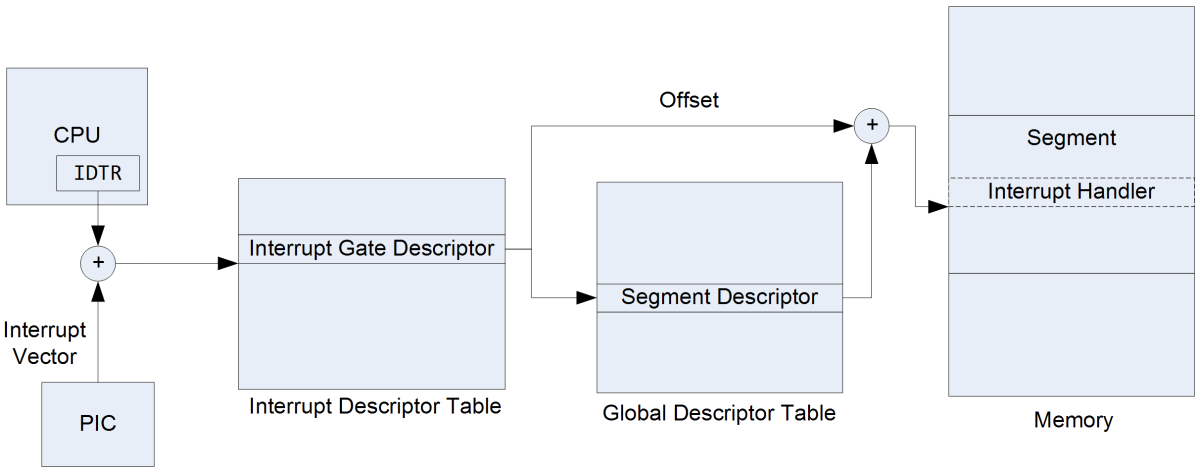

Abbildung A.1: Ausführung eines Interrupt-Handlers auf Grundlage eines Interrupt-Vektors

A.1.2.2 Rezeption des Interrupts durch die CPU

Nachdem die CPU eine Anweisung ausgeführt hat, stehen in den Registern `CS` und `EIP` der CPU die logische Adresse der nächsten Anweisung. Bevor diese allerdings ausgeführt wird, prüft die CPU, ob ein Interrupt (Pin `INTR`) anliegt.

Wenn dem so ist, wird zunächst der Interrupt-Vektor ermittelt, indem aus dem I/O-Port des PIC gelesen wird. Nun muss die CPU den für diesen Interrupt passenden Code finden und ausführen. Im Speicher abgelegt gibt es dafür eine sog. *Interrupt Descriptor Table* IDT. Diese verknüpft jeden Interrupt-Vektor mit der Adresse des entsprechenden Interrupt-Handlers. Ein Eintrag in dieser Tabelle wird *Interrupt Gate Descriptor* genannt. Dieser enthäl u.a. einen *Segment Selector* und einen *Offset* vom Anfang des angesprochenen Speicher-Segments.

Nachdem der passende Interrupt Gate Descriptor gefunden wurde, werden einige Überprüfungen durchgeführt, um z.B. sicherzustellen, dass keine Nutzeranwendung den Code eines Interrupt-Handlers ausführen kann. Dies soll hier aber nicht weiter vertieft werden.

Um später mit dem Programmablauf fortfahren zu können, den der Interrupt unterbrochen hat, wird der Inhalt der Register `SS`, `ESP`, `EFLAGS`, `CS` und `EIP` auf dem Stack gespeichert. Das Stapeln der Werte ermöglicht z.B. ein einfaches Verschachteln von Interrupt-Handlern, wobei der Status des unterbrochenem Interrupt-Handlers genauso wie der eines normalen Programmes gesichert wird.

Der Segment Selector aus dem Interrupt Gate Descriptor verweist auf einen Eintrag in der *Global Descriptor Table* GDT, in der in Form von *Segment Descriptors* hinterlegt ist, welches logische Speichersegment an welcher linearen Adresse liegt. Diese Adresse wird in das `CS`- und in das `EIP`-Register der CPU geladen. Die nächste echte Instruktion der CPU wird also die erste Anweisung des Interrupt-Handlers sein. Der gesamte Ablauf der Berechnung der linearen Adresse des Interrupt-Handlers aus dem Interrupt-Vektor

ist in Abbildung A.1 dargestellt. [BC06, S. 140]

Die letzte Anweisung eines Interrupt-Handlers muss der Assemblerbefehl `IRET` sein, der die CPU dazu veranlasst, die gesicherten Werte von `CS` und `EIP` vom Stack wieder in ihre Register zu laden, um mit dem ursprünglichen Ablauf fortfahren zu können.

A.1.3 Interrupt eines Tastendruckes

Ein Tastendruck wird von einem in der Tastatur integrierten Mikrocontroller registriert, welcher einen sog. *Make-Code* spezifisch für die gedrückte Taste erzeugt. Beim Loslassen der Taste wird analog dazu ein *Break-Code* erzeugt[6]. Die Zuordnung der physischen Tasten zu je einem Make- und Break-Code wird in sog. *Scan Code Sets* festgelegt. Das von mir verwendete Keyboard benutzte das Set 1, in dem der A-Taste ein Make-Code von `0x1E` und ein Break-Code von `0x9E` zugewiesen ist. [Cha04]

Nachdem der Mikrocontroller im Keyboard den jeweiligen Make- bzw. Break-Code ermittelt hat, sendet er diesen an den Hardware-Controller im Computer, der diesen in einem Register speichert. Danach sendet der Hardware-Controller einen IRQ an den PIC. Da die Tastatur normalerweise die IRQ-Line 1 belegt, haben Tastatur-Interrupts den Vektor $1 + 32 = 33$ bzw. `0x21` in hexadezimaler Schreibweise.

Nach dem oben beschrieben Ablauf führt die CPU den Code aus, der vom 33. Eintrag der IDT referenziert wird. Die Organisation der IDT (und damit die Gesamtorganisation der Interrupt-Behandlung) sowie die spezifische Ausprägung der Interrupt Handler ist Aufgabe des Betriebssystems.

Für den obligatorischen Datenfluss vom Hardware-Controller zur CPU innerhalb der Interrupt-Behandlung hat der Keyboard-Controller zwei Ports, auf die die CPU zugreifen kann: `0x60` und `0x64`. Hinter diesen Ports verbergen sich allerdings drei relevante Register: Input, Output und Status. Port `0x60` bietet lesenden Zugriff auf den Input und schreibenden Zugriff auf den Output, Port `0x64` bietet lesenden Zugriff auf das Status-Register und schreibenden Zugriff, um Befehle zu senden.

A.2 Kernel

Die Abschnitte A.2.1, A.2.2, A.2.3 und A.2.4 wurden [ZIE09, S.16 ff.] entnommen.

A.2.1 Aufgaben des Kernels

Der Kernel eines Betriebssystems hat zwei Hauptaufgaben zu erfüllen:

1. Verwaltung der Hardware

[6] Mit Ausnahme der „Pause“- bzw. „Break“-Taste, diese hat ironischerweise nur einen Make-Code

2. Bereitstellen einer Umgebung, um Nutzer-Programme ausführen zu können

Linux trennt diese Bereich recht strikt und erlaubt den laufenden Anwendungen im Normalfall keinen direkten Zugriff auf die Hardware. Diese Trennung ist nur sinnvoll umsetzbar, wenn auch die Hardware ein entsprechendes Konzept bietet. Die Hardware eines Computers setzt das mit verschiedenen Ausführungsmodi der CPU um. Linux nutzt diese Fähigkeit, um zwischen *Kernel Mode*, also dem uneingeschränkten Zugriff auf alle Ressourcen und dem *User Mode*, der einige Speicherbereiche und Befehle sperrt, zu unterscheiden[7]. [BC06, S. 8]

A.2.2 Aufbau und Strukturierung des Kernels

Der Linux-Kernel hat an sich eine sog. monolithische Strukur, das heißt, dass das gesamte Betriebssystem ein großes Programm ist, welches komplett im Kernel Mode läuft. Im Gegensatz dazu beschränken sich Betriebssysteme mit dem Aufbau eines Microkernels auf das Allernötigste (Speicher- und Prozessverwaltung, sowie die Interprozess-Kommunikation) – alle anderen Bestandteile sind quasi Nutzerprogramme.

Allerdings gibt es bei Linux eine entscheidene Einschränkung des monolithischen Konzepts: da es selten Sinn macht, einen omnipotenten Kernel auszuführen, der sich mit jeder möglichen Hardware verständigen kann, werden viele sog. Module nicht statisch, sondern im laufenden Betrieb und nur bei Bedarf in den Kernel eingebunden. Ich erwähne das, weil einige Strukturen, auf die ich später im Text zu sprechen kommen werde, auf diesem Mechanismus aufbauen. [BC06, S. 3]

Der Großteil des Quellcodes des Linux-Kernels ist komplett unabhängig von der Rechnerarchitektur, weil die Designer darauf geachtet haben, die Unterschiede, die verschiedene Bauformen bieten in einer möglichst dünnen Schicht zu verbergen. Innerhalb dieser Schicht beginnt auch die grundlegende Interruptbehandlung.

A.2.3 Anforderungen an die Interrupt-Bearbeitung unter Linux

Die Bearbeitung eines Interrupts erfolgt im sog. *Kernel Mode*, was bedeutet, dass einige kritische Maschinenbefehle, die sonst nicht ausgeführt werden können, erlaubt sind und dass auf einem anderen Speicherbereich als im sog. *User Mode* gearbeitet wird. Während der Abarbeitung eines Interrupts findet kein Kontextwechsel statt, d.h. dass u.a. nur auf dem Kernel-Stack gearbeitet wird. Das hat eine entscheidende Konsequenz: Interrupt-Handler dürfen nicht blockieren, indem sie z.B. auf die Reaktion eines Gerätes warten. Das Blockieren eines Prozesses würde den Scheduler auf den Plan rufen, der einen Kontextwechsel zu einem anderen Prozess vornehmen würde.

[7] Die von mir betrachtete x86-Architektur bietet ingesamt vier Abstufungen, von denen Linux aber nur Stufe 0 und Stufe 3 benutzten. Der Modus der CPU ist im Register `CPL` *Current Privilege Level* als 2-Bit-Wert gespeichert

Im Linux-Kernel ist aber durchaus vorgesehen, dass Interrupts einander unterbrechen können. Zum einen, um den Durchsatz des PICs und der Hardware Controller zu erhöhen. Wenn die CPU gerade mit der Ausführung eines Interrupt-Handlers beschäftigt ist und keine Unterbrechungen erlaubt wären, würden PIC und die Hardware Controller so lange blockieren, bis die CPU mit der Abarbeitung fertig ist. Wenn aber die Verschachtelung von Interrupts erlaubt wird, kann die CPU sofort auf den IRQ reagieren und die neuen Informationen verarbeiten. Zum anderen ist dadurch das Interrupt-Modell sehr einfach, weil es keine Prioritäten benötigt. Jeder Interrupt kann jeden anderen unterbrechen und es müssen im Kernel keine Hierarchien verwaltet werden, was sich letztendlich auch auf die Ausführungsgeschwindigkeit auswirkt. [BC06, S. 144f]

Eine direkte Kommunikation des Tastatur-Interrupt-Handlers mit der Office-Anwendung ist damit ausgeschlossen, weil diese eine abwechselnde Ausführung mit einem Prozess aus dem User Space und damit eine Unterbrechung des Interrupt-Handlers bedeuten würde. Die Aufgabe des Interrupt-Handlers besteht also darin, die empfangenen Daten für andere Prozesse möglichst effizient vorzubereiten und sich dann zu beenden.

A.2.4 Implementierung der Interruptbehandlung in Linux

Die erste Aufgabe der Interruptbehandlung besteht darin, aus dem übergebenen Interrupt-Vektor den passenden Interrupt-Handler zu finden, was zu großen Teilen schon die CPU selbst erledigt. Dem Linux-Kernel bleibt es aber überlassen, welche konkreten Anweisungen sich hinter den Einträgen der IDT verbergen. Die Implementierung der Interrupt-Behandlung von Linux habe ich in folgende Abschnitte untergliedert:

Vereinheitlichung der Interrupt-Requests und Verteilung auf die Interrupt-Deskriptoren Verschiedene Architekturen bereiten das Betriebssystem auf unterschiedliche Arten auf die Interrupt-Behandlung vor. Allein schon die zu sichernden Register und die Befehle zum Sichern differieren zwischen verschiedenen Bauarten. In dieser Schicht sollen diese Unteschiede versteckt werden und ein allgemeiner Zugriff auf Interrupts angeboten werden.

Abarbeiten der Action Chain Für jeden IRQ können mehrere Aktionen vorgesehen werden, die nacheinander ausgeführt werden (*Action Chain*). Diese Operation erfolgt unabhängig von der Art des auslösenden Hardware-Controllers.

Daten aus der Hardware auslesen Dieser Schritt ist wiederum spezifisch für die einzelnen Hardware-Controller. Geräte, die zwar unterschiedliche IRQs belegen, aber über den gleichen Hardware-Controller kommunizieren, werden hier zusammengefasst. Das ist z.B. bei Maus und Tastatur der Fall, da sie beide das PS/2-Protokoll benutzen. Das Auslesen des Status und der Daten erfolgt in dieser Schicht. Die rohen Daten werden dann an die nächsthöhere Schicht weitergegeben. Soweit möglich sollen hier auch Fehlerzustände der Hardware korrigiert oder zumindest sinnvoll bearbeitet werden.

Erstellen von Events In der nächsten Ebene beschäftigen sich die Funktionen nicht mehr mit Protokoll-Einzelheiten (bestimmte Statusmeldungen, etc.), sondern greift

auf die bereits gelesenen und vorverarbeiteten Daten zu. Die Aufgabe dieser Schicht besteht nun darin, unabhängig vom konkreten Gerätetyp eine einheitliche Form für Input-Events zu bieten.

Zustellen der Events an die Empfänger In der nächsten Schicht werden die empfangen Input-Events von allen möglichen Geräten gesammelt. Neben dem „natürlichen“ Weg eines Inputs, gibt es auch die Möglichkeit, Input-Events künstlich zu erzeugen (*Input Injection*). Unabhängig von der tatsächlichen Herkunft wird hier auf Grundlage der Event-Typen und ihrer Ausprägung entschieden, ob und wie sie weitergereicht werden. Event-Handler können sich in dieser Schicht anmelden, um z.B. bei Tastatur-Events aufgerufen zu werden. Auch hier gibt es wieder die Möglichkeit mehrere Handler für einen Event-Typ anzumelden, die der Reihe nach aufgerufen werden.

Behandlung der Events In der darüberliegenden Schicht siedeln die Event-Handler, so z.B. der Keyboard-Handler, dessen Aufgabe darin besteht, die Eingaben von der Tastatur so lange zu speichern, bis sie abgefragt werden.

Dieser Aufbau und Übersichsgrafiken über den gesamten Ablauf der Tastatur-Interrupt-Behandlung finden sich schematisiert im Anhang A.

A.2.4.1 Vereinheitlichung der Interrupt-Requests und Verteilung auf die Interrupt-Deskriptoren

Von der IDT referenzierte Anweisungen

Im Linux-Kernel sind alle Interrupt-Handler zu Beginn gleich aufgebaut. Der Interrupt-Handler für den `IRQ n` speichert zunächst den Wert ¬`n` auf dem Stack[8]. Durch diese Auslagerung der Informationen über den Interrupt kann nun einheitlich für alle Interrupts der gleiche Code ausgeführt werden, der sich beim Label `common_interrupt` befindet.

Dort werden alle CPU-Register, die von Interrupt-Handlern verändert werden könnten, auf den Stack gesichert (ausgenommen `EFLAGS`, `CS`, `EIP`, `SS` und `ESP`, die bereits von der CPU selbst gesichert wurden). Nach diesen Vorbereitungen wird die Funktion `do_IRQ()` aufgerufen.

do_IRQ()

Die `do_IRQ()`-Funktion erwartet einen Parameter in Form eines Structs. Dieser wird wie bei Parameterübergaben üblich auf dem Stack erwartet. Beim Zugriff auf ein Feld des übergebenen Parameters wird daher entsprechend weit im Stack nach unten gegangen. So kann und wird auf die Position zugegriffen, an der vorhin der bitweise negierte Wert für den IRQ abgelegt wurde.

[8] Linux stellt die IRQ-Werte intern negativ dar, weil die positiven Werte für Syscalls (also Systemaufrufe aus Nutzerprogrammen) reserviert sind.

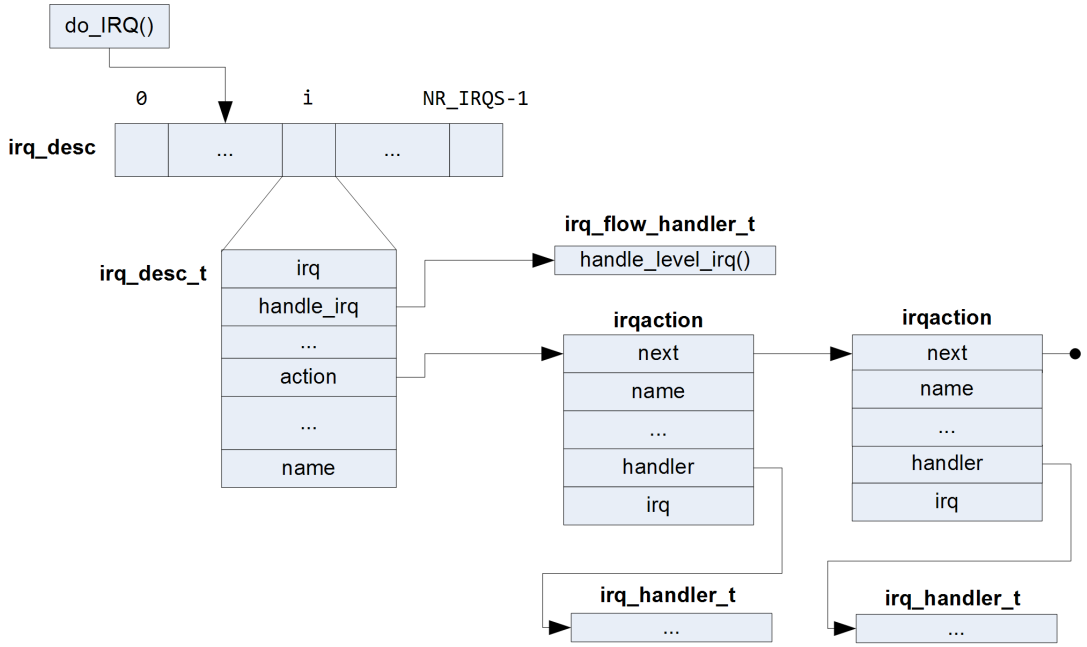

Abbildung A.2: Zugriff der Funktion `do_IRQ()` auf die IRQ-Strukturen

Nun wird aus dem statischen Array `irq_desc`, der für jeden IRQ einen sog. Deskriptor parat hält, der passende Eintrag ausgewählt und auf diesem die `handle_irq`-Funktion aufgerufen, hinter der sich eine Funktion vom Typ `irq_flow_handler_t`, im Normalfall die Funktion `handle_level_irq()` verbirgt (siehe Abbildung A.2).

A.2.4.2 Abarbeiten der Action Chain

handle_level_irq() und handle_IRQ_event()

`handle_level_irq()` maskiert den Interrupt (es werden von diesem Gerät also keine neuen Interrupts angenommen und auch nicht zwischengespeichert) führt dann die Funktion `handle_IRQ_event()` aus und nimmt die Maskierung des Interrupts wieder zurück, sobald der Interrupt-Handler eine Arbeit abgeschlossen hat. Die Funktion `handle_IRQ_event()` ruft im Prinzip nur nacheinander die im jeweiligen IRQ-Descriptor assoziierten IRQ-Actions auf. Durch die Anmeldung mehrerer Actions im IRQ-Deskriptor ist es also möglich, verschiedene Gerätertreiber auf den gleichen IRQ lauschen zu lassen.

Mit der Auswahl des passenden `irq_desc_t`-Objekts[9] wird der architekturspezifische Teil der Interruptbehandlung verlassen. Alle Implementierungen der grundlegenden Interrupt-Behandlung greifen auf das `irq_desc`-Array zu.

Die Actions, die für Tastatur und Maus zuständig sind, werden durch das Laden des

[9]Ich verwende den Begriff „Objekt“ hier im Sinne von „Konkrete Ausprägung eines Structs“

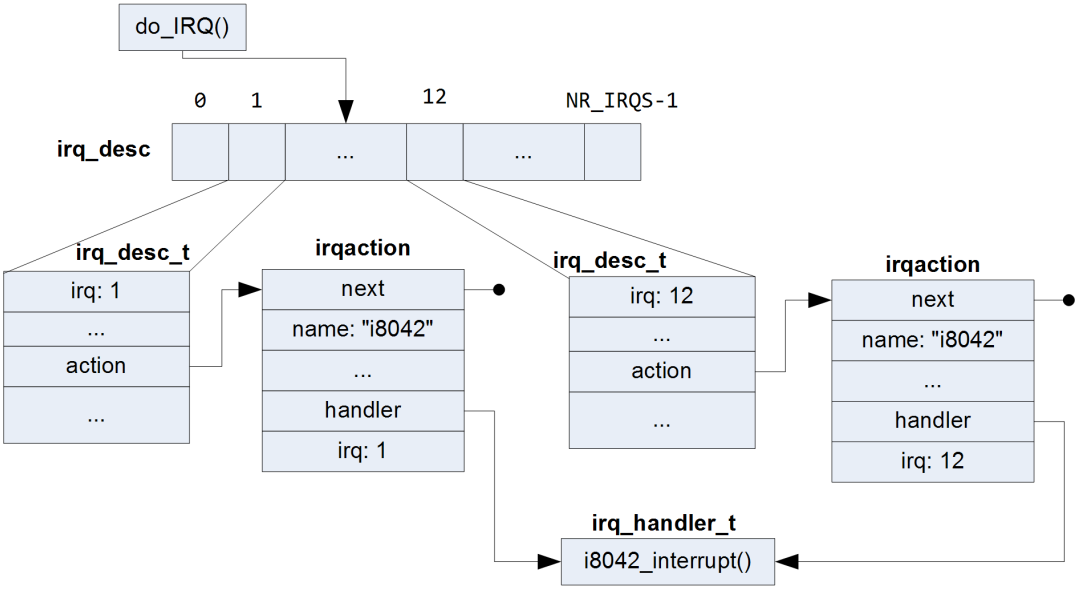

Abbildung A.3: Verknüpfung von IRQ und Interrupt Service Routine bei Keyboard und Maus

Kernel-Moduls `i8042` angelegt [10]. Der genaue Ablauf beim Laden des Kernel-Moduls und Anmelden von Treibern und IRQ-Handlern, soll hier nicht beleuchtet werden. Die dabei entstehenden Strukturen sind in Abbildung A.3 skizziert. Bemerkenswert ist, dass beide Actions den gleichen IRQ-Handler referenzieren: `i8042_interrupt()`. Das ist einfach zu erklären, denn schließlich benutzen Maus und Tastatur beide die PS/2-Schnittstelle und kommunizieren über den i8042-Controller.

A.2.4.3 Daten aus der Hardware auslesen

Die i8042_interrupt()-Funktion

Die erste Amtshandlung dieser Funktion besteht darin, den Status des Hardware-Controllers ausgelesen. Dafür gibt es die Hilfsfunktion `i8042_read_status()`, die sich über einen Assembler-Befehl ein Byte vom internen Port `0x64` liefern lässt. Im Falle des gedrückten „A“ ist das das Byte `0x1D`.

Vom gelesenen Byte interessieren uns primär zwei Bits:

OBF (Bit 0) Ist es gesetzt, ist der Output-Puffer gefüllt und es kann gelesen werden. Andernfalls braucht der Interrupt-Handler nichts weiter tun und wird beendet.

[10] Dabei wird die Funktion `i8042_init()` aufgerufen, welche u.a. eine Struktur `i8042_driver` erstellt und diese an die Funktion `platform_driver_register()` übergibt. Die `probe`-Funktion dieser Struktur ist `i8042_probe()`. Beim Aufruf eben dieser wird neben einem Selbsttest und einigen Initialisierungsaufgaben, erst die Maus eingerichtet (`i8042_setup_aux()`) und dann die Tastatur (`i8042_setup_kbd()`). Beide Funktionen nutzen dabei die Routine `request_irq()`, um sich für `IRQ12` (Maus) bzw. `IRQ1` (Keyboard) anzumelden.

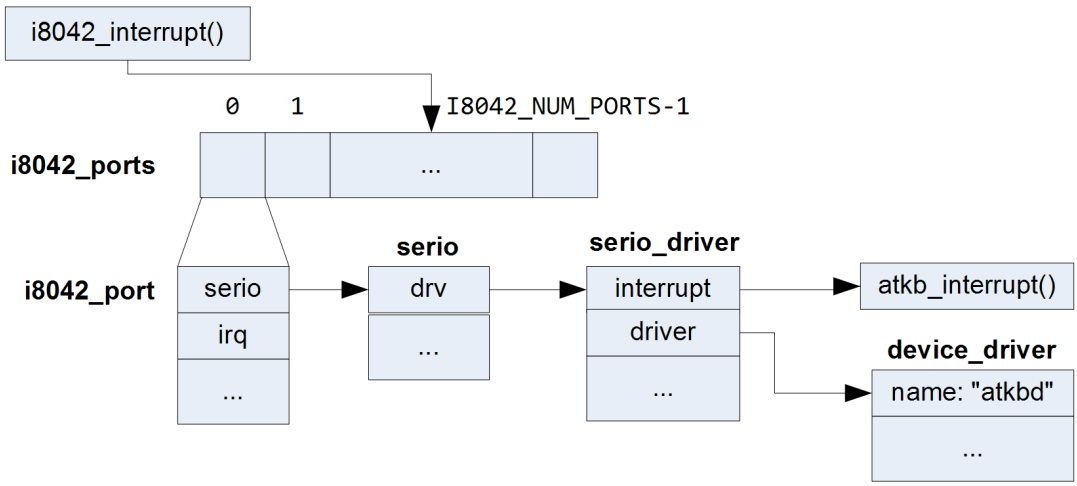

Abbildung A.4: Schema der `i8042_ports` mit exemplarischer Darstellung des Keyboard-Ports

AUXDATA (Bit 5) Wenn OBF gesetzt ist, dann steht hier, woher die Daten kommen. 0 sind Keyboard-Daten, 1 sind Maus-Daten.

Der Status `0x1D` (00011101) heißt also: es gibt Daten im Buffer und sie stammen vom Keyboard.

Anschließend werden die Daten aus dem Input-Register gelesen, wofür die Hilfsfunktion `i8042_read_data()` bereitsteht, welche ein Byte von Port `0x60` liest. Wurde das „A“ gerade gedrückt, so wird das Byte `0x1E` gelesen, was gerade dem Make-Code der A-Taste gemäß Scan Code Set 1 entspricht.

Da der i8042-Controller Keyboard und Maus bedient, muss hier für die korrekte Weitergabe der Daten anhand des `AUXDATA`-Bits zwischen zwei Wegen gewählt werden. Diese Wege werden `i8042_port` genannt und alle verfügbaren dieser Art werden in dem Array `i8042_ports` gespeichert. Ein solcher `i8042_port` beinhaltet u.a. einen Zeiger auf ein `serio`-Objekt[11].

Dieses `serio`-Objekt wird der Funktion `serio_interrupt()` zusammen mit den von Port `0x60` gelesenen Daten übergeben.

Die serio_interrupt()-Funktion

Diese Funktion ruft das `interrupt`-Feld des in der `serio`-Struktur enthaltenen `drv`-Feldes auf (siehe Abbildung A.4). Dabei werden die `serio`-Struktur und die gelesenen Daten als Parameter übergeben.

Die Registrierung der entsprechenden `serio_driver`-Struktur, durch die die Verbindung mit der Funktion `atkb_interrupt()` hergestellt wird, geschieht bei der Initialisierung

[11] Ich vermute dass der Begriff `serio` von seriell kommt, und sich die im Kernel verwendeten Strukturen mit diesen Namen alle auf serielle Schnittstellen beziehen. Validieren konnte ich das aber nicht.

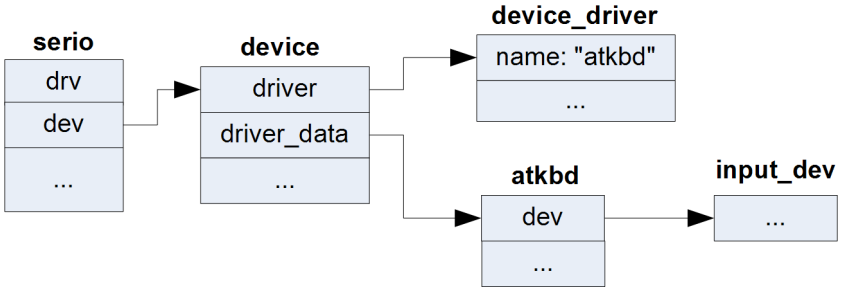

Abbildung A.5: Zugriff auf das `input_dev`-Objekt über das `serio`-Objekt

des Moduls `atkbd`. In der `atkbd_init()`-Funktion wird u.a. die Routine `serio_register_driver()` mit dem vorbereiteten `serio_driver`-Objekt aufgerufen.

Interrupts, die vom Keyboard kommen, werden also über den entsprechenden `i8042_port` an die Funktion `atkbd_interrupt()` verwiesen.

A.2.4.4 Erstellen von Events

Die atkbd_interrupt()-Funktion

Die Hauptaufgabe der `atkbd_interrupt()`-Funktion ist es, die empfangenen Daten in Events umzuwandeln. Diese Umwandelung geschieht in mehreren Schritten. Höhere Schichten, die die Events verarbeiten, werden in mehreren Durchgängen über das Geschehen in der Hardware informiert. Diese Information erfolgt über die Funktion `input_event()`, welche von Eingabe-Geräten aller Art benutzt wird, um Events bekannt zu geben. Diese Schnittstelle stellt bereits die dritte Vereinheitlichung der Inputs dar – die erste erfolgte im PIC, die zweite über do_IRQ(). Für eben diese einheitliche Behandlung sind folgende Informationen wichtig:

- Welches Gerät hat das Event erzeugt?
- Um welche Art von Event handelt es sich?
- Was sind die spezifischen Eigenschaften dieses Events? Um diese Eigenschaften strukturiert ausdrücken zu können, wird hierbei zwischen dem Event-*Code* und dem Event-*Wert* unterschieden.

Bevor jedoch Daten ausgelesen und verarbeitet werden können, verschafft sich die Funktion erst einmal Zugriff auf das `input_dev`-Objekt, das das Keyboard intern repräsentiert. Dafür werden die Treiberdaten aus dem mit dem `serio`-Objekt verknüpften Device-Objekt benutzt. In diesen Treiberdaten steht auch das verwendete Gerät, welches in den folgenden Funktionen als „Absender“ der Events angegeben wird (siehe A.5).

- Das erste so generierte Event ist vom Typ `EV_MSC` (also Kategorie Sonstiges), hat den Code `MSC_RAW` (d.h. es werden rohe, unbearbeitete Daten geliefert), und den Wert `code`, der gerade dem von Port `0x60` gelesen Daten entspricht. Beim `A` wäre das also der Wert `0x1E`.

- Danach werden die rohen Daten bei Bedarf[12] umgewandelt und das nächste Event gesendet. Dieses ist wieder ein `EV_MSC`-Event, hat aber diesmal den Event-Code `MSC_SCAN` (d.h. es wird der Scancode übertragen) und besitzt als Wert die umgewandelten Daten. Beim `A` wurde hier nicht viel geändert, es wird wieder der Wert `0x1E` gesendet.
- Der dritte Durchgang erfolgt als `EV_KEY`-Event, also als Tastaturevent. Der Event-Code ist der Keycode, der abhängig von der Keycode-Tabelle der `atkbd`-Struktur aus dem Scancode ermittelt wird; der Event-Wert gibt Auskunft darüber, ob die Taste gedrückt (Wert `1`), losgelassen (`0`) oder gedrückt gehalten wurde (`2`). Die Keycode-Tabelle der verwendeten Tastatur hat für das `A` wieder den gleichen Wert `0x1E` ausgegeben. Dieser wird mit dem Wert `1` verschickt.
- Der vierte und letzte Durchgang wird über die Hilfsfunktion `input_sync()` abgewickelt. Diese Funktion erzeugt ein Event vom Typ `EV_SYN` mit dem Code `SYN_REPORT` und dem Wert `0`.

A.2.4.5 Zustellen der Events an die Empfänger

Die input_event()-Funktion

Die `input_event()`-Funktion erfüllt u.a. die Aufgabe eines Türstehers. Sie reicht die an sie übergebenen Events nur weiter, wenn diese auch unterstützt werden. Dies ist nicht der Fall, wenn der Event-Code größer als der maximal für Event-Codes vorgesehene Wert `EV_MAX` ist (derzeit bei `0x1F`, also 31) oder wenn das Gerät den übergebenen Event-Typ gar nicht unterstützt (z.B. ein Sound-Event `EV_SND` bei einer Maus). Für diesen Zweck haben `input_dev`-Objekte eine Bitmap, die die verfügbaren Event-Typen kodiert (siehe Abbildung A.6).

Ist das Event unterstützt, wird es an die Funktion `input_handle_event()` weitergereicht.

Interessant ist bei dieser Funktion noch, dass sie sämtliche unterstütze Eingaben benutzt, um damit ein Rauschen für Zufallszahlen zu generieren, welche z.B. für sicherere SSL-Schlüssel genutzt werden. Um das Einpflegen dieser Daten samt Ignorierung von wiederholten Eingaben kümmert die Hilfsfunktion `add_input_randomness()`.

Die input_handle_event()-Funktion

Die `input_handle_event()`-Funktion agiert in erster Linie als Dispatcher, sie verteilt also die eben gebündelten Events wieder auf ihre jeweiligen Bestimmungsorte. Für diesen Zweck wird der übergebene Event-Typ benutzt. Er entscheidet darüber, wie mit dem Event verfahren wird und an wen es weitergereicht wird. Nachfolgend gehe ich nur auf die Event-Typen `EV_MSC` und `EV_KEY` ein und vernachlässige die Synchronisationsevents.

[12] Der Bedarf entsteht auf Grund von speziellen Eigenschaften der `atkbd`-Struktur, z.B. wenn das Keyboard das Scan Set 3 benutzt, oder sonst irgendeine Übersetzung verlangt.

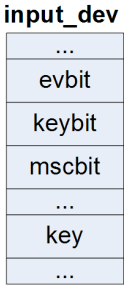

Abbildung A.6: Auswahl der Bitmaps

EVMSC Bei diesen eher unspezifischen Events wird einfach nur geprüft, ob das sendende Gerät dieses Event unterstützt. Im Gegensatz zu der vorhergehenden Kontrolle in der Funktion `input_event()`, bei der der Event-*Typ* überprüft wurde, wird hier der Event-*Code* kontrolliert. Hierfür hat jedes `input_dev`-Objekt eine spezielle Bitmap für `MSC`-Events (`mscbit`, siehe Abbildung A.6). Sollte das Event ungültig sein, so wird das Event ignoriert und die Funktion ohne weitere Aktionen beendet. Handelt es sich um ein gültiges Event, so wird dieses am Ende der Funktion an die registrierten Handler geschickt.

EVKEY Key-Events werden auf gleiche Weise geprüft, um auch hier auszuschließen, dass vom Gerät gar nicht unterstützte Events verarbeitet werden (Bitmap `keybit`, Abbildung A.6). Weiterhin wird aber noch verlangt, dass sich der Status des Keys geändert hat: nur wenn der gelieferte Wert (beim Drücken also `1`) nicht dem bisherigen Wert der Taste entspricht (also Taste war nicht gedrückt) wird das Event weiter gereicht. Wenn keine Änderung stattgefunden hat, so wird das Event ignoriert. Hat sich der Status geändert, so wird der neue Wert in der entsprechenden Bitmap des `input_dev`-Objektes (Bitmap `key`, Abbildung A.6) gespeichert und zum Abschluss die registrierten Handler benachrichtigt.

Wenn die Taste gedrückt wurde (Wert `1`), dann wird die Hilfsfunktion `input_start_autorepeat()` aufgerufen. Linux verlässt sich nicht auf die Fähigkeiten von Keyboards, wiederholte Zeichen schicken zu können, sondern kümmert sich selbst um die Wiederholung [13].

Ein Spezialfall stellt dabei der Wert `2` dar, der geliefert wird, wenn die Taste gedrückt gehalten wird und das Keyboard dies auch mitteilt. Hierbei wird die Bitmap `key` nicht verändert und die Wiederholung wird auch nicht vom Kernel selbst bewerkstelligt.

Die etwaig vorher festgelegte Benachrichtigung der Handler wird mit der Funktion `input_pass_event()` ausgeführt.

[13] Das ist ein kleines Beispiel für eine Grundeinstellung von Linux: Es muss damit gerechnet werden, dass die Hardware fehlerhaft oder unspezifiert ist, daher müssen entsprechende Vorbereitungen getroffen werden. Genauso soll sich die Hardware aber auch „beweisen“ können: Wenn sie offensichtlich ein Feature anbietet, dann soll dieses auch genutzt werden.

Die input_pass_event()-Funktion

In der `input_pass_event()`-Funktion wird nun zwischen zwei Fällen unterschieden.

1. Es gibt ein Input-Handle, das für das betreffende Gerät einen alleinigen Anspruch angemeldet hat (*grab*).
   In diesem Fall werden alle Input-Events ausschließlich zu diesem Handle gereicht, indem auf dem `handler` des Handles die `event`-Funktion aufgerufen wird (siehe Abbildung A.7).
2. Es gibt kein solches Handle.
   In diesem Fall werden alle assoziierten und offenen Handles benachrichtigt. Die Gesamtmenge der Handles wird als doppelt verkettete Liste gespeichert, deren Anfang über `h_list` erreichbar ist. Physisch handelt es sich bei diesen `list_head`-Objekten auch um `input_handle`-Objekte (siehe Abbildung A.8)

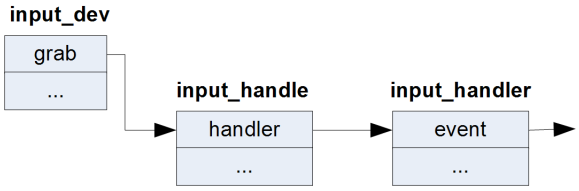

Abbildung A.7: Zugriff auf ein Handle, das das Device für sich allein beansprucht

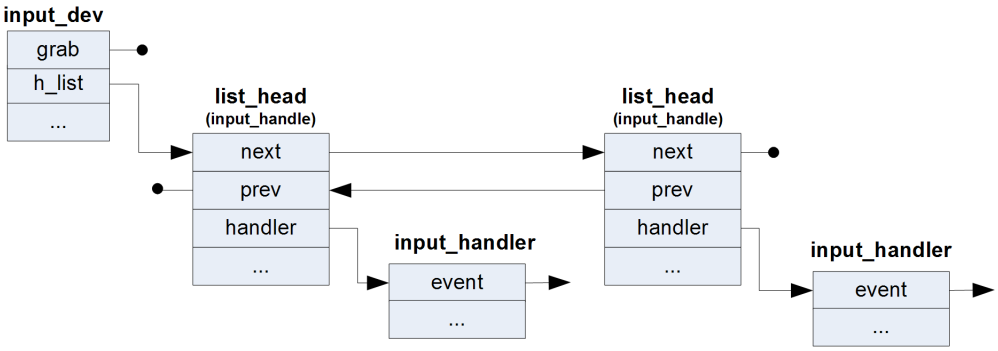

Abbildung A.8: Zugriff auf mehrere Handles eines Devices

Hier wird nun in der Vorarbeit die weitere Behandlung des Events durch den Handler `kbd` beschrieben. Für den weiteren Ablauf ist dieser jedoch nicht relevant, weshalb des Rest des Abschnitts hier ausgelassen wird. Stattdessen folgen meine eigenen Beobachtungen zum Ablauf.

A.2.5 Behandlung der Events

Nachdem zur Tastatureingabe ein Event erstellt wurde, wird dieses an verschiedene Handler weitergeleitet. Für die Weiterleitung der Eingabe an den X.Org-Server ist dabei der `evdev`-Handler verantwortlich.

`Evdev` (event device) ist eine Komponente des Kernels, die verschiedene Eingabe-Events (z. B. Maus, Tastatur, Joystick usw.) in ein generisches Event-Objekt umwandelt, welches vom X.Org-Server verstanden werden kann.

Es wird die `event`-Funktion des `evdev`-Handlers aufgerufen.

evdev_event()

Hier werden alle ankommenden Events an alle `evdev`-Clients weitergereicht. Hinter einem `evdev`-Client verbirgt sich nichts weiter als ein `event`-Puffer und ein dazugehöriges Eingabegerät mit einem Verweis auf dessen Gerätedatei (Unter Linux ist eine Gerätedatei eine Schnittstelle zu einem Gerätetreiber, die wie eine Datei behandelt wird).

In unserem Fall gibt es nur einen Client; dort wird das Ereignis in den Puffer geschrieben.

Nach dem Interrupt

Zu einem späteren Zeitpunkt wird der Puffer des `evdev`-Clients geleert und in die entsprechende Gerätedatei geschrieben; für die Tastatur ist diese unter `/dev/input/event0` zu finden. Dort können die Events nun vom X-Server ausgelesen werden.

A.3 X.Org-Server - Eingabe

Um die Abläufe im X.Org-Server besser zu verstehen, gibt es einige Aspekte, die man sich stets vor Augen halten sollte:

- Client-Server-Architektur[XMAN],
- X.Org ist eine Implementation des X-Window-Systems[XMAN],
- Eingabegeräte werden in Gruppen abstrahiert[XORGb].

An erster Stelle steht die Client-Server-Architektur des Programms. Der X.Org-Server ist so ausgelegt, dass Benutzerprogramme und Server auf verschiedenen Geräten laufen können; die Eingabegeräte sind jedoch immer am Server angeschlossen. Die Kommunikation zwischen Client und Server erfolgt über Netzwerk-Sockets. Vom Server aus werden an die Clients Events verschickt, die wie hier z. B. einen Tastendruck repräsentieren (Abbildung A.9).

Dabei können die Clients vorher festlegen, welche Art von Events sie überhaupt erhalten möchten (z. B. KeyPress, KeyRelease, ButtonPress, ButtonRelease, PointerMotion, Button1Motion usw.). Vom Client werden an den Server Anfragen geschickt, eine besondere Event-Art, die z. B. das Darstellen eines grafischen Elements auf dem Bildschirm erbittet.

Ebenso sollte man beachten, dass es sich bei X in erster Linie um ein unabhängiges Protokoll handelt und wir eine spezifische Implementation untersuchen (in diesem Fall

X.Org Server für Linux). So kommt es also beim Übergang zwischen rein internen und auch von außen zugänglichen Komponenten zu Umwandlungen zwischen systemspezifischen und allgemeinen Datenformaten.

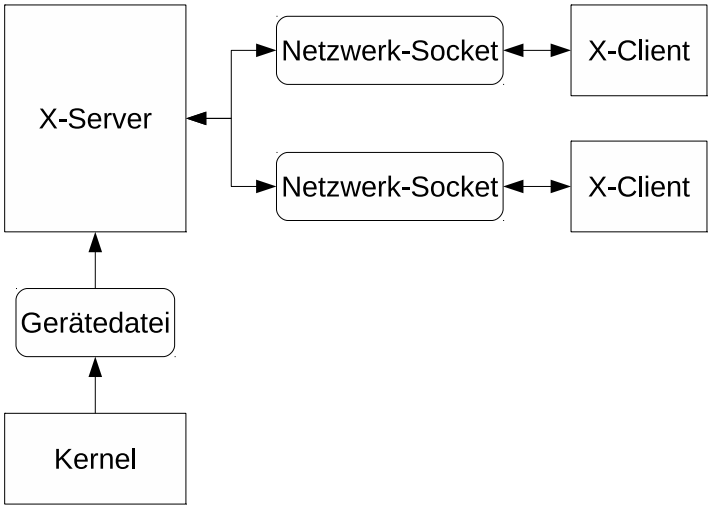

Abbildung A.9: Client-Server-Architektur von X

Speziell für die Tastatureingabe ist es außerdem noch wichtig zu wissen, dass X auch für den Umgang mit mehreren Tastaturen ausgelegt ist und alle Gruppen von Eingabegeräten (z. B. Mäuse oder Tastaturen) hinter virtuellen Geräten versteckt werden. In älteren Versionen wurde auch noch ein Eingabegerät einer Klasse als Hauptgerät (core device) hervorgehoben.

Eine weitere Besonderheit beim Umgang mit Eingabegeräten ist die Möglichkeit, diese per Grab-Befehl an einen Client zu binden. Dabei wird zwischen Core- und Device-Grabs unterschieden. Bei einem Core-Grab erhält ein Client nur Events des gebundenen Geräts und bei einem Device-Grab erhält kein anderer Client Events des Geräts.

Was den allgemeinen Programmablauf betrifft, sind vor allem die Methoden `Dispatch` und `WaitForSomething` relevant. `Dispatch` ist die Hauptschleife des X.Org-Servers, `WaitForSomething` wird von hier aufgerufen um auf neue Events zu warten.

In dem hier betrachteten Fall lässt sich die vom X.Org-Server verrichtete Arbeit beim Verarbeiten der Tastatureingabe in zwei Abschnitte einteilen[XORGb]:

- Erstellen eines internen Eingabe-Events
- Ausliefern des Events an Clients

A.3.1 Event-Erstellung

In diesem Schritt wandelt der X.Org-Server das vom Kernel aus anfallende Eingabe-Ereignis in ein internes X-Event um, welches dann in eine interne Queue für Events und Nachrichten eingereiht wird.

WaitForSomething

Falls die interne Event-Queue leer ist – also keine Events vorliegen, die an Clients verschickt werden sollen – wartet der X.Org-Server hier auf neue Ereignisse. Dabei ist der Programmablauf unterbrochen. Ereignisse, bei denen der Ablauf fortgesetzt wird, sind:

- Anfragen eines Clients
- Eingehende Eingabe-Events
- Andere Geräte-Events

Im Falle einer Client-Anfrage liegt diese im entsprechenden Netzwerk-Socket vor, im Falle eines Events in der Gerätedatei des Eingabegeräts (hier `/dev/input/event0`).

Realisiert wird dieser Wartevorgang durch den Systemaufruf `select(...)`. Diesem werden zu beobachtende Datei-Handle, z. B. für die oben genannte Gerätedatei, übergeben. Stellt der Kernel dann Änderungen an diesen Dateien fest, wird der wartende Prozess geweckt.

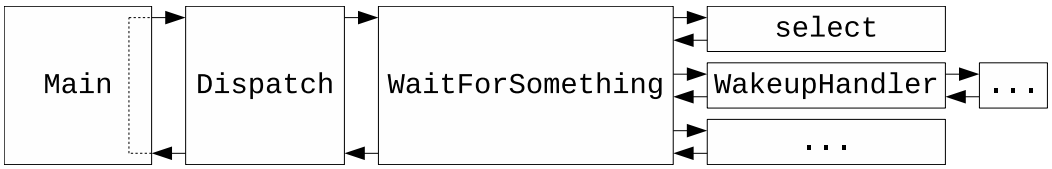

Abbildung A.10: Hauptschleife X-Server

Anschließend wird der allgemeine Wakeup-Handler mit `WakeupHandler(...)` aufgerufen, der dann alle zuvor angemeldeten Wakeup-Handler aufruft. Hier gibt es nur einen Wakeup-Handler, `xf86Wakeup`.

xf86Wakeup

Hier wird nun festgestellt, um was für ein Event es sich handelt (Eingabe-Event) und von welchem Gerät dieses stammt (Tastatur). Dann wird die entsprechende Methode zum Lesen der Eingabe aufgerufen. Dabei verlassen wir den geräteunabhängigen Teil des X-Servers und betreten den Tastaturtreiber, der als austauschbares Modul des X-Servers vorliegt.

stdReadInput

Das Eingabe-Event wird gelesen, also der Scancode und die Art des Events (key up oder key down). Anschließend werden die Informationen weiterverarbeitet.

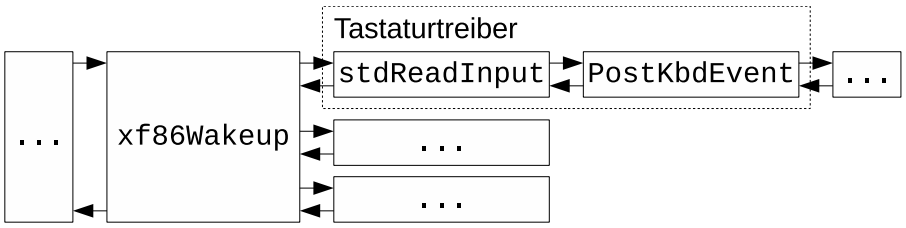

Abbildung A.11: Funktionen des Tastaturtreibers

PostKbdEvent

In dieser Methode wird zunächst überprüft, ob die gedrückte Taste gesondert behandelt werden muss (z. B. Num, Caps-Lock, usw.).

Bei einer normalen Taste wird nun aus dem Scancode (hier z. B. 30 für das „a“) nach der im X.Org-Server eingestellten Keymap (welche Tasten haben welche Bedeutung, z. B. QWERTZ, Dvorak) ein entsprechender X-Keycode erstellt (hier z. B. 38).

xf86PostKeyboardEvent

Wir verlassen den Tastaturtreiber und sind wieder im geräteunabhängigen Teil des X.Org-Server. Hier werden nun die folgenden Methoden aufgerufen, welche die zusammengetragenen Daten in die interne Form für Events bringen und das Event in die interne Queue einreihen.

GetKeyboardEvents

Die Daten der Tastatureingabe werden jetzt in die interne Darstellungsform für Ereignisse gebracht (`xEvent`).

Desweiteren wird hier das Gedrückthalten von Tasten und das damit einhergehende Wiederholen von Events behandelt. Es ist anzumerken, dass der gesamte Code, der sich damit beschäftigt, aufgrund des möglichen Vorhandenseins mehrerer Eingabegeräte sehr komplex ist.

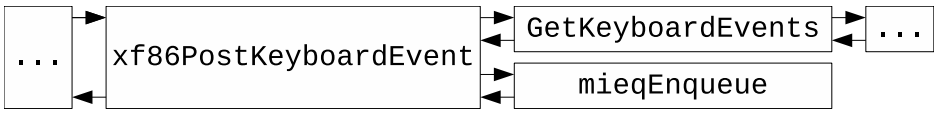

Abbildung A.12: Einreihen des Events

mieqEnqueue

Das oft im Quellcode auftretende Kürzel `mieq` steht für „machine independent event queue“. Die eben erstellten Events werden nun in genau diese interne Event-Queue des X.Org-Servers eingereiht.

Damit ist die Event-Erstellung abgeschlossen. Vor der Rückkehr in die Methode `WaitForSomething` wird nur noch die Gerätedatei, aus der das Eingabe-Event gelesen wurde, freigegeben. An dieser Stelle wird noch der Timer für den Bildschirmschoner zurückgesetzt.

A.3.2 Event-Auslieferung

Nach Abschluss der Event-Erstellung wird `WaitForSomething` verlassen und zur Hauptschleife von X zurückgekehrt.

Bevor das Event tatsächlich in das Netzwerk-Socket des Clients geschrieben werden kann, müssen zunächst diverse Sonderfälle herausgefiltert werden. Außerdem wird überprüft, ob der Client diese Art von Event überhaupt erhalten möchte.

Dispatch

In der Funktion `Dispatch` werden wartende Ereignisse und Nachrichten verschickt und verarbeitet. Von hier aus wurde auch `WaitForSomething` aufgerufen, nachdem alle vorhandenen Ereignisse abgearbeitet waren.

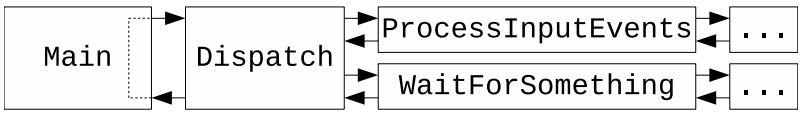

Abbildung A.13: Hauputschelife X-Server

Nachdem im letzten Schritt ein Event in die interne Queue eingestellt wurde, wird nun die entsprechende Funktion zum Verarbeiten aufgerufen.

ProcessInputEvents

Bevor die Event-Queue durchlaufen wird, werden hier die internen Variablen für Caps-, Num- und Scroll-Lock angepasst, falls diese sich verändern.

mieqProcessInputEvents

Hier werden nun alle Events durchgegangen und verarbeitet. Welche Funktionen dabei aufgerufen werden, ist durch die Art des Events (Maus oder Tastatur) und dem damit verbundenen Gerät vorgegeben.

ProcessKeyboardEvent

Jetzt wird zunächst überprüft, ob es sich auch um ein Key-Event handelt, also eine gedrückte oder losgelassene Taste.

Die vorherigen Methoden waren auf die Verarbeitung von abstrakten Events ausgelegt, an dieser Stelle greift nun die erste Komponente zum Filtern, die aufgrund der Art des Events (Tastatureingabe) aufgerufen wird, X Accessibility.

AccessXFilterPressEvent

In dieser Funktion werden verschiedene Hilfen zum barrierefreien Arbeiten umgesetzt. Dazu gehören z. B. das Verzögern eines Tastendruck-Events, bis die Taste eine gewisse Zeit gehalten wurde, oder dass eine Taste wie Shift nicht gedrückt gehalten werden muss, sondern automatisch zusammen mit dem nächsten eingehenden Tastendruck verarbeitet wird.

Da in unserem Fall keine dieser Optionen aktiviert ist, wird das Event weitergereicht.

XkbProcessKeyboardEvent

In dieser Methode werden noch weitere Sonderfälle wie z. B. Eingabegeräte mit physikalisch feststellbaren Tasten behandelt; auch hier ist im betrachteten Fall keine Sonderbehandlung notwendig.

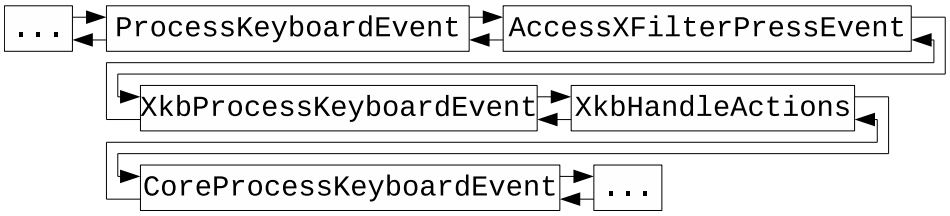

Abbildung A.14: Filterkette

XkbHandleActions

Im nächsten Schritt werden weitere Sonderfälle herausgefiltert, in diesem Fall die sogenannten Actions, die sich auf den X.Org-Server selbst beziehen. Es könnte z. B. eine Tastenkombination zum Wechseln des angezeigten X Bildschirms eingegeben werden oder auch Aktionen des Fenstermanagers ausgelöst werden.

CoreProcessKeyboardEvent

Während sich die letzten Funktionen noch mit dem Event bezogen auf das tatsächliche Eingabegerät befasst haben, wird es jetzt nur noch als abstraktes Core Event betrachtet - d. h. als ein Event, welches vom virtuellen Core Keyboard, das alle Tastaturen hinter einer gemeinsamen Abstraktion vereint, ausgelöst wurde.

Durch Aufrufen von `CheckDeviceGrabs(DeviceIntPtr device, ...)` wird festgestellt, ob das Eingabegerät momentan per Grab-Befehl an ein Programm gebunden ist. Wäre dies der Fall, müsste das Event nicht weiter bearbeitet werden, da das gebundene Programm bereits alle Events direkt erhält.

Es liegen keine Grabs vor, also geht es weiter mit der Auslieferung. Damit die nächste Funktion den korrekten Empfänger des Events feststellen kann, wird auf den Struct `sprite` zurückgegriffen, der abstrakte Informationen über den aktuellen Bildschirminhalt enthält, darunter auch das aktuell aktive Fenster.

DeliverFocusedEvent

Nun wird festgestellt, ob es derzeit einen Client (also ein Programm) gibt, welches den Fokus für die Tastatureingabe hat. Sollte dies nicht der Fall sein, wird das Event zum Root-Fenster geleitet. Das Root-Fenster gehört zu keinem Client und umfasst den gesamten sichtbaren Bildschirm.

Da der Fokus für Tastatur und Maus unterschiedlich zugeteilt werden kann, ist diese Funktion noch spezifisch für Tastatur-Events; ab der nächsten Funktion ist die Auslieferung für Maus- und Tastatureingaben jedoch identisch.

DeliverDeviceEvents

Durch Aufrufen von `FixUpEventFromWindow(...)` werden bestimmte Eigenschaften des Events an das entsprechende Fenster angepasst, z. B. werden die X- und Y-Koordinaten des Events von Bildschirmkoordinaten auf Fensterkoordinaten umgerechnet.

Anschließend kommt es zur eigentlichen Auslieferung.

Abbildung A.15: Event-Auslieferung

DeliverEventsToWindow

Dieser Funktion wird eine Liste von Events übergeben, wobei in unserem Fall diese Liste immer nur ein Event enthält (Drücken oder Loslassen der Taste).

Vor der Auslieferung wird hier überprüft, ob das Fenster überhaupt Events empfangen will. Sollte dies nicht der Fall sein, wird die Auslieferung hier abgebrochen.

Die Funktion versucht nun alle Events auszuliefern. Sollte ein Client nicht erreichbar sein, wird das Event verworfen.

TryClientEvents

Bevor das Event tatsächlich in den Netzwerk-Socket geschrieben wird, überprüft X noch, ob der Client überhaupt Events dieser Art erhalten möchte. Dies wird über `Mask mask` und `Mask filter` realisiert. `mask` enthält die Art der Events, die der Client erhalten möchte (z. B. nur Events, die von der ersten Maustaste ausgelöst wurden oder nur Tastendruck-Events) und `filter` enthält die Event-Art des aktuellen Ereignisses.

WriteEventsToClient

Sollte man seinen X.Org-Server mit mehreren physikalischen Bildschirmen betreiben und so mehrere Clients haben, die das gleiche Event erhalten sollen, wird dies hier umgesetzt.

Schließlich wird jetzt mit `WriteToClient(pClient, count * sizeof(xEvent), (char *) events)` das Eingabe-Event in das Netzwerk-Socket des Client geschrieben.

Damit ist die Auslieferung des Events abgeschlossen.

A.4 OpenOffice.org

Die Abschnitte A.4.1, A.4.2, A.4.3 und A.4.4 wurden [ZIE09, S. 30 ff.] entnommen.

A.4.1 Anforderungen an OpenOffice.org

Der OpenOffice.org Writer soll auf verschiedenen Betriebssystemen funktionieren, darunter auch auf einem Linux-System unter Verwendung des X Window Systems. Die Eingaben, die aus diesem System „hochgereicht werden“, sollen empfangen und verwertet werden können.

Verschiedene Office-Anwendungen (Writer, Calc, Impress, etc.) sollen einen ähnlichen Aufbau besitzen und möglichst große Programmteile gemeinsam benutzen können. Weiterhin sollen sich die konkreten Anwendungen (eben Writer, Calc, etc.) nicht mit der

Verbindung zum Betriebssystem auseinander setzen müssen. Diese Anwendungen sollen also neutrale Eingaben erhalten, dann ihre spezifischen Arbeiten verrichten und dann neutrale Ausgaben erzeugen.

Die Betrachtung dieser Anforderungen machen den Grundaufbau einer OpenOffice-Anwendung schnell verständlich. Zwei wesentliche Komponenten sind hierbei zu unterscheiden: der sog. *Visual Component Layer* VCL und die eigentliche Office-Anwendung, z.B. Writer. Der VCL bietet der eigentlichen Anwendung genau die gewünschte Abstraktion vom Betriebssystem und der Ausgabegrafik.

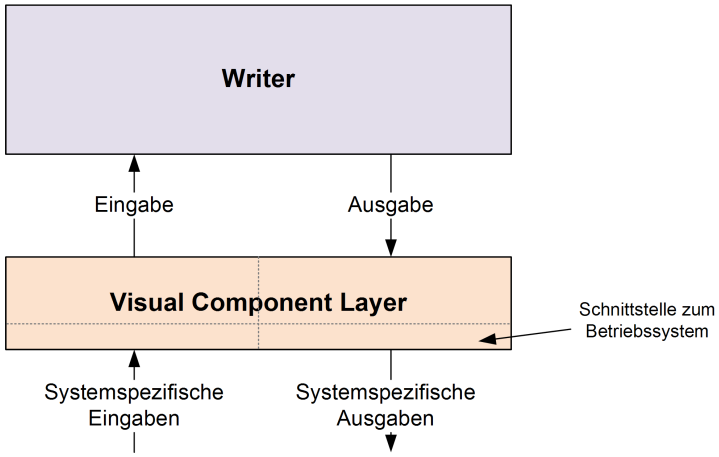

Abbildung A.16: Grobe Struktur von OpenOffice.org Writer

A.4.2 Visual Component Layer (Eingabe)

Um interaktive Anwendungen „am Leben zuerhalten“ wird standardmäßig das Konzept des Event Loop oder Ereignisschleife verwendet. Der Event Loop fragt Ereignisquellen ab und verteilt die Events auf die zuständigen Funktionen (Schema in Abbildung A.17). [XLIBa]

Der Programmteil, der den Bereich der Eingaben abdeckt, muss also auch den Event Loop beeinhalten. Letzterer lässt sich relativ unabhängig vom verwendeten Betriebssystem verwenden. Anders verhält es sich beim Abfragen der Ereignisse, da diese ja auf unterschiedliche Arten dargeboten werden können. Für die Linux-Fassung von OpenOffice wird ein Zugriff auf das X Window System benötigt, bei der Windows-Fassung ein Zugriff auf Win32-API, etc. Die Zuordnung der so gewonnenen Events zu den verarbeitenden Funktionen kann wiederrum losgelöst vom Betriebssystem erfolgen.

A.4.2.1 Der Event Loop

Der ausführende Programmteil des Event Loop ist die statischen Methode `Application::Execute()`, welche in einer Schleife die ebenfalls statische Methode `Application::Yield()` aufruft, die bei jedem Durchlauf folgende Aktionen ausführt:

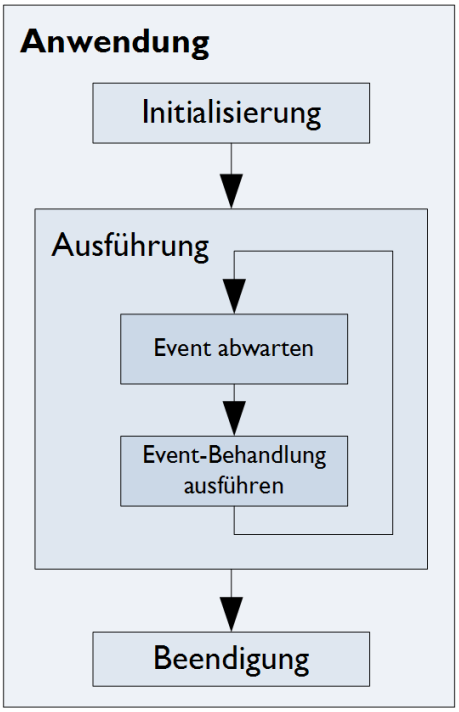

Abbildung A.17: Schema einer Interaktiven Anwendung

1. Alle Timer aufrufen, die seit dem letzten Schleifendurchgang abgelaufen sind.
   So werden Aktionen ausgeführt, die nicht bei jeder einzelnen Nutzerinteraktion ausgeführt werden müssen bzw. sollen. Darunter fällt z.B. das Anlegen automatischer Sicherheitskopien des geöffneten Dokuments in Intervallen von einigen Minuten.
2. Eventuell aufgelaufene Events bearbeiten.
   Sämtliche Aktionen wie Tastatureingaben, Mausbewegungen und Fensteraktionen (Größe ändern, Fokus verlieren, etc.) werden hier abgefragt und bei Bedarf behandelt. Das ist der Kern des Event Loop.
3. Nicht mehr benötigte Objekte löschen.
   Einige Objekte sollten nicht während der Ausführung eines Event-Handlers gelöscht werden, z.B. die Fenster, in denen die Events stattgefunden haben, sonst kann es leicht passieren, dass man den Ast absägt, auf dem man gerade sitzt. Daher melden sich solche Objekte im Falle einer Zerstörung bei einem sog. `LazyDeletor` an, der in dieser dritten Phase mit Rücksicht auf die Reihenfolge alle zu löschenden Objekt sauber entfernt.

Die Beendigung des Programms wird im Übrigen dadurch erreicht, dass sich der Zustand einiger statischen Variablen beim Abarbeiten der Events ändert, wodurch die Schleife kein weiteres Mal durchlaufen wird.

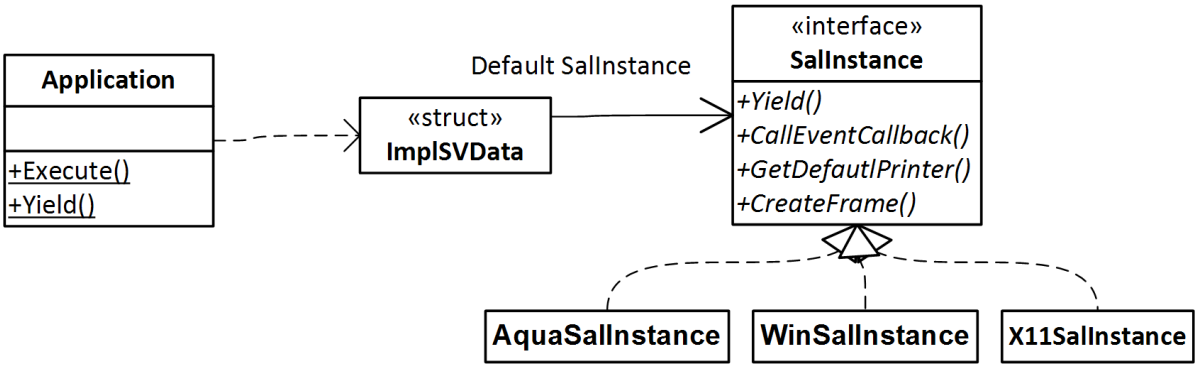

Abbildung A.18: Abstraktion vom Betriebssystem (Auswahl an Schnittstellen-Funktionen)

A.4.2.2 Einlesen der Eingaben

Für eine interaktive Anwendung sind vor allem Ereignisse wie Mausbewegungen, Tastatureingaben oder Fensteraktionen relevant. Diese werden wie oben erwähnt unter Linux vom X Window System bereitgestellt. OpenOffice soll aber auch auf anderen Betriebssystemen als Linux laufen können, die dann auch andere Fenstermanager benutzten. Nachfolgend soll geklärt werden, wie die unterschiedlichen Zugriffsarten in OpenOffice verwaltet werden.

Für die Verwaltung der im Event Loop notwendigen Daten gibt es in OpenOffice eine zentrale statische Struktur vom Typ `ImplSVData`. In ihr werden eine Vielzahl von Metadaten über die Anwendung bereit gehalten, in denen u.a. auch eine Schnittstelle zum Betriebssystem über ein Objekt vom Typ `SalInstance` enthalten ist. Dieses Objekt wurde beim Start des Programms angelegt wurde, sodass die aufrufende Methode nicht zu erkennen braucht, auf welchem System sie tatsächlich läuft. Unter Linux handelt es sich dabei um ein Objekt der Klasse `X11SalInstance`. Für andere Betriebssystem gibt es andere Klassen, die ebenfalls das Interface `SalInstance` implementieren, darunter `WinSalInstance` für Windows und `AquaSalInstance` für Mac OS X. Diese Zusammenhänge sind in Abbildung A.18 dargestellt.

Innerhalb der `Application::Yield()`-Methode wird auf eben dieses `SalInstance`-Objekt zugegriffen und dessen `Yield()`-Methode aufgerufen.

Im konkreten Fall der `X11SalInstance` werden dafür nochmals die Metadaten bemüht, in denen sich auch eine Struktur mit Daten über das Betriebssystem befindet, die `SalData`. Diese hat je nach Betriebssystem eine unterschiedliche Ausprägung, im Falle von Linux handelt es sich dabei um ein Objekt der Klasse `X11SalData`. Innerhalb dieser systemspezifischen Daten wird über ein `SalXLib`-Objekt der Zugang zum X Window System verwaltet. Das `X11SalInstance`-Objekt implementiert seine `Yield()`-Methode einfach indem es den Aufruf an das `SalXLib`-Objekt delegiert.

Die SalXLib::Yield()-Methode

In der `SalXLib`-Klasse werden mehrere sog. `YieldEntry`-Objekte verwaltet, wovon jedes einen File-Deskriptor zum Lesen vom X Server und drei `YieldFunc`-Objekte umfasst:

pending Wird benutzt, um in Erfahrung zu bringen, ob es ausstehende Events gibt, die noch nicht behandelt wurden. Belegt mit der Funktion `DisplayHasEvent()`.

queued Wird benutzt, um ausstehende Events zu lesen und in die Warteschlange einzureihen. Belegt mit der Funktion `DisplayQueue()`.

handle Wird benutzt, um die Events aus der Warteschlange der Reihe nach zu behandeln. Belegt mit der Funktion `DisplayYield()`.

Diese Belegung ist durch den Konstruktor von `SalX11Display` festgelegt, der diese drei Funkionen als neuen `YieldEntry` mit `SalXLib::Insert()` in den Xlib-Wrapper einfügt. Auf dem gleichen Weg könnten weitere Einträge hinzugefügt werden. Die `YieldEntry`-Objekte verfügen darüber hinaus noch über einen Verweis auf das zugehörige `SalX11-Display`-Objekt. Somit kann in den drei Yield-Funktionen auf das zugehörige Display zugegriffen werden.

Wenn der Xlib-Wrapper nun die Events einlesen soll, prüft er erst, ob bereits in einer der Warteschlangen ein Event liegt. Das ermittelt er über die `pending`-Funktion des jeweiligen `YieldEntry`-Objekts. Ist dem so, wird dieses Event behandelt (mit der `handle`-Funktion) und anschließend das Einlesen beendet. Sind alle Warteschlangen leer, wird einen Augenblick lang darauf gewartet, dass ein neues Event eintrifft (einer der File-Deskriptoren wird lesbar). Ist das Timeout verstrichen, und es ist kein neues Event eingetroffen, wird die Funktion beendet. Trifft aber während der Wartezeit ein Event bei einem der YieldEntry-Objekt ein, werden alle Einträge mit der `queued`-Funktion überprüft (wodurch etwaige Events in die Warteschlange eingetragen werden) und dann das erste der Events über die `handle`-Funktion abzuarbeiten.

Der Aufruf des Xlib-Wrapper führt also unterm Strich dazu, dass das erste Event in einer der Warteschlangen behandelt wird, oder (wenn sie alle leer sind) dass alle verfügbaren Events in die Warteschlangen eingereiht werden und jeweils das erste in jeder Warteschlange behandelt wird.

Die Methoden `DisplayHasEvent()`, `DisplayQueue()` und `DisplayYield()` sind im Prinzip nur Wrapper-Funktionen für einige Xlib-Funktionen. Die Funktionsweise wird nun kurz beschrieben (siehe auch Abbildung A.19):

DisplayHasEvent() Diese Methode greift auf die `SalX11Display::IsEvent()`-Methode zu.
Jedes `SalX11Display`-Objekt hat einen Puffer für Events. Wenn sich Anfang und Ende dieser Liste unterschieden, dann wurde ein Event eingefügt und die Antwort kann schnell gegeben werden.
Falls das nicht der Fall ist, wird eine Xlib-Funktion angewendet: Die Funktion ruft `XEventsQueued()` im Modus `QueuedAlready` auf und übergibt das Display. Der

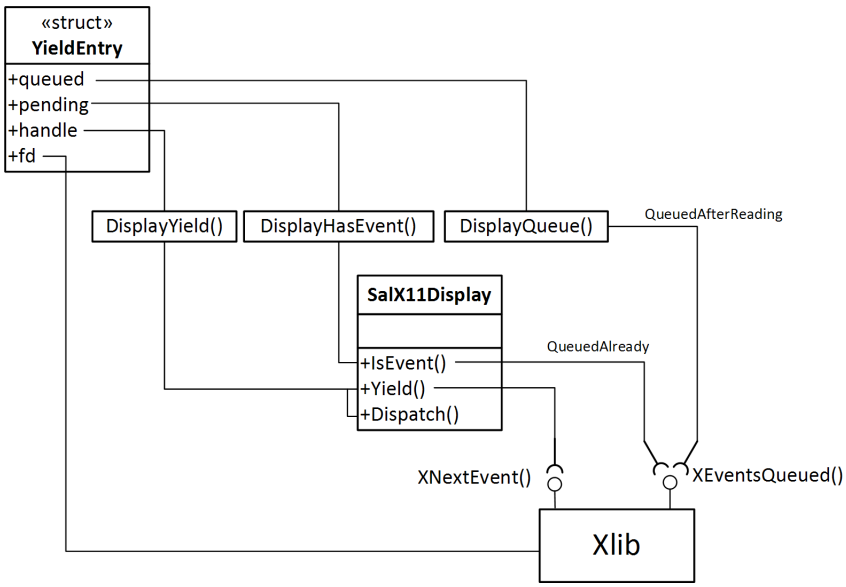

Abbildung A.19: Abbildung der YieldFunc-Einträge auf Xlib-Funktionen

gewählte Modus führt dazu, dass nur die Anzahl der eingereihten Elemente wiedergegeben wird und kein Syscall ausgeführt wird, also auch keine I/O-Operationen. [XLIBb]

DisplayQueue() Diese Methode greift direkt auf die Xlib-Funktion `XEventsQueued()` im Modus `QueuedAfterReading` zu und übergibt das Display als weiteren Parameter. Dieser Modus bewirkt, dass die Anzahl der eingereihten Elemente ausgegeben wird, wenn es Elemente gibt, und es wird versucht neue zu lesen, wenn es keine gibt. [XLIBb]

DisplayYield() Diese Methode greift auf die `SalX11Display::Yield()`-Methode zu. In eben dieser wird wiederum eine Xlib-Funktion benutzt und zwar `XNextEvent()`. `XNextEvent()` kopiert dabei das erste Event in der internen Warteschlange an die bestimmte Position und entfernt es dann aus der Warteschlange. Wenn diese aber leer ist, blockiert die Funktion so lange, bis ein Event in der Queue ankommt. Da aber vorher geprüft wurde, ob es ein Event gibt (mit `XEventsQueued()`), kann davon ausgegangen werden, dass die Funktion nicht blockieren wird und sofort das nächste Event liefert. Danach wird die Methode `SalX11Display::Dispatch()` mit dem eben empfangenen Event aufgerufen.

SalX11Display::Dispatch()

Zu Beginn dieser Funktion wird die im Abschnitt über das X Window System erwähnte Input-Method aufgerufen, um ihr die Chance zu geben, das Event abzufangen. Die einzige Konstellation, in der die Events nicht gefiltert werden ist, wenn keiner der Frames des Displays mit dem Fenster des Events übereinstimmt. „*Frames*“ werden dabei die

konkreten Ausprägungen (konkret in Bezug auf das Betriebssystem) eines Fensters oder eines Teilfensters genannt.

Die eigentliche Aufgabe dieser Funktion besteht darin, den Frame im Display zu finden, zu dem das eingegange Event gehört. Wird dieser Frame gefunden, dann wird auf diesem die `Dispatch()`-Funktion ausgeführt. Für den Fall, dass dieser Frame nicht gefunden werden kann, wird Event an die statische Methode `X11SalObject::Dispatch()` weitergereicht.

X11SalFrame::Dispatch()

In diser Funktion werden die eingehenden Events nach ihrem Typ aufgesplittet. Tastendrücke werden an die Funktion `HandleKeyEvent()` weitergereicht. Neben den KeyEvents werden hier natürlich auch andere Events dispatcht: `HandleMouseEvent()`, `HandleFocusEvent()` und `HandleSizeEvent()` sind einige der Funktionen, die von hier aus aufgerufen werden können.

X11SalFrame::HandleKeyEvent()

In dieser Funktion wird mit Hilfe des X Servers der sog. KeySym-Wert ermittelt. Der KeySym-Wert ist der Wert, der tatsächlich vom Benutzer „gemeint“ war. Die Kombination von `Shift` und `5` hätte also den KeySym %. Dafür werden vom X Server die Kontext-Informationen benutzt, die bei der Herstellung der Verbindung ausgetauscht wurden. So wird das Zeichen in der von OpenOffice unterstützten Kodierung übertragen. Die Umwandlung wird von der Funktion `X11SalDisplay::GetKeySym()` bewerkstelligt, welche wiederrum auf die Xlib-Funktion `XLookupString()` bzw. `XmbLookupString()` und `XKeycodeToKeysym()` zugreifen.

Generell werden in dieser Funktion viele Kodierungen hin- und hergeschoben, worauf ich hier nicht näher eingehen möchte. Mit dem Aufruf von `CallCallback()` wird der Übergang zum Verteilen der Eingaben getätigt.

A.4.2.3 Verteilen der Eingaben

Im voherigen Abschnitt wurde gezeigt, wie der Linux spezifische Code von OpenOffice die Eingaben vom X Window Server erfragt und in ein Format bringt, das unabhängig vom Betriebssystem ist. In diesem Abschnitt soll die Brücke zur konkreten Ausprägung von OpenOffice.org geschlagen werden, an deren Ende Writer, Calc oder Impress ansetzen. Wie schon Eingangs erwähnt ist dieser Programmteil komplett unabhängig vom Betriebsystem.

SalFrame::CallCallback() und ImplWindowFrameProc()

Die Funktion `SalFrame::CallCallback()` ist lediglich eine Hilfsfunktion um den sog. *Processor* aufzurufen. Die Aufgabe des Processors besteht darin, ein auf ein Fenster

bezogenen Event richtig zuzuordnen. Diese Aufgabe kommt der Funktion `ImplWindowFrameProc()` zu.

Der Aufbau von `ImplWindowFrameProc()` entspricht wieder dem eines Dispatchers: je nach Typ des übergebenen Events werden verschiedene Handler aufgerufen. Bei einem Tastendruck, der dem Typ `SAL_KEYINPUT` entspricht, wird das übergebene Event zunächst in ein `SalKeyEvent` gecastet und dann aufgeteilt in `Code`, `CharCode` und `Repeat` als Parameter beim Aufruf der Funktion `ImplHandleKey()` verwendet.

ImplHandleKey()

Beim Betreten dieser Funktion werden die drei Bestandteile des ehemaligen `SalKeyEvent`-Objekts in einem `KeyEvent`-Objekt verabeitet. Dieses neue Objekt wird jetzt an etwaige applikationsweite `KeyListener` weitergereicht. In den untersuchten Fällen gab es keinen solchen Listener.

Für den weiteren Verlauf ist nun wichtig, das Fenster zu identifizieren, in das der KeyInput gehen soll. Das erledigt die Funktion `ImplGetKeyInputWindow()`, die als Parameter das Fenster erhält, das auch diese Funktion beim Aufruf übergeben bekommen hat.

Aus diesem Fenster und dem KeyboardEvent wird nun ein NotifyEvent erstellt, welches dafür benutzt wird, um ein sog. Pre-Notify durchzuführen. Das ist eine Art Vorrecht vor den eigentlichen Empfänger über ein Event informiert zu werden. Die dafür nötigen Aufrufe sind in `ImplCallPreNotify()` implementiert. Darauf soll hier aber nicht näher eingegangen werden, weil bei einem normalen Tastendruck keiner der Vorinformierten auf das Event reagiert. Die einzigen Tastenkombinationen, die ich auf diesem Pfad verfolgen konnte waren `Strg+Tab`, `Alt+Tab` und `F6`.

Da das Event noch nicht „verbraucht“ wurde, wird es nun aufgeteilt: die Tastendrücke gehen an die `KeyInput()`-Funktion des vorher ermittelten Fensters, die Events, die ein Loslassen anzeigen, gehen an die `KeyUp()`-Funktion. Mit dem Aufruf dieser Funktion wird der VCL verlassen und die Eingabe an die „eigentliche“ Anwendung weitergegeben, im betrachteten Fall also an den OpenOffice.org Writer.

Danach folgen noch einige Anweisungen, die nur ausgeführt werden, wenn das gefundene Fenster die Tastatureingabe nicht verarbeitet hat. Die Betrachtung dieser Ausnahmen liefert jedoch keinen Zugewinn für das Verständnis des Systems.

Der Zusammenhang der bisher passierten Klassen und Methoden ist in Abbildung A.20 nochmal in einer Übersicht dargestellt. Dabei wurde auf die Darstellung der Interfaces verzichtet.

A.4.3 OpenOffice.org Writer

Sämtliche Programmteile, die mit dem Writer zusammenhängen, haben den Präfix „SW“. Dieser leitet sich aus der ursprünglichen Codebasis von OpenOffice ab, die aus einer Zeit

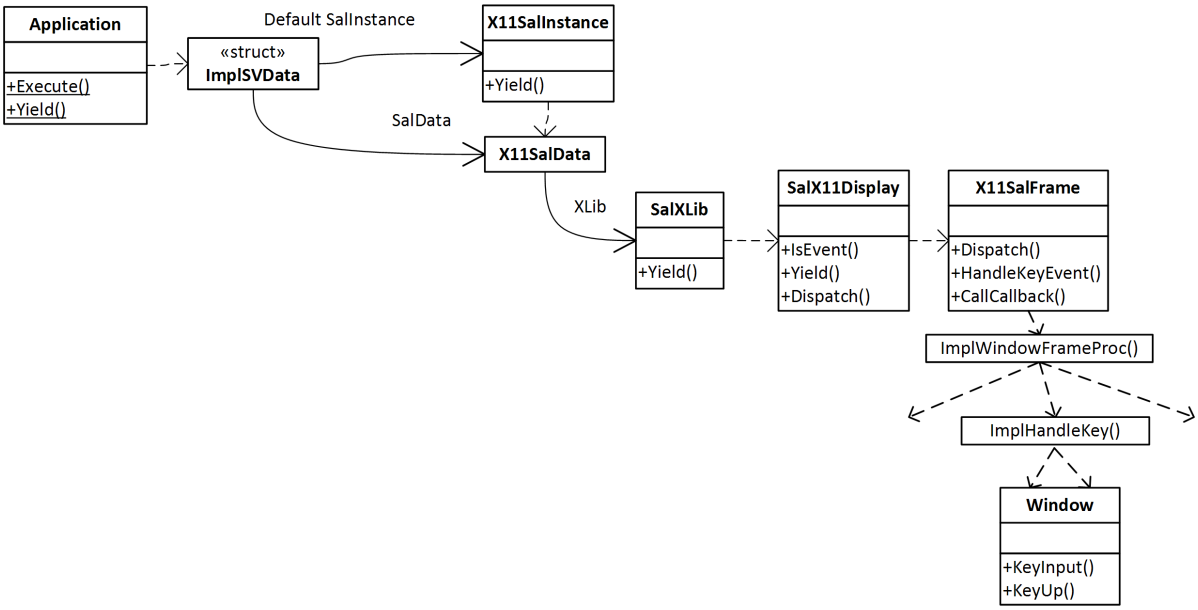

Abbildung A.20: Event-Loop, Einlesen und Verteilen der Eingaben im Überblick

stammt, als OpenOffice noch StarOffice hieß. „SW“ steht dementsprechend für „Star-Writer“.

Im ersten Teil von OpenOffice wurden die Eingaben vom jeweiligen Betriebssystem bzw. seinem grafischen Aufbau empfangen und über einige Schleusen und Ausnahmebehandlungen an den Writer weitergereicht. Dieser Teil beschreibt nun die Vorgänge, die ein Tastendruck auf dem „A“ innerhalb des Writers auslöst.

Im Wesentlichen sind drei große Operationen durchzuführen:

1. Es muss die interne Repräsentation des Dokumentes geändert werden, da sie jetzt um einen Buchstaben reicher ist.
2. Durch diese Änderungen können sich Teile im Dokument verschieben. Etwaige Änderungen sind hier zu berechnen.
3. Diese Änderung müssen dem VCL mitgeteilt werden, damit am Ende das „a“ auch auf dem Bildschirm erscheinen kann.

Es liegt nahe, die Operationen auch in dieser Reihenfolge auszuführen, da eine Operation die andere voraussetzt.

A.4.3.1 Anpassung der internen Repräsentation

Ausgehend vom VCL besteht eine OpenOffice-Anwendung aus Fenstern, auf denen Methoden wie `KeyInput()` aufgerufen werden können. So verhält es sich auch mit dem `SwEditWin`, der Klasse von Fenster die im Falle von OpenOffice Writer auf diese Weise angesprochen wird. Man kann sich Objekte dieser Klasse am besten als Eingabe-Fenster

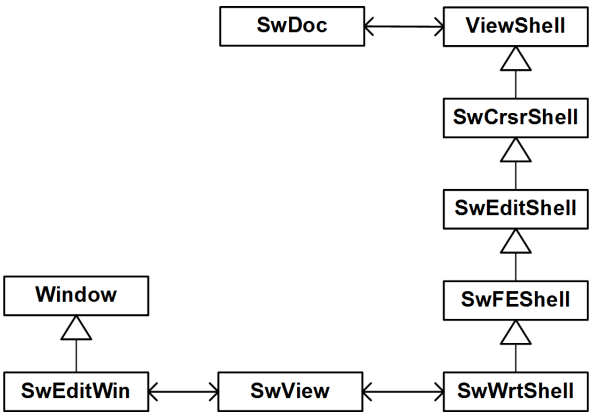

Abbildung A.21: Verbindung von SwEditWin und SwDoc

vorstellen. Eng mit dem Fenster verknüpft die aktuelle Sicht auf das Fenster, die sog. View. In der View sind z.B. die Daten der horizontalen und vertikalen Scrollleisten enthalten, die tatsächliche Dokumentgröße, der gerade sichtbare Bereich, etc. Eine View hat aber keinen Zugriff auf die „Interna“ eines Dokumentes. Dafür wird eine Shell benötigt. Ähnlich wie bei einem Betriebssystem, bei dem die Shell eine Hülle um das eigentliche System bilden, verhält es sich auch mit der Writer-Shell, die alle Zugriff auf das Innenleben des Dokuments steuert. Von Programmteilen außerhalb des Writer-Codes wird das Fenster angesprochen, im Writer-Code wird dafür häufig die Shell benutzt.

SwEditWin::KeyInput() und SwEditWin::FlushInBuffer()

Das Fenster, das in der letzten Funktion des ersten Teils gefunden wurde und auf dem `KeyInput()` und `KeyUp()` aufgerufen wurde, ist im Falle vom Writer ein Fenster der Klasse `SwEditWin`, welche sich direkt von `Window` ableitet.

Ein paar Vorbereitungen, die unter anderem auch den `InBuffer` dieses Fensters verändern können. Wird am Ende der Funktion festgestellt, dass sich Daten in diesem Buffer befinden, wird die Funktion `SwEditWin::FlushInBuffer()` aufgerufen.

`FlushInBuffer()` holt sich als erstes die Shell (`SwWrtShell`) über den View (`SwView`), um auf das Dokument zugreifen zu können. Auf der `SwWrtShell` ruft sie dann die Funktion `Insert()` mit dem `InBuffer` auf.

SwWrtShell::Insert()

Die `SwWrtShell` prüft vor dem Einfügen, ob ein Bereich markiert ist, oder ob der Cursor einfach nur blinkt. Ich gehe im Folgenden davon aus, dass der Cursor einfach nur im leeren Dokument blinkt. Dann wird unterschieden, ob im „Einfügen“- oder im „Überschreiben“-Modus getippt wird. Auch hier gehe ich vom Normalfall aus: „Einfügen“. In diesem Fall wird die `Insert()`-Methode der `SwEditShell` aufgerufen, einer Basisklasse der `SwWrtShell`.

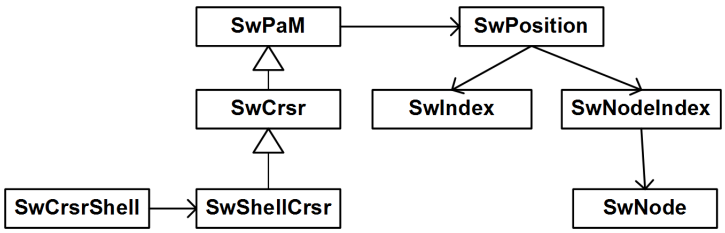

Abbildung A.22: Verberbung und Assoziation eines Cursors

SwEditShell::Insert()

Alle Operationen, die `SwEditShell::Insert()` durchführt, um den vorgegebenen Text in das Dokument zu bringen, werden von einem Funktionspaar umrandet: `StartAllAction()` und `EndAllAction()`.

StartAllAction iteriert ausgehend von sich selbst über alle nachfolgenden Shells und ruft auf jeder `StartAction()` (je nach Typ entweder `SwEditShell::StartAction()` oder `ViewShell::StartAction()`) auf.

EndAllAction iteriert analog mit `EndAction()`.

Innerhalb dieser Klammerung wird der aktuelle Cursor geholt und dann für jeden existierenden Cursor `Insert()` auf dem Dokument (Klasse `SwDoc`) aufgerufen.

SwDoc::Insert()

Erst wird die Position aus dem übergebenen Cursor ermittelt (`SwPosition`), und dann aus dieser Position der Textknoten (gecasteter `SwNode`). Dahinter steckt keine große Magie, weil die Position im `SwShellCrsr` schon drin steht, wie in Abbildung A.22 zu sehen ist.

Auf diesem Knoten wird dann eine `Insert()`-Methode aufgerufen, die den einzufügenden Inhalt, die Position innerhalb des Knotens und den Schreibmodus übergeben bekommt.

Am Ende der `SwDoc::Insert()`-Methode wird `SetModified()` aufgerufen, um dem Dokument mitzuteilen, dass es gerade geändert wurde. Dabei wird unter anderem das Flag `mbModified` im `SwDoc` gesetzt, und wenn es Ole2.0-Objekte gibt, dann werden die auch benachrichtigt. Dieser Mechnismus würde aber hier zu weit führen.

A.4.3.2 Berechnung der Änderungen

Der Mammutteil der Berechnung der Änderungen findet seinen Anfang in der unscheinbaren Funktion `SwEditShell::EndAllAction()`, deren Pendant als öffnende Klammer die Bearbeitung des Dokumentes eingeläutet hat.

SwEditShell::EndAllAction()

Wie oben schon erwähnt wird hier lediglich über den Ring von Shells iteriert, und auf jeder einzelnen EndAction() aufgerufen. Dabei wird wieder zwischen `SwEditShell::EndAction()` und `ViewShell::EndAction()` unterschieden.

Die `SwEditShell::EndAction()` existiert nicht namentlich sondern wird durch die in der Vererbungshierarchie höher stehende `SwCrsrShell::EndAction()` ausgeführt. Deren einzige relevante Anweisung ist der Aufruf der `ViewShell::EndAction()`.

ViewShell::ImplEndAction()

Die einzige Anweisung, die ich hier verfolgt habe, war `pLayout->Paint()`.

A.4.3.3 Weitergabe der Änderungen an den VCL

Das Grundschema beim Zeichnen scheint dem „Divide and Conquer"-Prinzip entlehnt zu sein: zuerst wird der Zeichenauftrag auf Seiten aufgeteilt, dann auf Zeilen.

SwRootFrm::Paint()

Diese Funktion hat die Aufgabe, jede sichtbare Seite des Dokuments, die vom übergebenen Rechteckt berührt wird, einmal zu zeichnen.

Zu Beginn der Funktion wird berechnet, wie groß ein Pixel auf dem Ausgabegerät dargestellt wird. Dafür gibt es die Hilfsfunktion `SwCalcPixStatics()` welcher ein `OutputDevice` übergeben werden muss. Dafür wird das Ausgabegerät der aktuellen Shell benutzt. Berechnet wird hierbei die Größe eines Pixels in Twips ($\frac{1}{1440}$ Zoll = 17,6 μm), so lange es nicht notwendig wird, mit Pixeln zu rechnen wird intern mit Twips gerechnet.

Nun wird die Schnittmenge aus dem übergebenen Rechteck und dem sichtbaren Bereich der `ViewShell` gebildet[14].

Für die spätere Verarbeitung wird an dieser Stelle geprüft, ob es sog. „Extra-Daten" gibt. Darunter fallen zur Zeit Zeilennummern und Änderungsmarkierungen im Dokument. Weiterhin wird hier auch ermittelt, ob es sonstige Besonderheiten in der Darstellung gibt: Buch-Layout, umgekehrte Text-Richtung, soll ein Gitter gezeichnet werden, etc.

Ausgehend von der ersten sichtbaren Seite wird nun iteriert: zuerst werden die Seitenumrandung und die Hintergründe gemalt, dann die sog. „Hölle" (das sind alle Rahmen und Zeichenobjekte, die unter dem Text-Inhalt liegen), dann der Dokument-Inhalt und dann den sog. „Himmel" (Objekte über der Text-Ebene).

[14] Das ist bei allen experimentellen Situationen unnötig gewesen: selbst wenn die aktuelle Zeile nicht komplett im sichtbaren Bereich war, wurde nur das sichtbare Rechteck übergeben, sodass der Schnitt nur das Rechteck selbst war.

Umrandung und Hintergründe zeichnen Zuerst wird ein Rechteck berechnet:

- Dafür wird erst das „Border“-Rechteck ermittelt: `AlignRect()` und danach nach allen Seiten um einen Pixel erweitert. Dieses Rechteck umfasst die gesamte Seite, auch wenn sie nicht komplett sichtbar ist, daher kann dieses Rechteck aus dem Bildschirm herausragen.
- Danach das „RightShadow“-Rechteck: ein schmales Rechteck rechts neben dem eigentlichen Frame. Diese beiden werden vereint.
- Danach das „BottomShadow“-Rechteck: ein flaches Rechteck unter dem eigentlichen Frame. Auch dieses wird mit dem bisherigen vereint.
- Wenn das so ermittelte Rechteck über dem liegt, das laut Parameter gezeichnet werden soll, so wird die Schnittmenge der beiden gebildet, um nicht zu viel zeichnen zu müssen. Weiterhin wird ein Speicher für zu zeichnende Linien angelegt, darin können doppelte Linienzüge erkannt und so Zeichenarbeit gespart werden.
- Das entstandene Rechteck wird nun noch auf volle Pixelgröße vergrößert, sodass alle Pixel, die es überlappt, mit aufgenommen werden.

Anschließend wird die Hilfsfunktion `PaintBaBo()`) mit dem ermittelten Rechteck und der aktuellen Seite aufgerufen. Durch das Rechteck wird sichergestellt, dass keine Bereiche zum Zeichnen in Auftrag gegeben werden, die am Ende gar nicht sichtbar sind.

Die Hölle zeichnen (Rahmen und Zeichenobjekte die unter dem Dokument liegen) Für das Zeichnen wird die Shell bemüßigt, die eine Methode `PaintLayer()` anbietet. Diese wird zum Zeichnen von Himmel und Hölle benutzt.

Den Dokumentinhalt (Text) zeichnen . Zum Zeichnen des Dokumentinhaltes inklusive des getippten Buchstabens wird nun auf der aktuelle Seite die `Paint()`-Methode aufgerufen. Auch ihr wird das vorher berechnete Rechteck übergeben, um unnötige Berechnungen zu vermeiden.

Den Himmel zeichnen (Layer, der über dem Dokument liegt) Wie oben erwähnt, wird der Himmel auch durch die Shell gezeichnet.

SwLayoutFrm::Paint() und SwTxtFrm::Paint()

Die Abläufe in der Methode `SwLayoutFrm::Paint()` sind mir nicht ganz klar geworden. Bei einigen Testdurchläufen wurde diese Methode mehrmals ineinander geschachtelt aufgerufen. Ich vermute, dass in etwa folgendes passiert:

Die Frames sind ineinander geschachtelt und die Aufgabe dieser Methode ist es, Ebene um Ebene vorzudringen, bis letztendlich der vorgefundene Inhalt kein SwLayoutFrm mehr ist, sodass eine andere `Paint()`-Methode aufgerufen wird.

Tatsächlich wird die `SwTxtFrm::Paint()`-Methode aufgerufen. In dieser werden nach einigen Vorbereitungen nacheinander jede sichtbare Zeile gezeichnet. Dafür wird ein `SwTxtPainter`-Objekt erzeugt, auf dem in einer Schleife die Methode `DrawTextLine()` aufgerufen wird.

SwTxtPainter::DrawTextLine()

Das Schema wird auch in dieser Funktion fortgesetzt und die Textzeilen werden in `SwLinePortion`-Objekte zerhakt, die einzeln mit `Paint()` zum Zeichnen angeregt werden. Als Parameter werden hierbei jeweils Objekte der Klasse `SwTxtPaintInfo` übergeben.

SwTxtPortion::Paint()

Auf den übergebenen Infos wird `DrawMarkedText()` aufgerufen.

SwTxtPaintInfo::DrawMarkedText()

Ruft nur eine Unterfunktion auf: `SwTxtPaintInfo->_DrawText()`

SwTxtPaintInfo::_DrawText()

In dieser Methode wird ganz ausführlich ein `SwDrawTextInfo`-Objekt zusammengesetzt, in das z.B. Informationen über das Kerning (veränderter Zeichenabstand bei bestimmten Buchstabenkombinationen) gespeichert werden. Letztendlich wird die Methode `_DrawText()` auf einem `SwFont`-Objekt aufgerufen. Dieses ist ein Attribut der Klasse `SwTxtPaintInfo`.

SwFnt::_DrawText()

Jedes SwFont-Objekt besteht aus mehreren „Sub-Schriftarten“, zur Zeit sind es Latin, CJK (Chinese, Japanese, Korean) und CTL (complete text layout, Hebrew, Arabic). Die Aufgabe dieser Funktion besteht darin, die richtige auszuwählen und diese mit dem Zeichnen des Textes zu beauftragen.

SwSubFnt::_DrawText()

Diese Funktion benutzt ein Attribute names `pLastFont`, in dem die zuletzt verwaltete Schriftart liegt. Beim Wechsel der Schriftart werden die benutzten Schriftarten in einem Cache gespeichert, sodas sie schneller zugreifbar sind. Hier wird auf den akuellsten Eintrag zugegriffen und die Methode `DrawText()` aufgerufen.

SwFntObj::DrawText()

Diese Methode ist sehr umfangreich und macht im Quelltext über 1000 Zeilen aus. Die letzte Anweisung, die noch in den Zuständigkeitsbereich des Writers fällt ist der Aufruf an das Ausgabegerät, das in dem übergebenen `SwDrawTextInfo`-Objekt referenziert ist. Auf dem `OutputDevice` wird die Methode `DrawTextArray()` aufgerufen. Als Parameter wird die Position, der Text-Inhalt sowie Kerning-Informationen übergeben.

A.4.4 Visual Component Layer (Ausgabe)

Die Schnittstelle zum Writer oder auch zu anderen Anwendungen ist nicht spezifisch für ein Betriebssystem, sondern bezieht sich allgemein auf ein Ausgabegerät. Letztenendes muss aber auch die Verbindung zum X Window System hergestellt werden, was im zweiten Teil passiert.

A.4.4.1 Generische Behandlung

OutputDevice::DrawTextArray()

Zu Beginn wird mit den übergebenen Parametern ein Objekt der Klasse SalLayout erzeugt. Dieses wird als Parameter der Methode ImplDrawText() übergeben.

OutputDevice::ImplDrawText()

In dieser Methode wird ImplDrawTextDirect() aufgerufen.

OutputDevice::ImplDrawTextDirect()

Auf dem übergebenen `SalLayout`-Objekt wird die Methode `DrawText()` aufgerufen und als Parameter ein `SalGraphics`-Objekt übergeben.

ServerFontLayout::DrawText()

`ServerFontLayout` ist eine Implementierung von `SalLayout`. Diese Methode ruft `DrawServerFontLayout()` auf dem übergebenen `SalGraphics`-Objekt. Damit ist die allgemeine Behandlung beendet, weil das `SalGraphics`-Objekt eine auf das Betriebssystem ausgerichte Ausprägung hat.

A.4.4.2 Ausgabe an das X Window System

OpenOffice benutzt zumindest für Textausgaben die Bibliothek Cairo, die unter anderem das X Window System bedienen kann.

X11SalGraphics::DrawServerFontLayout()

Diese Methode beschafft sich ein CairoWrapper-OBjekt und ruft dann DrawCairoAA-FontString() auf.

X11SalGraphics::DrawCairoAAFontString()

Dieser Funktion wird ein `ServerFontLayout` übergeben. Aus diesem Layout wird mit der Methode `GetNextGlyphs()` das A gezaubert. Gespeichert wurde es vorher in der Liste der `GlyphItems`. Der Speicher, also der Bereich im `GenericSalLayout` ist dabei unabhängig vom Betriebssystem. Die Glypth-Items enthalten genügend Informationen, um verschiedene Fenstersysteme mit den jeweils benötigten Informationen zu versorgen. Für Cario wird lediglich der Index und die Position benötigt.

Abschließend wird die Hilfsfunktion `show_glyphs()` aufgerufen.

CairoWrapper::show_glyphs()

Die Parameter sind ein `cario_t`, der erste `cario_glyph` eines Vektors, und die Anzahl der Glyphen. Diese Methode ist die letzte untersuchte. Sie ruft die Cairo-Funktion `cairo_show_glyphs()` auf.

Es folgen nun meine eigenen Beobachtungen bezüglich der Kommunikation mit dem X-Server.

A.4.5 Kommunikation mit X.Org

Für die Kommunikation mit dem X.Org-Server (jeweils bei Ein- und Ausgabe) verwendet OpenOffice die Programmbibliothek Xlib (Abbildung A.23). Bei der Eingabe wird so das Event abgerufen, welches vom X-Server in das entsprechende Netzwerk-Socket geschrieben wurde (A.3.2).

Bei der Ausgabe wird Xlib indirekt verwendet. Zum Darstellen der Buchstaben verwendet OpenOffice.org die unabhängige Bibliothek Cairo. Cairo unterstützt verschiedene Grafiksysteme (z. B. auch das von Microsoft Windows) und verwendet Xlib, um die Funktionen zur Darstellung des „a“ als Anfragen an den X.Org-Server weiterzugeben.

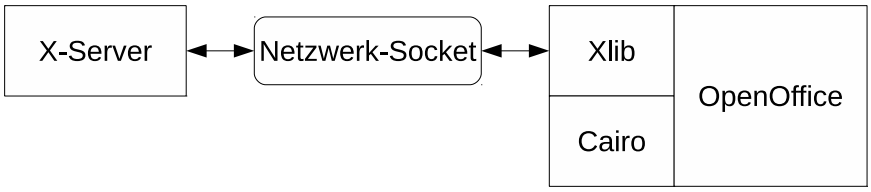

Abbildung A.23: Xlib

A.5 X.Org-Server - Ausgabe

Um den Vorgang der graphischen Ausgabe zu verstehen, sollten einige grundlegende Informationen über Grafikkarten bekannt sein. Im Arbeitsspeicher einer Grafikkarte gibt es einen als Scan-out-Buffer bezeichneten Bereich, in dem die aktuelle Farbe jedes Pixels, also das gesamte sichtbare Bild gespeichert ist. Der Scan-out-Buffer wird in regelmäßigen Abständen (der Bildwiederholungsrate) an den angeschlossenen Bildschirm geschickt.

Allgemein werden solche Repräsentationen des sichtbaren Bildes im Speicher als Framebuffer bezeichnet. Die Größe des Framebuffers ist von Auflösung und Farbtiefe abhängig, z. B. ergibt sich bei einer Auflösung von 1920 x 1080 Pixeln und einer Farbtiefe von 32 Bit (4 Byte) eine Größe von $1920 \cdot 1080 \cdot 4$ Byte = 829440 Byte.

Wie die Pixel im Speicher angeordnet sind, ist unterschiedlich. Im einfachsten Fall handelt es sich um eine lineare Anordung. Hierbei liegen einfach alle Pixel einer Reihe nebeneinander im Speicher, im Beispiel würde also jeweils nach $1920 \cdot 4$ Byte = 7680 Byte eine neue Reihe beginnen. Eine andere Möglichkeit besteht darin, das Bild in Segmente statt Reihen aufzuteilen. Bei einer Segmentgröße von 4×4 Pixel würden also die ersten 64 Byte die Informationen der ersten vier Pixel der ersten vier Reihen enthalten. Ein Vorteil dieser Anordnung ist, dass auch vertikal benachbarte Pixel im Speicher nebeneinander liegen.

Damit der X.Org-Server Ausgaben auf den Bildschirm bringen kann, muss dieser also im Endeffekt in den Scan-out-Buffer der Grafikkarte schreiben. Hierzu muss der X.Org-Server über die Position und Beschaffenheit dieses Framebuffers Bescheid wissen. Beim Starten des X.Org-Servers wird dazu ein entsprechender Grafikkartentreiber geladen, in unserem Fall ist dies VESA, der Standard-Treiber für Linux-Systeme. Dieser Treiber ist mit den meisten Grafikkarten kompatibel, kann deswegen aber nicht auf herstellerspezifische Funktionen zum schnelleren Berechnen von grafischen Veränderungen zurückgreifen. Dies führt dazu, dass sämtliche Veränderungen am Framebuffer zunächst im Hauptspeicher durchgeführt werden.

Bei X wird dies durch den sog. Shadow Layer[XORGc] realisiert, eine Kopie des Framebuffers auf der Grafikkarte. Nachdem diese Kopie verändert wurde, werden die veränderten Bereiche in den Framebuffer der Grafikkarte geschrieben.

Um das „a“ in den aktuellen Bildschirminhalt einzufügen, werden drei temporäre Bilder verwendet[Pac01]:

Mask
: Die einzufügende Form (hier der Umriss des „a“) als Graustufenbild, wobei die Farbwerte der einzelnen Pixel angeben, wie transparent das Quellbild ist (ein absolut schwarzer Pixel in der Maske würde dazu führen, dass nach dem Compositing an dieser Stelle der entsprechende Pixel des Quellbilds sichtbar wäre);

Source
: Der Inhalt der neuen Form (hier die Textfarbe Schwarz);

Destination

Der aktuelle Bildschirminhalt im betroffenen Bereich (hier ein Ausschnitt des weißen Dokuments).

Vereinfacht beschrieben wird also ein Stück in Form der Maske aus dem Quellbild herausgeschnitten und über das Zielbild gelegt (Abbildung A.24).

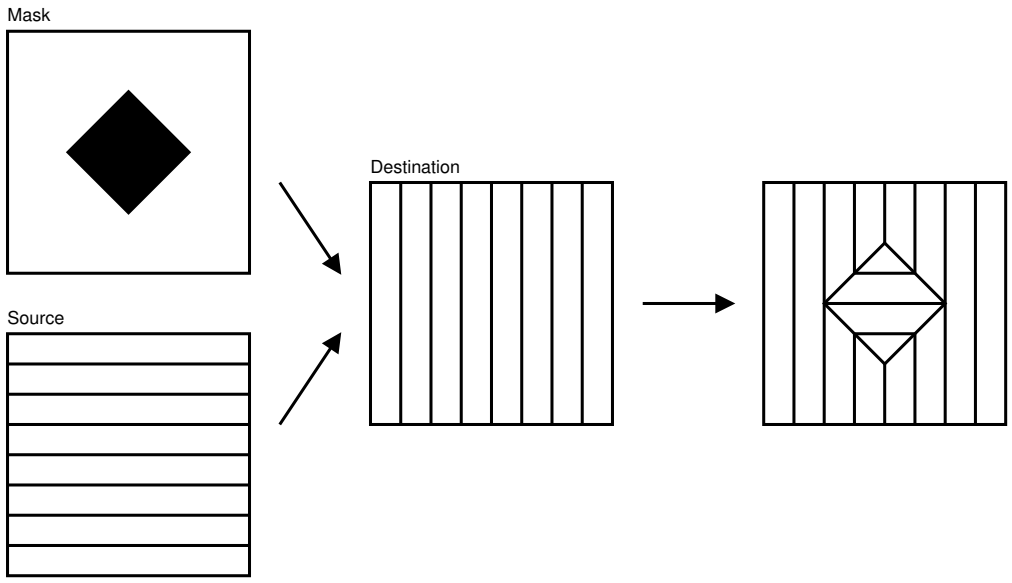

Abbildung A.24: Compositing

Dieses Zusammenfügen aus mehreren Elementen wird Compositing genannt, wobei im echten Anwendungsfall die Maske nicht nur zwischen sichtbaren und unsichtbaren Bereichen unterscheidet, sondern auch durchsichtige Bereiche markiert. Gerade beim Darstellen von Buchstaben werden solche transparenten Ränder eingesetzt, um die Kanten der Buchstaben weicher erscheinen zu lassen.

Um einen Buchstaben auf den Bildschirm zu bringen, gehen von OpenOffice drei Anfragen an den X.Org-Server:

1. Anlegen einer neuen Gruppe von Symbolen.

   In anderen Fällen können so auch mehrere Symbole zu einer Gruppe hinzugefügt und dann gleichzeitig dargestellt werden.

2. Hinzufügen des Buchstabens zu dieser Gruppe.

   Hier wird das „a“ an den X.Org-Server übertragen. Der X-Server selbst verwaltet dabei keine Schriftarten, sondern ist darauf angewiesen, dass er den Ort der darzustellenden Glyphe (einzelne Symbole einer Schriftart) vom Client mitgeteilt bekommt.

3. Darstellen der Symbole.

Die Glyphen werden durch Compositing in den Bildschirminhalt eingefügt, also in den Shadow Framebuffer geschrieben. Somit existiert das „a“ im X-Server nicht mehr als separates Objekt, sondern nur noch als Teil einer Grafik.

Bevor der X-Server wieder in den Ruhezustand übergeht, wird noch der tatsächliche Framebuffer an die veränderte Kopie angepasst.

Wie im Fall der Eingabe-Events landen auch diese Anfragen in der internen Event-Queue des X.Org-Servers und werden so ebenfalls beim Durchlaufen der `Dispatch`-Schleife zur weiteren Verarbeitung aufgegriffen.

A.5.1 Von der Anfrage zum Bild

Wir beginnen die genaue Untersuchung bei der Bearbeitung der Anfrage zum Darstellen. Zuvor hat der X-Server also bereits ein Glyph-Set angelegt und dieser das „a“ in Form einer Glyph-Referenz hinzugefügt.

Um möglichst hohe Kompatibilität beim Darstellen von Schriften zu gewährleisten hat der X-Server keine eigenen Informationen über die darzustellenden Schriften. Alles was zum Darstellen der Glyphen benötigt wird, muss also in der Glyph-Referenz enthalten sein. Dazu gehören neben des Speicherorts der Schriftart im Dateisystem und der Nummer der daraus darzustellenden Glyphe auch Informationen zur Positionierung der Glyphe in der Zeile. So liegt z. B. ein „a“ meist direkt auf der Zeile, während ein „Q“ weiter nach unten hinausragt (Abbildung A.15).

aQ

Abbildung A.25: Beispiel Glyphenposition

Wie schon bei der Behandlung des Eingabe-Events wird das Anfrage-Event innerhalb der `Dispatch`-Schleife aufgegriffen, die folgenden Funktionsaufrufe erfolgen geschachtelt.

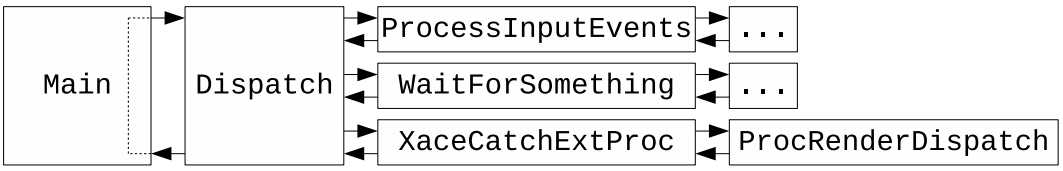

Abbildung A.26: Hauptschleife X-Server

XaceCatchExtProc

Zunächst wird die Anfrage allgemein behandelt: Es wird überprüft, ob der Anfragentyp bearbeitet werden kann, also ob die entsprechende Komponente (hier die Render-Erweiterung[Pac02]) im laufenden X-Server existiert. Die Prüfung fällt positiv aus und die Anfrage wird weitergeleitet.

ProcRenderDispatch

Hier wird die Anfrage auf Gültigkeit geprüft. Danach wird die spezifische Funktion zum Verarbeiten von Glyph-Render-Anfragen aufgerufen.

ProcRenderCompositeGlyphs

In dieser Funktion werden die Variablen für Source und Destination vorbereitet und der entsprechende Speicher reserviert. Außerdem wird geprüft, ob das darzustellende Glyph-Set gültig ist.

CompositeGlyphs

Nun wird überprüft, welche spezifische Methode für das Darstellen der Glyphs verwendet werden soll. Je nach Grafikkarte und Treiber kann diese unterschiedlich sein und wird beim Starten des X-Servers durch den Treiber festgelegt.

damageGlyphs

Da ein Glyph-Set mehrere Symbole enthalten kann, die gleichzeitig dargestellt werden sollen, werden hier die Koordinaten berechnet, um die einzelnen Symbole aufzureihen. Im Fall eines einzelnen Buchstabens ist dies nicht notwendig.

miGlyphs

In dieser Funktion findet die eigentliche Arbeit der Bilderstellung statt.

Als Zielbild dient ein rechteckiger Ausschnitt aus dem aktuellen Bildschirminhalt an der Stelle, wo das „a“ erscheinen soll. Die Maske, also das Bild des Buchstabens, wird geladen und auf die richtige Größe gebracht. Für das Quellbild wird einfach ein schwarzes Rechteck erstellt (Abbildung A.27).

Da nun die einzelnen Komponenten bereitliegen, kann das Zusammenfügen beginnen.

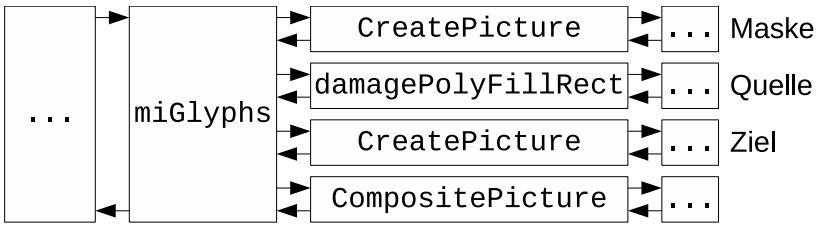

Abbildung A.27: Zusammensetzen der Bilder

CompositePicture

Zunächst wird allerdings noch überprüft, ob eine einfachere Compositing-Funktion verwendet werden kann; dies ist z. B. der Fall, wenn die Maske nur transparente und undurchlässige, aber keine teiltransparenten Bereiche markiert. Da bei der Darstellung von Buchstaben teiltransparente Bereiche verwendet werden, muss die normale Funktion verwendet werden.

damageComposite

Wie schon bei der Funktion zum Darstellen von Glyphen gibt es unterschiedliche Funktionen für das Compositing. Die vom Treiber beim Programmstart festgelegte Funktion wird nun aufgerufen.

fbComposite

Die drei Einzelbilder werden zusammengefügt, wobei die externe Grafikbibliothek pixman die Compositing-Funktion zur Verfügung stellt. Das Ergebnis ist am Ende im Zielbild, also in dem Ausschnitt des Bildschirminhalts, zu finden.

Wie am Anfang dieses Abschnitts beschrieben, handelt es sich dabei allerdings noch um eine lokale Kopie des wirklichen Bildschirminhalts, der im Arbeitsspeicher der Grafikkarte liegt.

A.5.2 Zur Grafikkarte

Bei der Untersuchung der Eingabebehandlung im X-Server kamen wir mit Wakeup-Handlern in Kontakt (A.3.1), die beim Verlassen des Ruhezustands aufgerufen werden. Analog dazu gibt es Block-Handler, die beim Übergang in den Ruhezustand aufgerufen werden.

Verwendet man den Shadow Layer, so gibt es einen entsprechenden Block-Handler, der beim Übergang in den Ruhezustand die Änderungen in den Scan-out-Buffer der Grafikkarte überträgt.

Wie der allgemeine Wakeup-Handler wird auch der Block-Handler von `WaitForSomething` aus aufgerufen (Abbildung A.28).

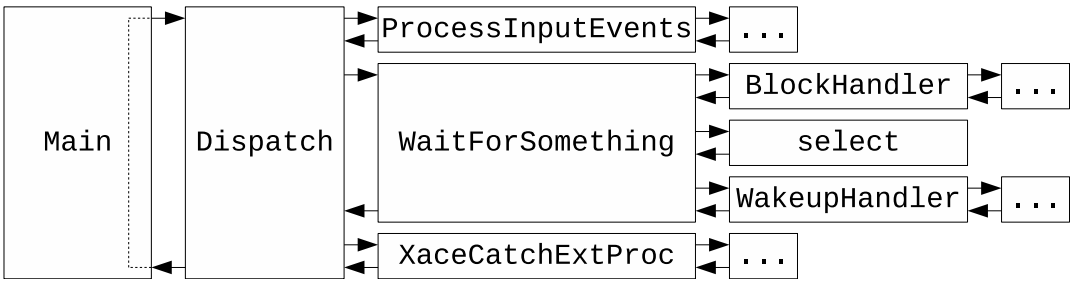

Abbildung A.28: Hauptschleife X-Server

BlockHandler

Der allgemeine Block-Handler ruft einfach nacheinander alle angemeldeten Block-Handler auf.

shadowRedisplay

Falls Unterschiede zwischen dem lokalen und dem Grafikkarten-Framebuffer vorliegen, wird der betroffene Bereich festgestellt und aktualisiert.

shadowUpdatePacked

Der zu aktualisierende Bereich wird Pixel für Pixel in den Framebuffer der Grafikkarte übertragen. Von dort wird das „a“ dann an den Bildschirm geschickt.

Anhang B

Präsentationsentwurf

Betrachtet man den gesamten Software-Ablauf, so lassen sich die Darstellungsformen des „a“ folgendermaßen aufteilen:

- Kernel
  - Interrupt
  - Kernel-Event
  - Evdev-Event
- X-Server
  - Keyboard-Event
  - X-Event
- OpenOffice
  - OO-Event
  - Teil des Dokuments
  - Teil einer Zeile
  - Glyphe
  - X-Anfrage
- X-Server
  - Teil eines Glyphen-Set
  - Bild
  - Teil eines Bilds

Dabei wurde eine Form jeweils der Komponente zugeordnet, in der sie als erstes auftritt. Eine weitere Unterteilung wäre möglich, wenn man die Unterklassen der verschiedenen Objekt-Formen hinzunehmen würde (z. B. ist ein X-Event eine abstrakte Oberklasse, hier von einem X-Tastatur-Eingabe-Event). Da in solchen Fällen aber keine expliziten Umwandlungen auftreten, habe ich davon abgesehen.

Die Abbildung B.1 bietet eine Übersicht der verschiedenen Formen zusammen mit den verwendeten Kommunikationsschnittstellen. Diese sind als abgerundete Rechtecke dargestellt und die Parallelogramme repräsentieren Ein- und Ausgabepunkte von Hardware-Komponenten.

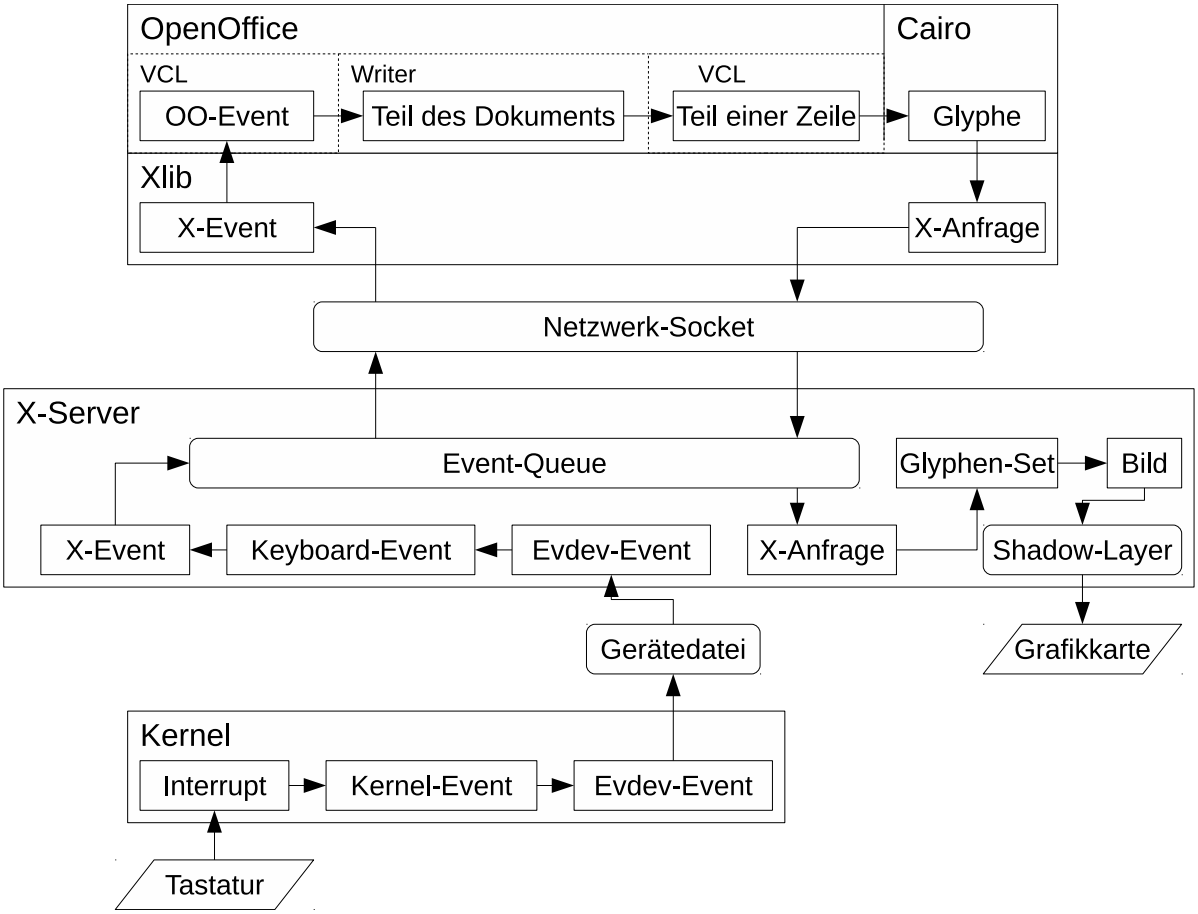

Abbildung B.1: Übersicht Formen des „a“

Von der Tastatur aus beginnend könnten nun die einzelnen Punkte nacheinander erläutert werden:

- Was ist ein Interrupt?
- Warum wird aus dem Interrupt ein Kernel-Event?
- ...

Anhang C

Inhalt der CD

Die beiligende CD enthält die der Arbeit zu Grunde liegenden Daten:

Diagramme enthält die vollständigen Ablaufdiagramme,

Dokument enthält die Quelldateien dieses Dokuments,

Event enthält den Quellcode des C-Programms zum Überwachen einer Datei auf Events,

Logs enthält die aufgenommene Kommunikation zwischen X und OpenOffice,

Traces enthält die zu den Ablaufdiagrammen gehörenden Stacktraces.

Die gleichen Dateien sind ebenfalls unter `http://page.mi.fu-berlin.de/nbdbag/dasa2.zip` abrufbar.

Literaturverzeichnis

[BC06] Bovet & Cesati, Understanding the Linux Kernel. O'Reilly Media 2006

[CC] ccache - a fast C/C++ compiler cache
http://ccache.samba.org/

[Cha04] Adam Chapweske, The PS/2 Keyboard Interface. Computer-Engineering.org 2004
http://www.computer-engineering.org/ps2keyboard

[DEB] Debian GNU/Linux 5.0 („Lenny“)
http://www.debian.org/releases/lenny/

[Güb04] Rolf Gübeli, Technische Informatik 2. vdf Hochschulverlag AG 2004

[INTEL88] 8259A Programmable Interrupt Controller (8259A/8259A-2). Intel Corporation 1988
http://bochs.sourceforge.net/techspec/intel-8259a-pic.pdf.gz

[INTEL99] Intel Architecture Software Developer's Guide Manual, Volume 2: Instruction Set Reference. Intel Corporation 1999
http://download.intel.com/design/PentiumII/manuals/24319102.PDF

[INTEL09] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. Intel Corporation 2009
http://www.intel.com/Assets/PDF/manual/253665.pdf

[Kar09] Christos Karayiannis (2009), X Window System Internals
http://xwindow.angelfire.com/

[KERNEL] Linux Kernel 2.6.28.9
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.28.9.tar.gz

[KGDB] KGDB Wiki
https://kgdb.wiki.kernel.org/

[OOOa] OpenOffice.org 3.1 Core (Milestone OOO310_m11)
ftp://ftp.fu-berlin.de/unix/X11/apps/openoffice/stable/3.1.0/OOo3.1.0srccore.tar.bz2

[OOOb] OpenOffice.org 3.1 System (Milestone OOO310_m11)
ftp://ftp.fu-berlin.de/unix/X11/apps/openoffice/stable/3.1.0/OOo3.1.0srcsystem.tar.bz2

[Pac01] Keith Packard (2001), Design and Implementation of the X Rendering Extension, Usenix Technical Conference 2001
http://keithp.com/~keithp/talks/usenix2001/xrender.pdf

[Pac02] Keith Packard (2002), The Xrender Library
http://www.x.org/releases/X11R7.5/doc/libXrender/libXrender.txt

[QEMU] QEMU
http://wiki.qemu.org/Main_Page

[Ray03] Eric S. Raymond (2003), The Art of UNIX Programming. Addison-Wesley.

[Rus05] Rusty Russell (2005), Unreliable Guide To Hacking The Linux Kernel
https://www.kernel.org/doc/htmldocs/kernel-hacking.html

[XLIBa] Xlib Programming Manual. O'Reilly & Associates, Inc. Ch. 2 „X Concepts“
http://www.sbin.org/doc/Xlib/chapt02.html

[XLIBb] Xlib Programming Manual. O'Reilly & Associates, Inc. Function Index „XEventsQueued()“
http://www.sbin.org/doc/Xlib/xlib/event-handling/XEventsQueued.html

[XMAN] X(7) Manual Page
http://www.x.org/archive/X11R7.0/doc/html/X.7.html

[XORGa] X.Org Server 1.4.2
http://cgit.freedesktop.org/xorg/xserver/snapshot/xserver-xorg-server-1.4.2.tar.gz

[XORGb] X.Org Wiki - Development
http://www.x.org/wiki/Development

[XORGc] Documentation for the X Window System Version 11 Release 6.8.1 (X11R6.8.1) (2004)
http://www.x.org/archive/X11R6.8.1/doc/

[ZIE09] Franz Zieris (2009), Das „a“, Bachelorarbeit an der Freien Universität Berlin
http://www.inf.fu-berlin.de/inst/ag-se/theses/Zieris09-das-a.pdf