Bachelorarbeit

Das „a“

im Studiengang Bachelor Informatik
des Fachbereiches Informatik
der Freien Universität Berlin

Sommersemester 2009

Franz Zieris
zieris@inf.fu-berlin.de
Matrikelnummer: 4138389

Betreuung: Prof. Dr. Lutz Prechelt

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Berlin, den 06. August 2009 _____

(Franz Zieris)

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation der Arbeit . . . 1
1.2 Ziel der Arbeit . . . 1
1.3 Aufbau der Arbeit und untersuchte Systeme . . . 1

2 Vorgehensweise 3
2.1 Methodischer Ansatz . . . 3
2.2 Technische Umsetzung . . . 3
2.3 Probleme und Lösungen . . . 4
2.4 Aufbereitung der Informationen . . . 6

3 Vorüberlegungen 7

4 Hardware 9
4.1 Koordination der Eingaben . . . 9
4.2 Hardwareseitige Behandlung von Interrupts . . . 9
4.2.1 Auslösen des Interrupts . . . 9
4.2.2 Rezeption des Interrupts durch die CPU . . . 10
4.3 Interrupt eines Tastendruckes . . . 12

5 Der Linux-Kernel 13
5.1 Aufgaben des Kernels . . . 13
5.2 Aufbau und Strukturierung des Kernels . . . 13
5.3 Anforderungen an die Interrupt-Bearbeitung unter Linux . . . 14
5.4 Implementierung der Interruptbehandlung in Linux . . . 14
5.4.1 Vereinheitlichung der Interrupt-Requests und Verteilung auf die Interrupt-Deskriptoren . . . 16
5.4.2 Abarbeiten der Action Chain . . . 17
5.4.3 Daten aus der Hardware auslesen . . . 18
5.4.4 Erstellen von Events . . . 20
5.4.5 Zustellen der Events an die Empfänger . . . 21
5.4.6 Behandlung der Events . . . 24
5.5 Nach dem Interrupt . . . 27
5.5.1 Die flush_to_ldisc()-Funktion . . . 27
5.5.2 Vermutungen über den weiteren Ablauf . . . 27

6 X Window System 28
6.1 Client-Server-Konzept 28
6.2 Fensterkonzept 28
6.3 Internationalisierung (I18N) 28

7 OpenOffice.org 30
7.1 Anforderungen an OpenOffice.org 30
7.2 Visual Component Layer (Eingabe) 31
7.2.1 Der Event Loop 31
7.2.2 Einlesen der Eingaben 32
7.2.3 Verteilen der Eingaben 36
7.3 OpenOffice.org Writer 38
7.3.1 Anpassung der internen Repräsentation 38
7.3.2 Berechnung der Änderungen 40
7.3.3 Weitergabe der Änderungen an den VCL 41
7.4 Visual Component Layer (Ausgabe) 44
7.4.1 Generische Behandlung 44
7.4.2 Ausgabe an das X Window System 45

8 Zusammenfassung 46

Literaturverzeichnis 47

Anhang A - Übersichtsgrafiken 49

1 Einleitung

1.1 Motivation der Arbeit

Office-Anwendungen sind von keinem Computer mehr wegzudenken. Ob beruflich oder privat: Jeder nutzt mehr oder weniger regelmäßig bestimmte Anwendungen, die den Arbeitsalltag oder das Hobby erleichtern sollen.

Dabei hat man sich schnell an die Zuverlässigkeit von Computerprogrammen gewöhnt. Viele Nutzer tippen beispielsweise mit 100 Anschlägen pro Minute und erwarten, dass die geöffnete Anwendung jedes ihrer getippten Zeichen ohne erkennbare Verzögerung auf den Bildschirm bringt. Sobald aber ein paar Zeichen nur leicht verzögert erscheinen, drängt sich bei vielen der Gedanke auf, dass irgendetwas mit dem Gerät nicht stimmt, denn schließlich hakt es ja! Bei einer Schreibmaschine konnte man noch sehen, wenn sich zwei Hämmerchen gegenseitig auf dem Hin- und Rückweg zum Papier verklemmt hatten. Dieser Mechanismus bereitet den Wenigsten Unbehagen, denn man sieht genau, was beim Tippen passiert und kann auch Notfalls manuell die Verklemmung lösen. Bei einem Computerprogramm sehen sich die meisten Menschen nicht derartig überlegen und fühlen sich leicht überfordert, wenn schon eine automatisch generierte geschachtelte Listenaufzählung nicht dem Willen des Nutzers folgt.

1.2 Ziel der Arbeit

Diese Arbeit soll einerseits das geheimnisvolle Innere eines solchen komplexen System exemplarisch an einem einzelnen Tastendruck in einem leeren Textdokument aufzeigen und erklären. Andererseits soll sie aber auch Ehrfurcht vor der großartigen Ingenieursleistung der Erbauer wecken.

1.3 Aufbau der Arbeit und untersuchte Systeme

Zu Beginn der Arbeit werde ich die allgemeinen Anforderungen umreißen, die ein heutiges System in Hinblick auf einen Tastendruck zu erfüllen hat. Daran anknüpfend werde ich die Zerlegung des Gesamtsystems in seine Komponenten vornehmen und diese abschnittsweise betrachten. Innerhalb jedes Abschnittes werde ich wiederum die Aufgabe

des Teilsystems beschreiben, dann eine konkrete Umsetzung erläutern und darin den Weg eines einzelnen Buchstaben nachzeichnen.

Für die unverzichtbaren Analysen der Quelltexte bieten sich ausschließlich offene Softwareprodukte an, sodass für diese Arbeit ein aktuelles Linux-System und die Anwendung Writer aus dem OpenOffice.org-Paket untersucht wurden.

Linux-Kernel

Die hardwarenahen Aspekte des Linux-Kernels beziehen sich auf eine x86-Architektur. Weite Teile des Kernel-Codes sind aber komplett unabhängig von der verwendeten Architektur, sodass diese Einschränkungen nur von geringem Ausmaß sind. Grundlage war der Linux-Kernel in der Version 2.6.28.9 [KERNEL].

OpenOffice.org

Bei der Analyse von OpenOffice.org habe ich lediglich die für den Betrieb unter Linux benötigten Teile betrachtet. Auch hier wurde aber beim Softwaredesign darauf geachtet, möglichst kleine Teile spezifisch für ein System zu entwickeln, sodass das Gros der Arbeit allgemeine Gültigkeit hat. Basis für die Analyse war OpenOffice.org 3.1 [OOOa] [OOOb][1].

[1] Der OpenOffice-Quellcode ist in sechs Teile gegliedert: `core`, `system`, `binfilter`, `l10n`, `extensions` und `testautomation`. Relevant für einen Tastendruck in OpenOffice.org Writer sind nur die Teile `core` und `system`, die anderen Teile stellen Filter für alte StarOffice-Format, Übersetzungen für alle unterstützten Sprachen, Erweiterungen sowie eine Testsuite für Entwickler zur Verfügung.

2 Vorgehensweise

2.1 Methodischer Ansatz

Mein Grundgedanke bei der Erkundung dieser riesigen Softwaresysteme war, dass ich einen Bottom-Up-Ansatz verfolgen wollte, also aus den unteren Tiefen des Systems hoch zur endgültigen Ausgabe auf dem Bildschirm. Da der Tastendruck als solcher sich ja zeitlich gesehen auch in dieser Richtung bewegt, erhoffte ich mir von diesem Ansatz die größte Klarheit. Das „A“ wird von der Hardware bzw. dem Menschen der sie bedient „erzeugt“ und beschreitet dann einen langen Weg durch ein komplettes Computer-System, um am Ende als eine Ansammlung von andersfarbigen Pixeln auf einem Bildschirm zu erscheinen.

Weiterhin stellte ich mir so auch die Suche im Quellcode wesentlich einfacher vor. Ähnlich wie Links im Internet sind Funktionsaufrufe unidirektional, sodass sich der Aufrufer nicht ohne Weiteres feststellen lässt. Eine rückwärtsgerichtete Exploration des Quellcodes mit der Suche nach Funktionen, durch die die aktuell betrachtete Funktion aufgerufen wird, erschien mir eher als Notlösung, wenn z.B. der konkrete Aufruf nicht eindeutig ist, weil die Funktion nicht namentlich benannt ist, etc.

Mir war dabei von Anfang an klar, dass ich durch eine ausschließlich statische Analyse des Quellcodes keine großen Sprünge machen würde. Für die dynamische Code-Analyse schien mir die Ergänzung der jeweiligen Quelltexte um strategisch sinnvoll platzierte Debugging-Ausgaben im Stile von *„Funktion xxx wird aufgerufen, übergebene Parameter: ...“* am günstigsten. Dieses Vorgehen war ich bereits aus der Entwicklung von Webanwendungen oder kleineren Desktop-Programmen gewöhnt.

2.2 Technische Umsetzung

Um die modifizierten Versionen der untersuchten Softwaresysteme auch ausführen und testen zu können, mussten diese natürlich auch übersetzt werden. Da ich nur über einen einzigen Computer volle Verfügung habe, meine Windows-Umgebung aber auch während der Kompilierzeit nutzen wollte, entschloss ich mich, eine virtuelle Maschine zu installieren, in der ich meine Änderungen testen konnte. Davon erhoffte ich mir gleich mehrere Vorteile:

- Virtuelle Maschinen lassen sich pausieren, sodass man den Kompilierprozess einfach anhalten und bei Belieben weiterlaufen lassen kann.
- Die Zustände der virtuellen Maschinen lassen sich speichern und wiederherstellen. Wenn also irgendetwas schief gehen würde, könnte man so das System einfach wiederherstellen.

Der letzte Punkt klingt zwar zunächst reizvoll, ist aber in der Praxis überflüssig. Beim Kompilieren des Kernels werden neue Pakete erstellt, die sich problemlos neben dem bisherigen Kernel installieren lassen. Sollte das Eigenkompilat also nicht funktionieren, kann man immer noch den alten Kernel booten und den Fehler beheben.

Für das Erkunden des Programmcodes habe ich mich aus reiner Gewohnheit für Eclipse entschieden, für das es auch eine passable C/C++-Version gibt. Zunächst hatte ich vor, einen SCP-Clienten damit zu beauftragen, das entfernte Verzeichnis in der VM mit dem Eclipse-Workspace synchron zu halten, sodass ich quasi direkt auf der VM, aber dennoch in Eclipse arbeiten kann. Diese Idee verwarf ich aber recht schnell wieder und installierte stattdessen lieber Samba in der virtuellen Maschine und gab den Ordner mit den Quellen im Netzwerk frei.

Fortan hatte ich immer zwei Eclipse-Umgebungen geöffnet: ein Eclipse für die lokalen Dateien, in dem ich schnell Sachen löschen oder ändern konnte und ein Eclipse, dass sich aus dem Netzwerkverzeichnis bediente und mir somit direkten Zugriff auf die Quellen lieferte. Hauptsächlich habe ich in meiner lokalen Version viele Makros gelöscht, die in anderen Projekten als dem aktuellen definiert waren, sodass sie als Syntax-Fehler gewertet wurden und so mitunter ganze Klassen nicht vom Eclipse-Code-Indexer erfasst wurden.

Auf der VM selbst lief nur eine Konsole, von der aus ich OpenOffice gestartet und die Ausgabe in eine Log-Datei umgelenkt habe. Die Steuerung der Kompiliervorgänge übernahm ich über eine SSH-Verbindung über Putty.

2.3 Probleme und Lösungen

Der Einsatz virtueller Maschinen barg aber auch einige Hindernisse. So habe ich z.B. die Datei mit der virtuellen Festplatte als „mitwachsend“ angegeben, sodass die Datei nur den Speicherplatz auf dem echten Datenträger verbraucht, der auch vom tatsächlichen Inhalt der virtuellen Festplatte gebraucht wird. Allerdings hatte ich diese Datei auf einer FAT32-Partition gespeichert, sodass die VM bei erreichen von 4 GByte den Dienst verweigerte. Glücklicherweise hat der Host der VM diesen Fehler abgefangen, und ich konnte die VM von einer NTFS-Partition unbeschadet weiterlaufen lassen.

Zeitweilig hatte ich Zugang zu einem Dual-Core-PC für die Ausführung der VM. Allerdings musste ich feststellen, dass VirtualBox in der freien Version nur einen Kern unterstützt, sodass der erhoffte Geschwindigkeitszuwachs beim Kompilieren der Quellen eher bescheiden ausfiel.

Der Hauptgrund, warum ich mich nach mehr Geschwindigkeit beim Kompilieren sehnte, war mein gewohnter Arbeitsrhythmus bei der Wartung von Webanwendungen. Hier konnte ich Debugging-Ausgaben einstreuen und quasi sofort das Ergebnis begutachten. Dass der Linux-Kernel beim ersten Durchlauf mehr als fünf Stunden kompilierte, verblüffte mich dann natürlich. Dass ich ein paar Wochen später mehr als 12 Stunden auf die Ausgabe den OpenOffice-Build-Prozesses warten würde, konnte ich da noch nicht ahnen.

Glücklicherweise ist dieses Problem nicht neu, und so wurden diverse Tools entwickelt, die die Kompilierzeit mitunter drastisch reduzieren können. Ich hab sehr erfolgreich das Programm CCache benutzt, das sich vor den echten Compiler klemmt und die Compiler-Aufträge und -ausgaben abfängt. Mit einem Hash aus Compiler-Optionen aus dem Aufruf, Ausgabe des Präprozessors und anderen Eigenschaften werden die Kompilate gespeichert und so bei wiederholten Kompilieraufträgen einfach aus dem Cache geladen. Die Einrichtung gestaltete sich mit dem Setzen von drei symbolischen Links auch sehr einfach. Dadurch schrumpfte die Kompilierzeit für den Linux-Kernel auf ca. eine halbe Stunde und für OpenOffice auf „nur“ vier bis fünf Stunden.

Bevor ich beginnen wollte, meine Debugging-Ausgaben in der Software zu verteilen, wollte ich die jeweiligen Systeme in ihrer Reinform kompilieren, um sicher zu gehen, dass eventuelle, später auftretende Fehler wirklich nur von meinen Änderungen herrühren. Zu meiner großen Überraschung schlug der Build-Prozess des Linux-Kernels beim ersten Versuch fehl. Bei der Suche nach dem Fehler stieß ich in der Build-Konfiguration des Kernels auf ein Modul, das ganz ähnlich wie die Fehlermeldung hieß. Ich deaktivierte es und schon ließ sich der Kernel komplett kompilieren.

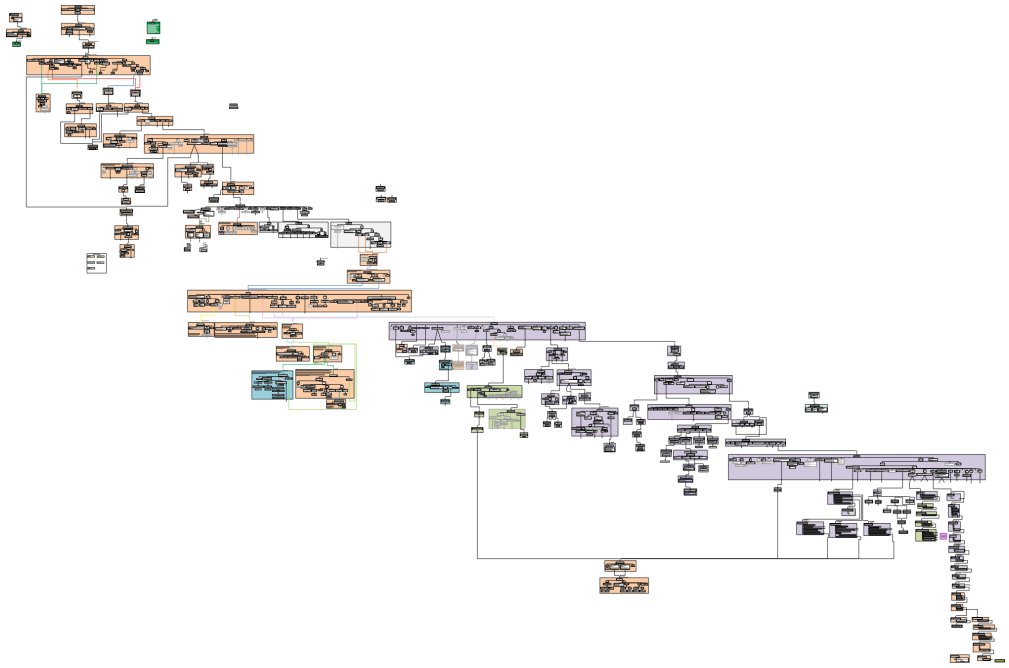
Bei OpenOffice stellte sich das Problem erst nach dem Build-Prozess heraus. Unter Ubuntu 9.04 ließ sich mein unverändertes, aber selbst kompiliertes OpenOffice nicht starten. Es stürzte immer ohne Fehlermeldung ab. Nach etlichen erfolglosen Versuchen fand ich in der Wiki von OpenOffice verborgen einen Hinweis auf eine fertig eingerichtete VM mit Ubuntu 8.10 für das Kompilieren von OpenOffice. In dieser VM funktionierte der Build-Prozess und die Programmausführung tadellos, aber die erstellten Installationsdateien wollten auf meiner eigenen VM nicht laufen. Ich wollte keine Zeit mit der Fehlersuche verschwenden, sondern habe einfach von da an zwei VMs benutzt: Eine für OpenOffice und eine für den Linux-Kernel.

Der Build-Prozess von OpenOffice wird auf vielen Seiten der OpenOffice-Wiki als sehr umfangreich und schwer verständlich beschrieben. Ähnlich verhält es sich mit der OpenOffice-Wiki an sich: es gibt keine erkennbare einheitliche Struktur, die sich durch alle Bereiche zieht. Für viele Teilgebiete gibt es mindestens drei angefangene How-Tos

und Einführungen, die maximal noch für OpenOffice 2.0 Gültigkeit hatten. Dieser Unübersichtlichkeit ist auch geschuldet, dass ich mehr oder weniger zufällig erst nach einigen Wochen Arbeit mit OpenOffice entdeckte, dass sich die vielen Einzelprojekte auch einzeln kompilieren lassen – und das sogar relativ einfach. Je nach dem zu kompilierenden Projekt bewegte sich nun die komplette Build-Dauer im Bereich von ca. 20 Minuten, was das Arbeiten wesentlich entspannte.

2.4 Aufbereitung der Informationen

Ich habe zu Beginn einige Zeit mit Darstellungsformen experimentiert, die mir einerseits möglichst viele Details und anderers auch einen guten Überblick bieten. Letztlich bin ich bei einer modifizierten Jackson-Notation angelangt, einer Art Baumstruktur für Abläufe und/oder Datenstrukturen. Ich habe für meine Zwecke in die Wurzel den Namen der Methode bzw. Funktion geschrieben und dann in der ersten Hierarchie-Ebene die Anweisungen geschrieben, die nacheinander ausgeführt werden. Eine Schleife oder eine Kontrollstruktur im Programmablauf führte dann zu einer tieferen Ebene. Um die gesamte Funktion legte ich dann einen Rahmen und färbte den Hintergrund entsprechend eines Ordners oder eine Paketes. Gegenseitige Funktionsaufrufe stellte ich durch Verbindungslinien dar, sodass sich aus den großen Funktionskästen wieder eine Baumstruktur ergab. Mein OpenOffice-Diagramm ist hier sehr stark verkleinert dargestellt:



3 Vorüberlegungen

Um zu verstehen, wie ein System funktioniert, ist es hilfreich, wenn nicht sogar notwendig, die Anforderungen zu kennen, die die Erbauer bei der Konstruktion erfüllen wollten.

Zunächst müssen einige grundlegende und scheinbar triviale Anforderungen erfüllt sein.

- Beim Druck auf die A-Taste wird erwartet, dass auf dem Bildschirm auch ein „A“ erscheint. Dafür muss das System natürlich ersteinmal mitbekommen, dass außerhalb des Rechners etwas passiert ist.
- Kaum ein Nutzer beschränkt sich ausschließlich auf die Tastur, um einen Computer zu bedienen. Eine Maus und andere Geräte müssen ebenso funktionieren.

Diese Anforderungen sind sehr grundlegend, sie sind größtenteils technisch und fallen daher in den Bereich der Hardware, also die elektronischen Bauteile aus denen ein Computer aufgebaut ist.

Der nächste Satz von Aufgaben sind vom Betriebssystem oder von unmittelbar aufsetzendener Softawre zu bewältigen. In dieser Arbeit wird beleuchtet, wie der Linux-Kernel das bewerkstelligt.

- Die bloße Beschriftung einer Taste hat natürlich keinen kausalen Zusammenhang zu den Buchstaben, die tatsächlich auf dem Bildschirm erscheinen. Diese Verbindung muss also im System hergestellt werden.
- Die Zuordnung von physischer Taste bzw. Tastenkombination zu einem Buchstaben (das sog. Tastatur-Layout) darf dabei nicht statisch sein. Kleine Unterscheidungen wie zwischen QWERTZ- und QWERTY-Tastatur sollen genauso behandelbar sein, wie komplett andere Strukturen wie z.B. die Dvorak-Belegung.
- Zwischen dem Tastendruck und der Reaktion des System soll möglichst keine merkbare Zeitspanne vergehen. Falls aber ein Programm z.B. durch eine hohe Prozessorlast kurzzeitig daran gehindert wird, sofort auf neue Eingaben zu reagieren, dann dürfen diese nicht verfallen, sondern müssen „nachgeholt“ werden.
- Um einen Text zu verfassen, sollte es unerheblich sein, ob man unterwegs einen Laptop oder im Büro einen Desktop-PC benutzt.

- Die geöffnete Office-Anwendung soll auch funktionieren, wenn der Nutzer mehr als ein Programm ausführt. Z.B. soll es problemlos möglich sein, neben der Arbeit auch Musik zu hören, oder ein E-Mail-Programm seine Postfächer kontrollieren zu lassen. Der Tastendruck darf dabei nicht irgendwo ankommen, sondern soll der Office-Anwendung zugestellt werden.

Eine Office-Anwendung, wie sie heute standardmäßig eingesetzt wird, hat unzählige Features, daher erhebt diese Zusammenstellung keinen Anspruch auf Vollständigkeit.

- Im konkreten Fall von OpenOffice soll die Anwendung im größtmöglichen Umfang plattformunabhängig sein. Aus Nutzersicht heißt das, dass sich die Anwendung auf einem Windows-System genauso verhält und darstellt wie auf einem Linux-System.
- Für die Entwicklung der Anwendung ist es u.a. ökonomisch vorteilhaft, wenn große Teile des Quellcodes für mehrere Betriebssysteme benutzt werden können, also nicht für jede Plattform eine komplett neue Anwendung programmiert werden muss. Konkret für Tastatureingaben folgt daraus, dass sie in einen Linux-Computer genauso behandelt werden können, wie unter Mac OS X oder OS/2.
- Der Nutzer erwartet eine Vielzahl von Möglichkeiten zur Gestaltung seiner Texte. Darunter fallen z.B. verschiedene Schrifttypen, -größen und -farben, verschiedene Textausrichtungen und -strukturierungen (Listen, Tabellen, etc.)
- Während der Eingabe soll eine Rechtschreibprüfung durchgeführt werden, evtl. Silben getrennt werden, automatisch Fehler korrigiert werden (häufige Buchstabendreher, Kleinschreibung am Satzanfang, etc.) usw.

4 Hardware

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

4.2 Hardwareseitige Behandlung von Interrupts

4.2.1 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

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

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]

4.2.2 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.

[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.

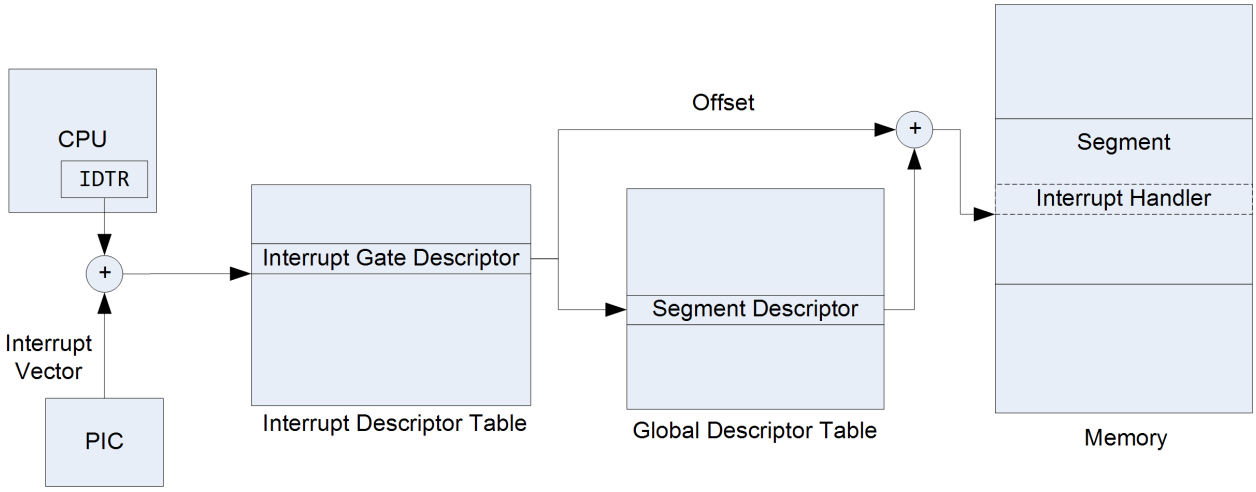


Abbildung 4.1: Ausführung eines Interrupt-Handlers auf Grundlage eines Interrupt-Vektors

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 4.1 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.

4.3 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.

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

5 Der Linux-Kernel

5.1 Aufgaben des Kernels

Der Kernel eines Betriebssystems hat zwei Hauptaufgaben zu erfüllen:

1. Verwaltung der Hardware
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[1]. [BC06, S. 8]

5.2 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]

[1] 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

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.

5.3 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.

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.

5.4 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.

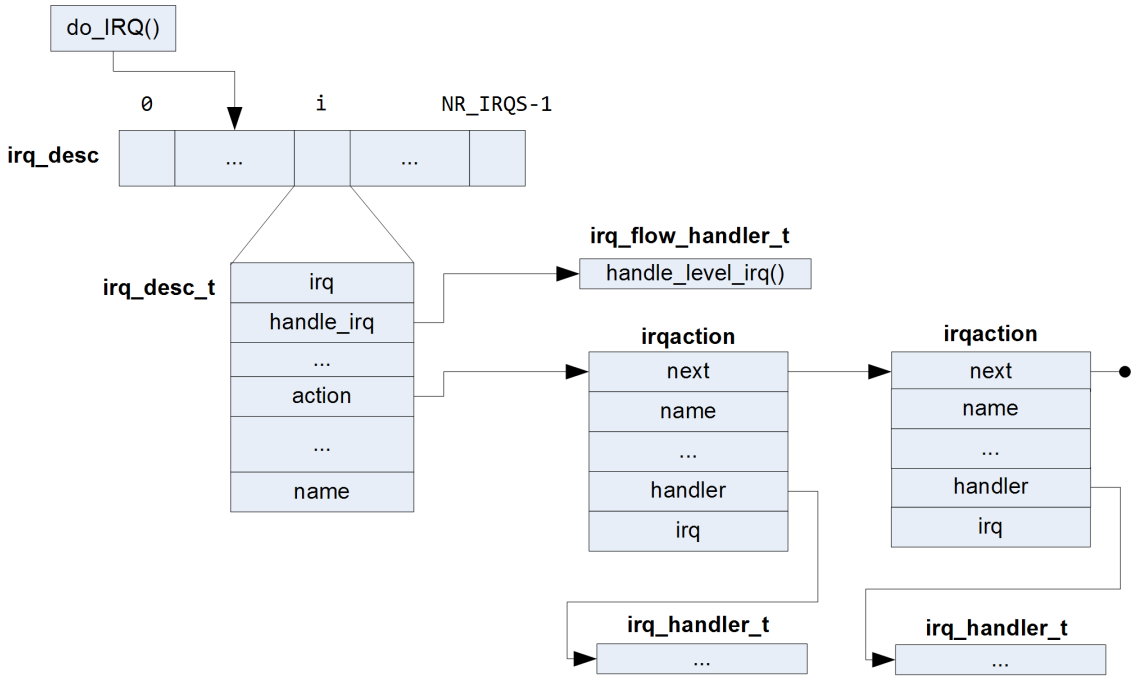


Abbildung 5.1: Zugriff der Funktion do_IRQ() auf die IRQ-Strukturen

5.4.1 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[2]. 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.

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

So kann und wird auf die Position zugegriffen, an der vorhin der bitweise negierte Wert für den IRQ abgelegt wurde.

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 5.1).

5.4.2 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[3] 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 Kernel-Moduls `i8042` angelegt [4]. 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 5.2 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.

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

[4] 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.

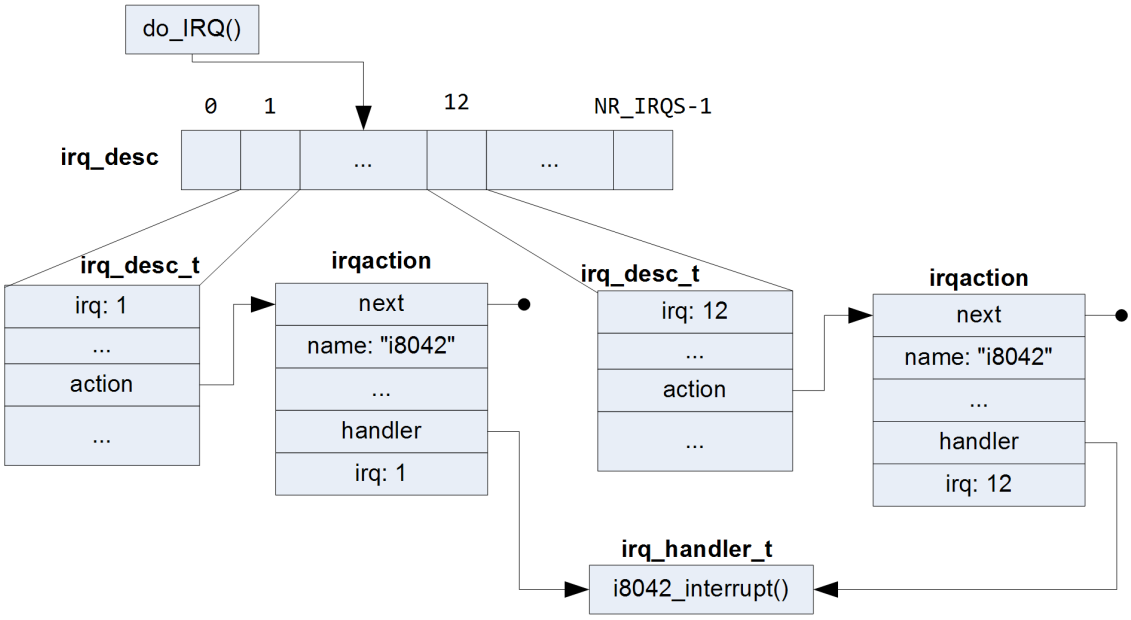


Abbildung 5.2: Verknüpfung von IRQ und Interrupt Service Routine bei Keyboard und Maus

5.4.3 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.

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 (00<u>0</u>1110<u>1</u>) 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.

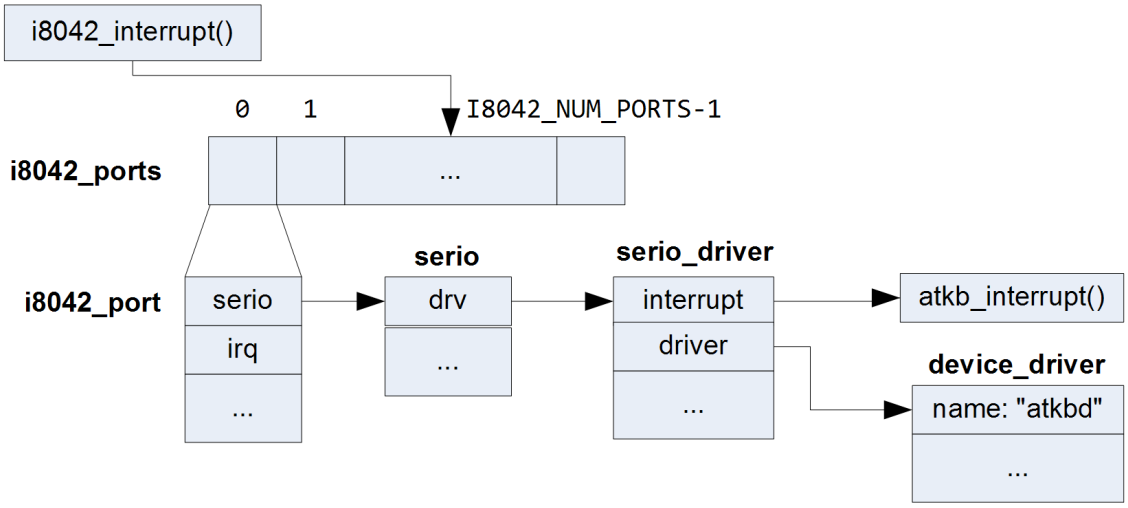


Abbildung 5.3: Schema der `i8042_ports` mit exemplarischer Darstellung des Keyboard-Ports

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[5].

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 5.3). 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 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.

[5] 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.

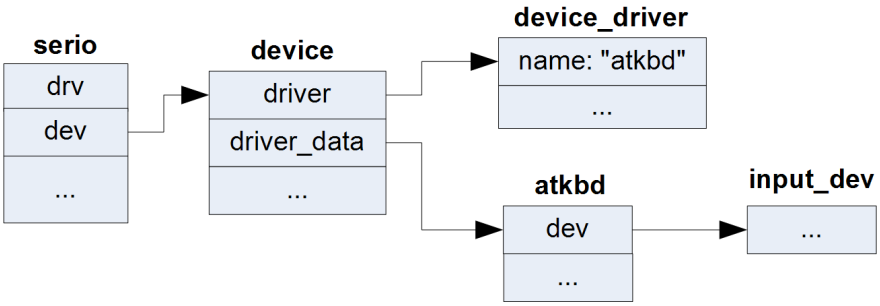


Abbildung 5.4: Zugriff auf das `input_dev`-Objekt über das `serio`-Objekt

5.4.4 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 5.4).

- 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[6] 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.

5.4.5 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 5.5).

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

[6] 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.

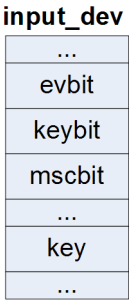


Abbildung 5.5: Auswahl der Bitmaps

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.

EV_MSC 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 5.5). 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.

EV_KEY Key-Events werden auf gleiche Weise geprüft, um auch hier auszuschließen, dass vom Gerät gar nicht unterstützte Events verarbeitet werden (Bitmap `keybit`, Abbildung 5.5). 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 5.5) 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 [7].

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

[7] 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.

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.

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 5.6).
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 5.7)

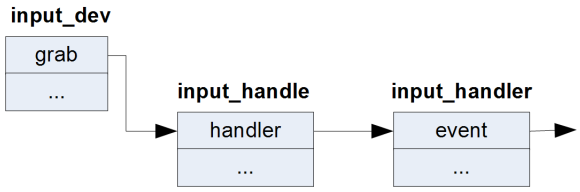


Abbildung 5.6: Zugriff auf ein Handle, das das Device für sich allein beansprucht

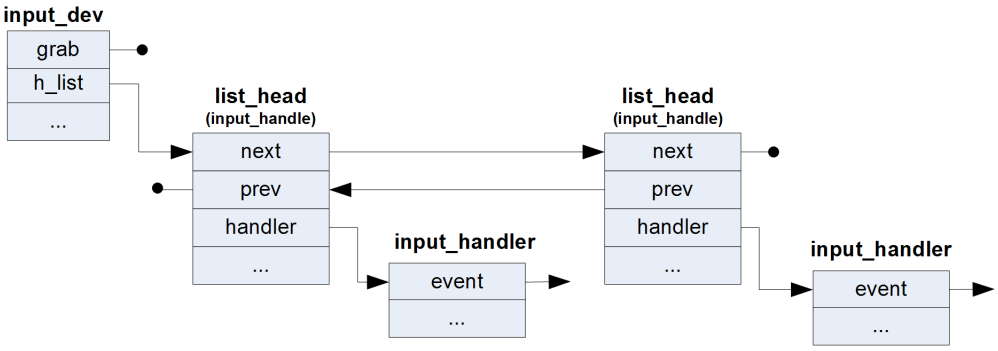


Abbildung 5.7: Zugriff auf mehrere Handles eines Devices

Das erste ausgelöste Event vom Typ EV_MSC mit dem Code MSC_RAW wird nicht an die Funktion input_pass_event(), also auch nicht an etwaige Handler weitergegeben.

Das zweite Event vom Typ EV_MSC mit dem Code MSC_SCAN hingegen wird an den Handler des ersten Eintrages der h_list, an ein Input-Handler namens kbd weitergereicht. Dessen event-Funktion ist die Funktion kbd_event().

Das dritte Event vom Typ EV_KEY wird ebenfalls durchgewunken. Und da es vom gleichen Gerät stammt, werden auch die gleichen Handler aufgerufen, also kbd_event().

5.4.6 Behandlung der Events

Die kbd_event()-Funktion

Die kbd_event()-Funktion ruft die Funktion kbd_rawcode() auf, wenn ihr ein EV_MSC-Event mit Code MSC_RAW übergeben wurde. Diese Events werden aber im hier skizzierten Fall schon von input_handle_event() ignoriert.

Wenn die Funktion ein EV_KEY-Event erhält, dann wird die Funktion kbd_keycode() aufgerufen.

Die kbd_keycode()-Funktion

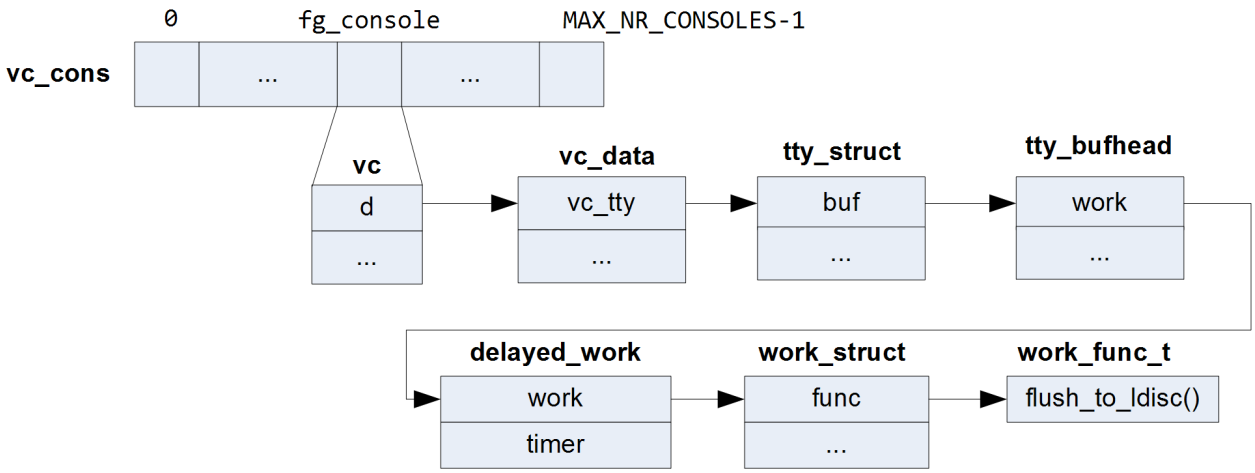


Abbildung 5.8: Verwaltung der Konsolen und ihrer Buffer

Die kbd_keycode()-Funktion wird einmal beim Drücken und dann noch einmal beim Loslassen der Taste aufgerufen. Ihre Aufgabe besteht darin, die so eingegeben Zeichen an die übergeordnete Schicht weiterzugeben. Diese Schicht befasst sich mit mehreren Konsolen, aber nur eine erhält auch die Eingaben. Je nach Art der Eingaben werden ein oder auch mehrere Zeichen in den Puffer der jeweiligen Konsole eingefügt.

In der `kbd_keycode()`-Funktion wird auf eine Liste von virtuellen Konsolen zugegriffen. `fg_console` ist dabei der Index der aktuellen Konsole (siehe Abbildung 5.8). Dieser Wert wird auf 0 initialisiert und beim Wechseln der aktuellen Konsole (z.B. durch `STRG+ALT+F1`, Wechsel zu `TTY1`) aktualisiert. Somit agiert die Funktion `kbd_keycode()` immer mit der aktuellen Konsole. .

Nachdem sich Zugriff auf die passende Konsole verschafft wurde, wird geprüft, ob der Nutzer einen sog. SysRQ (System-Request) abgesetzt hat[8]. Wird eine solche Eingabe registriert, dann wird eine entsprechende Hilfsfunktion `handle_sysrq()` aufgerufen, die dann die gewünschte Aktion ausführt. Durch die Einbettung in den Interrupt-Handler für Tastatureingaben, ist die schnelle Bearbeitung der Anfrage garantiert, selbst wenn das grafische System in einer Endlosschleife „hängt“.

Parallel zu den Konsolen wird auch eine Liste von Keyboard-Informationen gespeichert. Die Informationssätze und Konsolen sind dabei einander 1:1 zugeordnet, haben also in der Liste die gleichen Indizes. In diesen Informationen steht z.B. in welcher Form die erhaltenen Daten ausgewertet werden sollen. Zur Auswahl stehen dabei vier Modi:

`XLATE`	Die Keycodes werden mit einer Keymap übersetzt
`MEDIUMRAW`	Es werden die Keycodes benutzt
`RAW`	Es werden die rohen Daten, also die Scancodes benutzt
`UNICODE`	Die Zeichen werden in Unicode kodiert

Tabelle 5.1: Mögliche Ausprägungen von `kbdmode`

Bei den getesteten Keyboards wurde immer der `MEDIUMRAW`-Modus benutzt. Bei Keycodes, die kleiner als 128 sind, wird der Code beim Drücken unverändert und beim Loslassen mit einer 1 im hochwertigsten Bit verarbeitet. Bei dem „A“ ergeben sich dabei die Werte 30 (binär 00011110) beim Make und 158 (10011110) beim Break. Für größere Werte als 127 wird der Keycode auf drei Bytes aufgeteilt: das erste für die Angabe Make/Break, das zweite für die höherwertigen Bits und das letzte für die niedrigwertigen Bits.

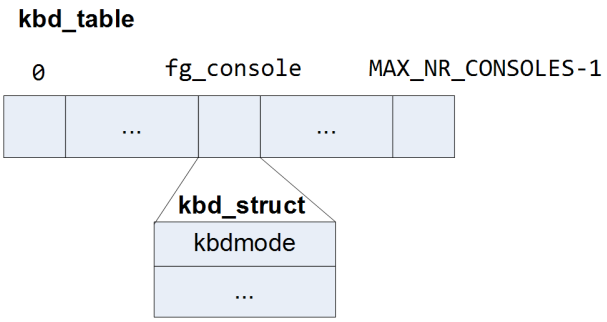


Abbildung 5.9: Verwaltung der Keyboard-Eigenschaften

[8] SysRQs (auf deutsch: S-Abf, System-Abfragen) sind in einigen Betriebssystemen vorgesehen, um mit dem Computer selbst dann noch kommunizieren zu können, wenn dieser scheinbar nicht mehr reagiert. Zu solchen Requests gehören unter anderem das Leeren des Schreib-Caches auf die Festplatte, das Beenden von Prozessen oder einen kontrollierten Neustart des Systems. Angesprochen werden sie über die Tastenkombination `ALT+SysRQ` bzw. `ALT+S-Abf` und einem weiteren Buchstaben, der die Art des Requests angibt.

Nach dem ein- oder mehrmaligen Aufrufen von `put_queue()` wird ein anderer Kommunikationsweg angeboten: die sog. *Atomic Notifier Chains*. Kleine Funktionen können sich über `register_keyboard_notifier()` in dieser Notifier Chain anmelden und werden dann *während der Interrupt-Behandlung* aufgerufen. Das schränkt den Funktionsumfang derartiger Funktionen natürlich ein, denn sie dürfen keinen Kontextwechsel herbeiführen, indem sie blockieren (siehe oben „Anforderungen an die Interrupt-Bearbeitung unter Linux“). Ich habe zwar keine Funktionen gefunden, die von dieser Schnittstelle Gebrauch machen, aber es ist zumindest vorgesehen.

Die put_queue()-Funktion

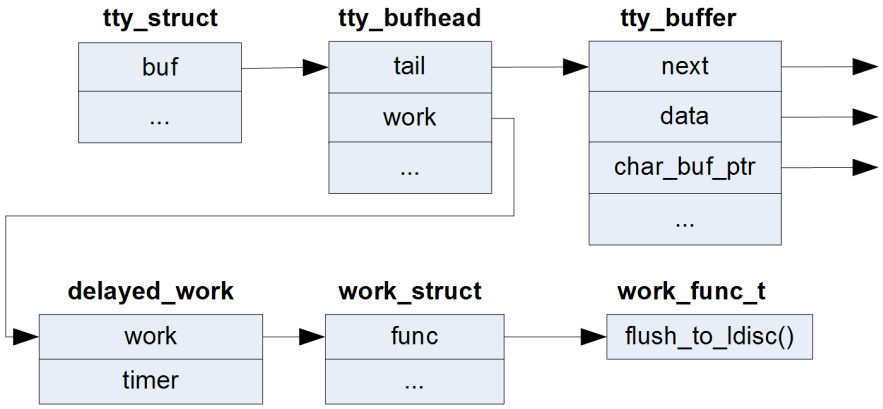


Abbildung 5.10: Zugriff auf den TTY-Buffer

Hier wird das gelesene Zeichen in den Buffer des TTYs abgelegt. Dafür wird erst `tty_insert_flip_char()` aufgerufen und dann `con_schedule_flip()`.

`tty_insert_flip_char()` schreibt das Zeichen an das Ende des `char_buf_ptr` und erhöht den Zähler, sodass andere Funktionen auch wissen, von wo bis wo die Daten gehen.

`con_schedule_flip()` ruft `schedule_delayed_work()` mit dem `delayed_work`-Objekt des `tty_structs` auf.

`schedule_delayed_work()` ist eine Funktion die ein `work`-Objekt in die allgemeine *Work Queue* einfügt. Die Work Queues sind ein Kernel-Konstrukt, mit dem Arbeiten vorbereitet und aktiviert werden können, die aber erst zu einem späteren Zeitpunkt ausgeführt werden sollen. Von daher sind sie Ideal zur Beendigung eines Interrupts-Handlers. Das eingegebene Zeichen ist im Puffer des aktuellen TTYs gespeichert und in der allgemeinen Work Queue wartet die Funktion `flush_to_ldisc()` darauf ausgeführt zu werden. Damit ist die Arbeit für den Interrupt-Handler getan.

5.5 Nach dem Interrupt

Das eingereihte `work_struct` wird zu einem späteren Zeitpunkt ausgeführt, indem seine `func`-Funktion aufgerufen wird. In der Funktion `tty_buffer_init()`, die eine neu erzeugte TTY-Struktur mit initialen Werten versorgt, wird die Funktion `flush_to_ldisc()` als `func`-Funktion gesetzt. Das gilt damit auch für TTY7, das standardmäßig nach dem Booten vom X Window System benutzt wird.

5.5.1 Die flush_to_ldisc()-Funktion

Die Funktion `flush_to_ldisc()` wird mit einem `work_struct` aufgerufen. Diese Funktion verschafft sich erst Zugriff auf den sog. *Line Discipline*, setzt dann eine Sperre auf den Puffer, sodass dieser nicht von anderen Prozessen verändert werden kann und setzt dann den Status des TTYs auf „FLUSHING“.

Der Puffer wird bis zum Ende durchgelesen, wenn nicht vorher ein Fehler auftritt. Dabei ist der entscheidenden Funktionsaufruf dieser: `disc->ops->receive_buf()`. `disc` ist der Line Discipline und `receive_buf()` ist ein nicht benannte Funktion. Dahinter könnte sich z.b. die Funktion `n_tty_receive_buf()` verbergen. Das konnte ich allerdings leider nicht mehr während der Bearbeitungszeit dieser Arbeit validieren.

5.5.2 Vermutungen über den weiteren Ablauf

Meine Vermutungen gehen dahin, dass der Line Discipline eine Implementierungsschicht für SysCalls wie z.B. `read()` beim Zugriff auf Dateideskriptoren, hinter denen sich Zeichenbasierte Gerätedateien verbergen, darstellt. Demnach müsste das X Window System versuchen, Zeichen von einer Gerätedatei zu lesen, was im Normalfall zur Blockierung der Anwendung führen würde, so lange keine Inhalte zum Lesen vorhanden sind.

6 X Window System

6.1 Client-Server-Konzept

Das X Window System ist als klassische Client-Server-Architektur entworfen worden. Der Client besteht in Form einer Anwendung, die sich der grafischen Bibliothek Xlib bedient. Der Server wartet auf Arbeitsanweisungen durch einen oder mehrere Clienten. Sämtliche Informationen über gezeichnete Fenster etc. befinden sich beim X Server, sind aber über entsprechende Xlib-Funktionen bei Bedarf auch für die Clienten abrufbar.

Mit der Xlib-Funktion `XOpenDisplay()` wird eine Verbindung zum X Server aufgebaut und eine Sitzung gestartet. Diese Verbindung erfolgt über ein Netzwerkprotokoll und wird im Normalfall innerhalb eines Rechners hergestellt. Es ist aber genauso möglich, diese Verbindung zu anderen Rechnern aufzubauen. [Brau03, S. 326]

6.2 Fensterkonzept

Die Xlib enthält nur sehr einfache Grafikfunktionen und Eingabeereignisse. Ausgehend von einem Grundfenster werden alle darin definierten Fenster nur innerhalb der Grenzen des Grundfensters angezeigt (*clipping*). Fenster sind hierbei einfache Rechtecke, die auf der Ebene von Xlib definiert sind. Jedes Fenster kann mehrere Unterfenster enthalten, wodurch sich eine hierarchische Baumstruktur ergibt, deren Wurzel das Grundfenster ist. [Brau03, S. 327f]

6.3 Internationalisierung (I18N)

Im X Window System gibt es das Konstrukt der sog. Input-Method. Die Input-Method erlaubt es, besondere Buchstaben durch eine Folge von Tastenanschlägen zu „erstellen“. Auf einer deutschen Tastatur bspw. findet sich keine Taste für das á (a mit accent aigu). Um dieses Zeichen tippen zu können, wird erst der Accent getippt und dann das „a“. Die Input-Method hält beim ersten Anschlag den Accent zurück und wartet auf weitere Zeichen. U.a. bei einem „a“ würde die Input-Method an dieser Stelle „zupacken“ und ein á erstellen. [Nye92a, S. 361]

Eine Anwendung, die von dieser Hilfestellunge Gebrauch machen möchte, muss zu Beginn der Programmausführung die Input-Method kontaktieren und dafür dafür `XOpenIM()` aufrufen. Dabei wird die Lokalisierung, die zu diesem Zeitpunkt im X Window System aktiv ist auch für die Input-Method gesetzt.

7 OpenOffice.org

7.1 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.

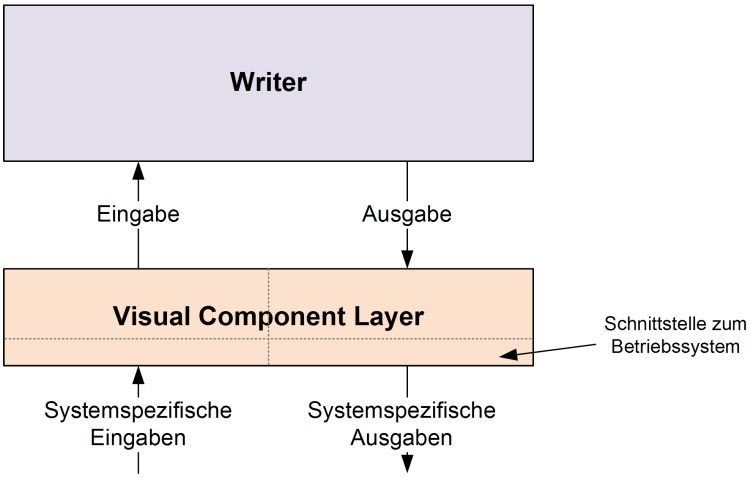


Abbildung 7.1: Grobe Struktur von OpenOffice.org Writer

7.2 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 7.2). [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.

7.2.1 Der Event Loop

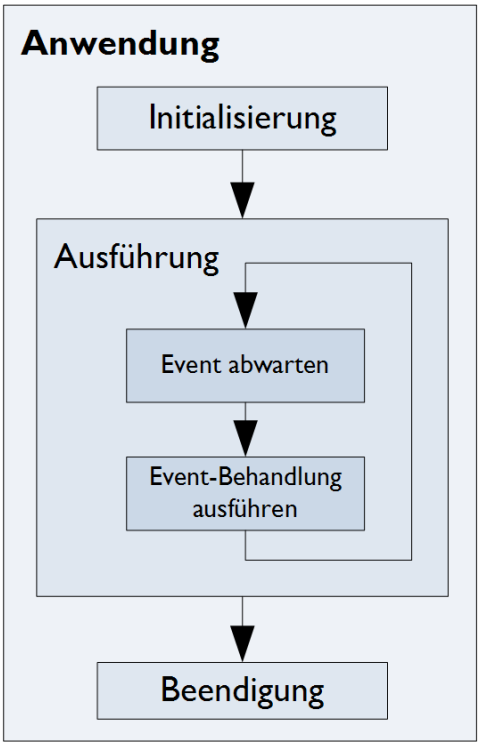


Abbildung 7.2: Schema einer Interaktiven Anwendung

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:

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.

7.2.2 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 7.3 dargestellt.

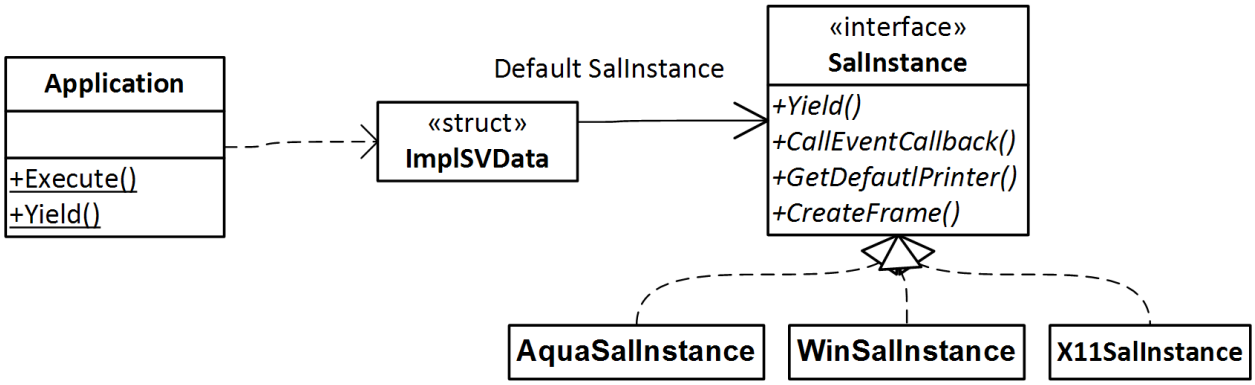


Abbildung 7.3: Abstraktion vom Betriebssystem (Auswahl an Schnittstellen-Funktionen)

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 `SalX11Display`-Objekt. Somit kann in den drei Yield-Funktionen auf das zugehörige Display zugegriffen werden.

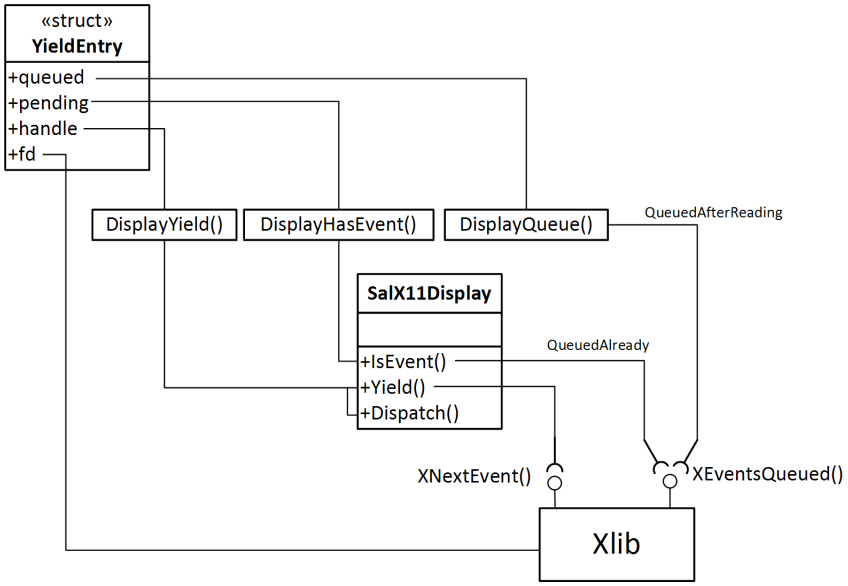


Abbildung 7.4: Abbildung der YieldFunc-Einträge auf Xlib-Funktionen

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 7.4):

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 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.

7.2.3 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 et-

waige 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.

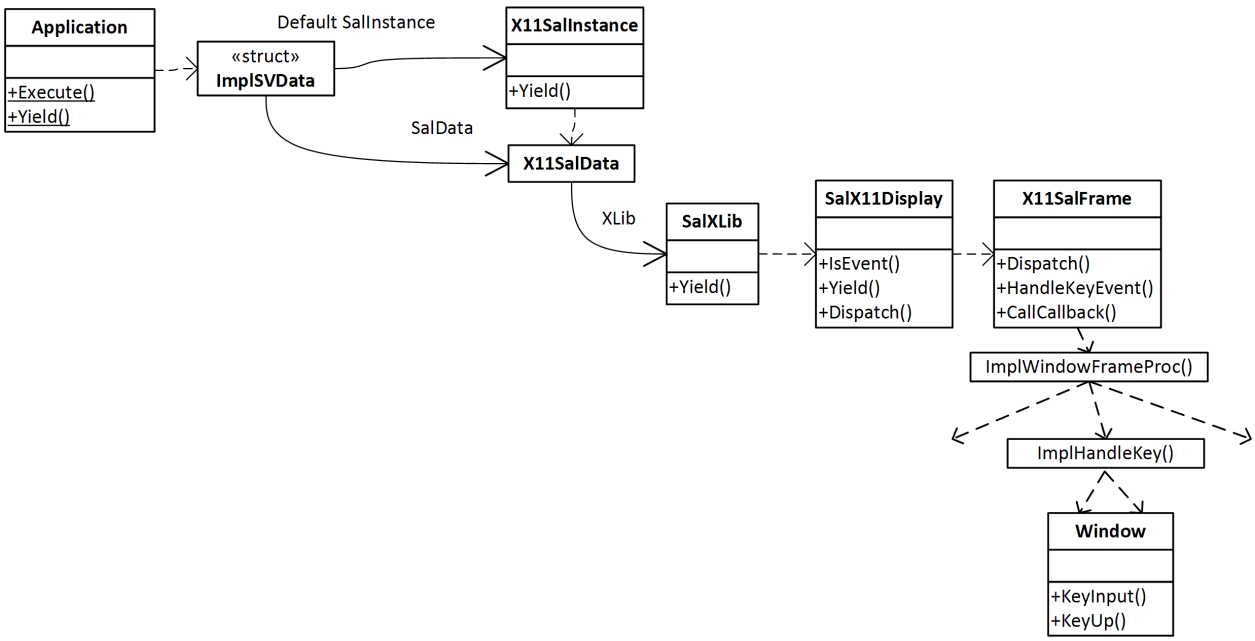


Abbildung 7.5: Event-Loop, Einlesen und Verteilen der Eingaben im Überblick

Der Zusammenhang der bisher passierten Klassen und Methoden ist in Abbildung 7.5 nochmal in einer Übersicht dargestellt. Dabei wurde auf die Darstellung der Interfaces verzichtet.

7.3 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 stammt, als OpenOffice noch StarOffice hieß. „SW“ steht dementsprechend für „StarWriter“.

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.

7.3.1 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 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.

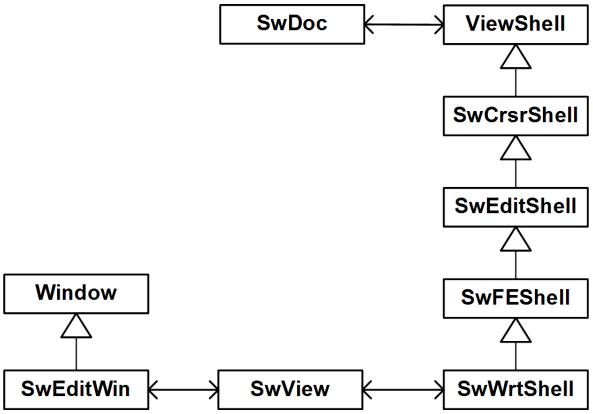


Abbildung 7.6: Verbindung von SwEditWin und SwDoc

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`.

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()`.

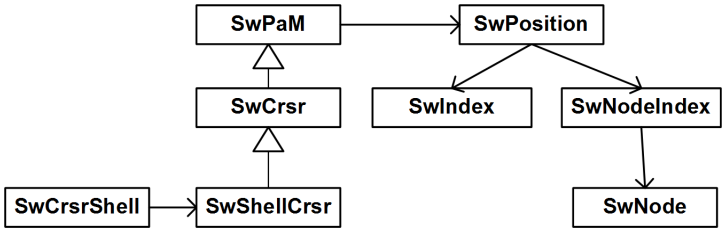


Abbildung 7.7: Verberbung und Assoziation eines Cursors

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 7.7 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.

7.3.2 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()`.

7.3.3 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[1].

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

[1] 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.

Zeichenobjekte, die unter dem Text-Inhalt liegen), dann der Dokument-Inhalt und dann den sog. „Himmel“ (Objekte über der Text-Ebene).

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.

7.4 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.

7.4.1 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.

7.4.2 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 DrawCairoAAFontString() 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.

8 Zusammenfassung

Ich habe es leider nicht geschafft, den kompletten Weg eines einzelnen Buchstaben innerhalb der Bearbeitungszeit von zwölf Wochen zu eruieren und darzustellen. Ich habe dennoch versucht, die Teile, die ich bearbeiten konnte, so gut wie möglich zu analysieren und zu erklären. Diese Aufteilung hat einen kleinen Vorteil für einen anderen Enthusiasten, der die Arbeit vervollständigen möchte: der oder die jenige muss sich nicht mit dem kompletten System befassen, mit all seinen Komponenten, sondern kann sich auf ein oder zwei konzentrieren.

Die Schwierigkeit dieses Themas lag in der genauen Abgrenzung nach unten, aber auch nach oben. Ist es interessant zu wissen, wie die Signale einer Tastatur in einen Computer gelangen? Ich finde ja! Ohne das Studium der Hardware-Details hätte ich sicherlich einige Stellen im Linux-Kernel nur mit viel Mühe verstehen können. Gerade die Berührungspunkte mit der Hardware, der architekturspezifische Teil des Linux-Kernels, setzen die Kenntnis der Abläufe in der Hardware voraus.

Ein großes Problem für mich war, die Größe des gesamten Systems abzuschätzen. Ich hätte bis vor wenigen Wochen nicht gedacht, dass OpenOffice eine derartig umfangreiche Codebasis hat. In dieser Arbeit habe ich lediglich *zwei* Quellcode-Ordner dargelegt. Meine Untersuchungen dehnten sich teilweise auf sechs oder sieben Ordner aus. OpenOffice hat aber in seiner Gesamtheit weit über 150 Ordner auf der obersten Ebene. Laut Ohloh (`http://www.ohloh.net/p/openoffice/analyses/latest`) umfasst OpenOffice derzeit knapp 10 Millionen Zeilen Quelltext. Der Linux-Kernel bewegt sich in ähnlichen Dimensionen (`http://www.ohloh.net/p/linux/analyses/latest`), aber nach meinem Empfinden ist der Linux-Kernel wesentlich klarer strukturiert und viel einfacher zu verstehen als OpenOffice.

Literaturverzeichnis

[Brau03] Rüdiger Brause, Betriebssysteme, 3. Auflage. Springer 2003

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

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

[GSS03] Garfinkel, Spafford & Schwartz, Pratical UNIX and Internet Security, 3rd Edtion. O'Reilly Media 2003

[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

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

[Nye92a] Adrian Nye, Definitive Guides to the X Window System, Vol. 1 Xlib Programming Manual. O'Reilly Media 1992

[Nye92b] Adrian Nye, Definitive Guides to the X Window System, Vol. 2 Xlib Reference Manual. O'Reilly Media 1992

[OOOa] OpenOffice.org 3.1 Core (Milestone OOO310_m11)
ftp://ftp.fu-berlin.de/unix/X11/apps/openoffice/stable/3.1.0/OOo_3.1.0_src_core.tar.bz2

[OOOb] OpenOffice.org 3.1 System (Milestone OOO310_m11)
ftp://ftp.fu-berlin.de/unix/X11/apps/openoffice/stable/3.1.0/OOo_3.1.0_src_system.tar.bz2

[OOOIDL] OpenOffice.org API.
http://api.openoffice.org/docs/common/ref/com/sun/star/module-ix.html

[OOOUno] OpenOffice.org Wiki. Understanding UNO.
http://wiki.services.openoffice.org/w/index.php?title=Uno/Article/Understanding_Uno&oldid=20638

[RC05] Rubini, Corber & Kroah-Hartman, Linux Device Drivers, 3rd Edition. O'Reilly Media 2005
http://oreilly.com/catalog/9780596005900/book/index.csp

Ch. 10 http://oreilly.com/catalog/linuxdrive3/book/ch10.pdf

[XLIBa] Xlib Programming Manual. O'Reilly & Associates, Inc. Ch. 2 „X Concepts“
http://www.sbin.org/doc/Xlib/chapt_02.html

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

Anhang A - Übersichtsgrafiken

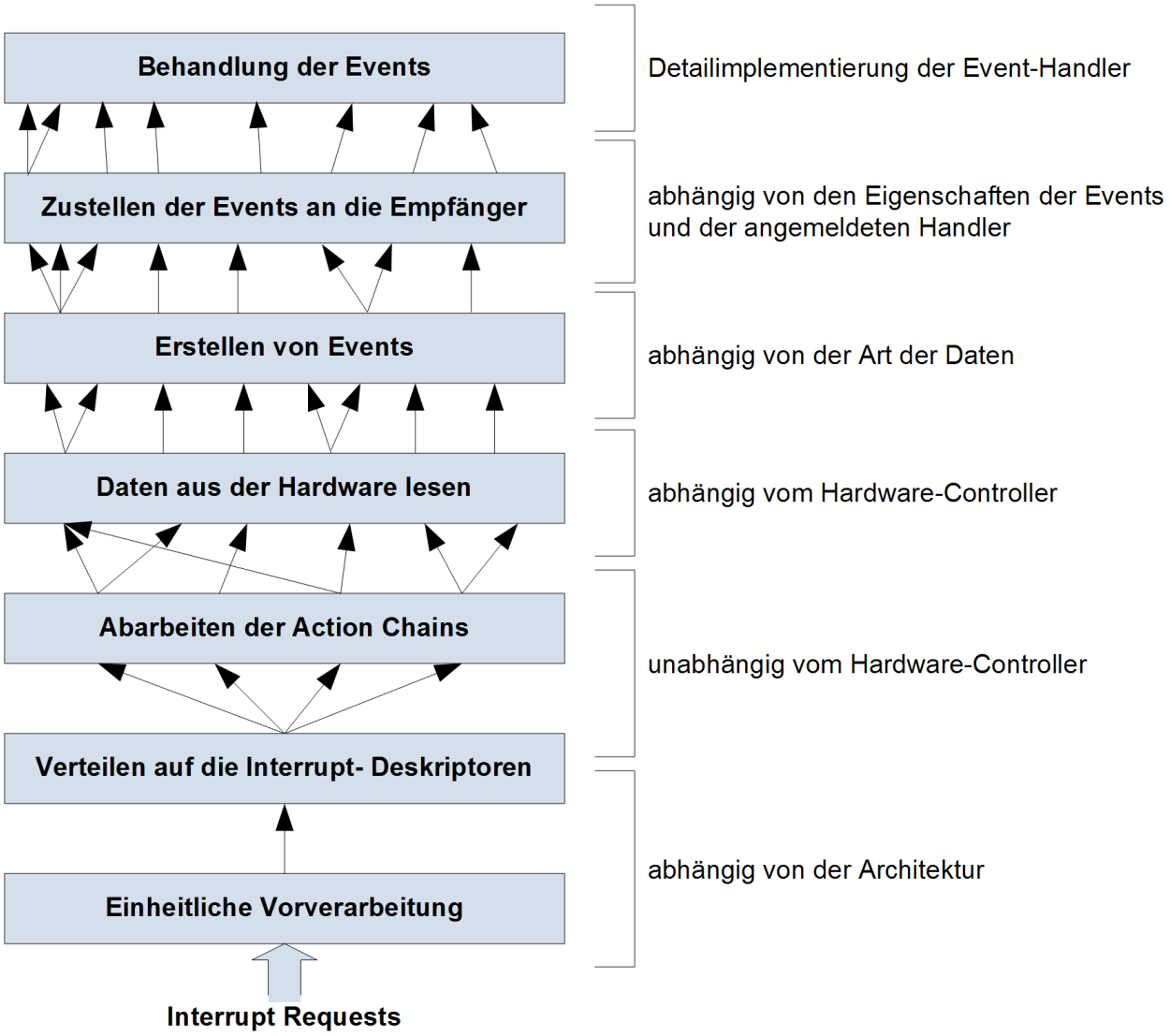


Abbildung 1: Schematische Interruptbehandlung im Kernel

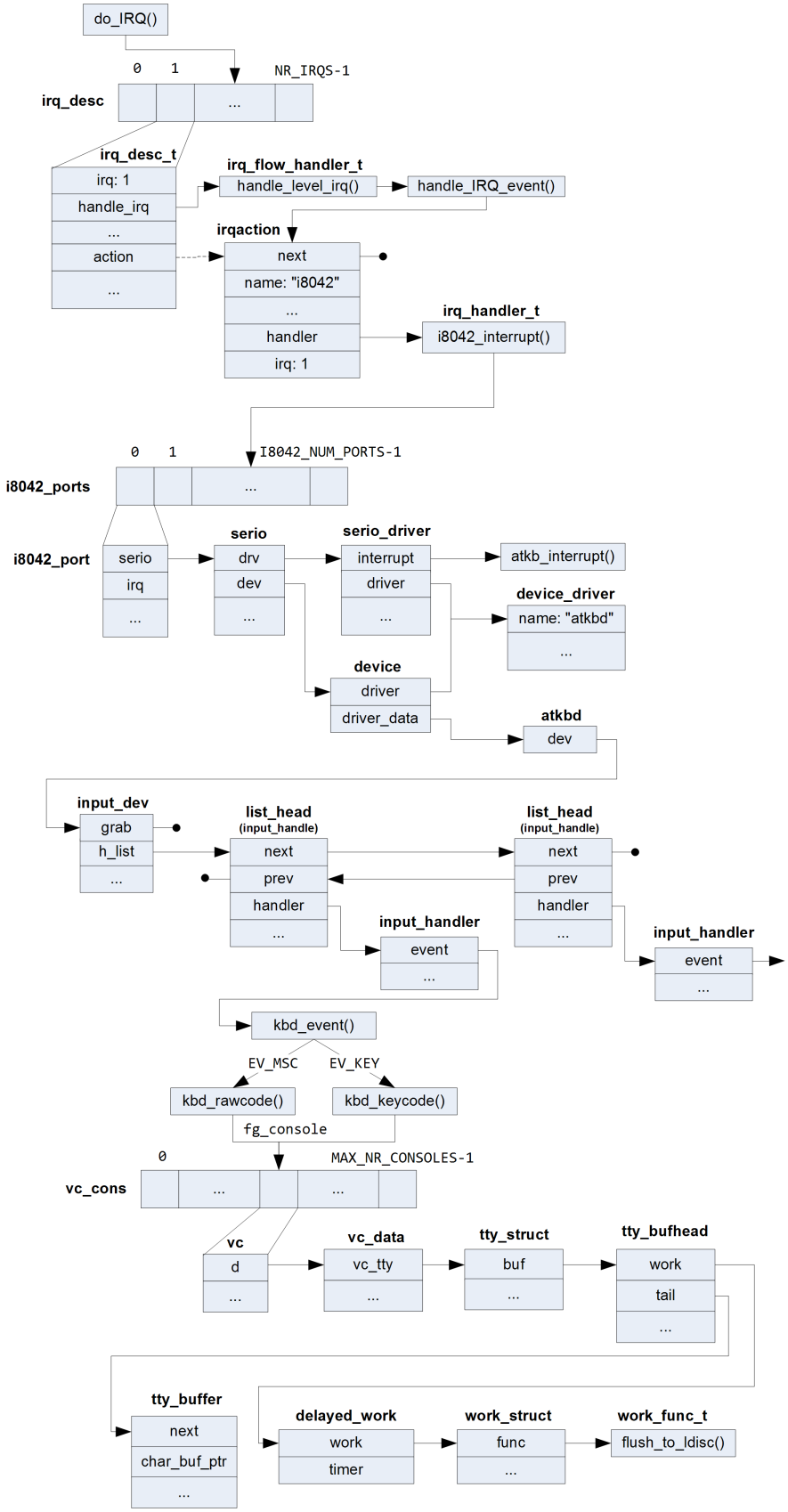


Abbildung 2: Der komplette Interrupt-Handler (Strukturen)

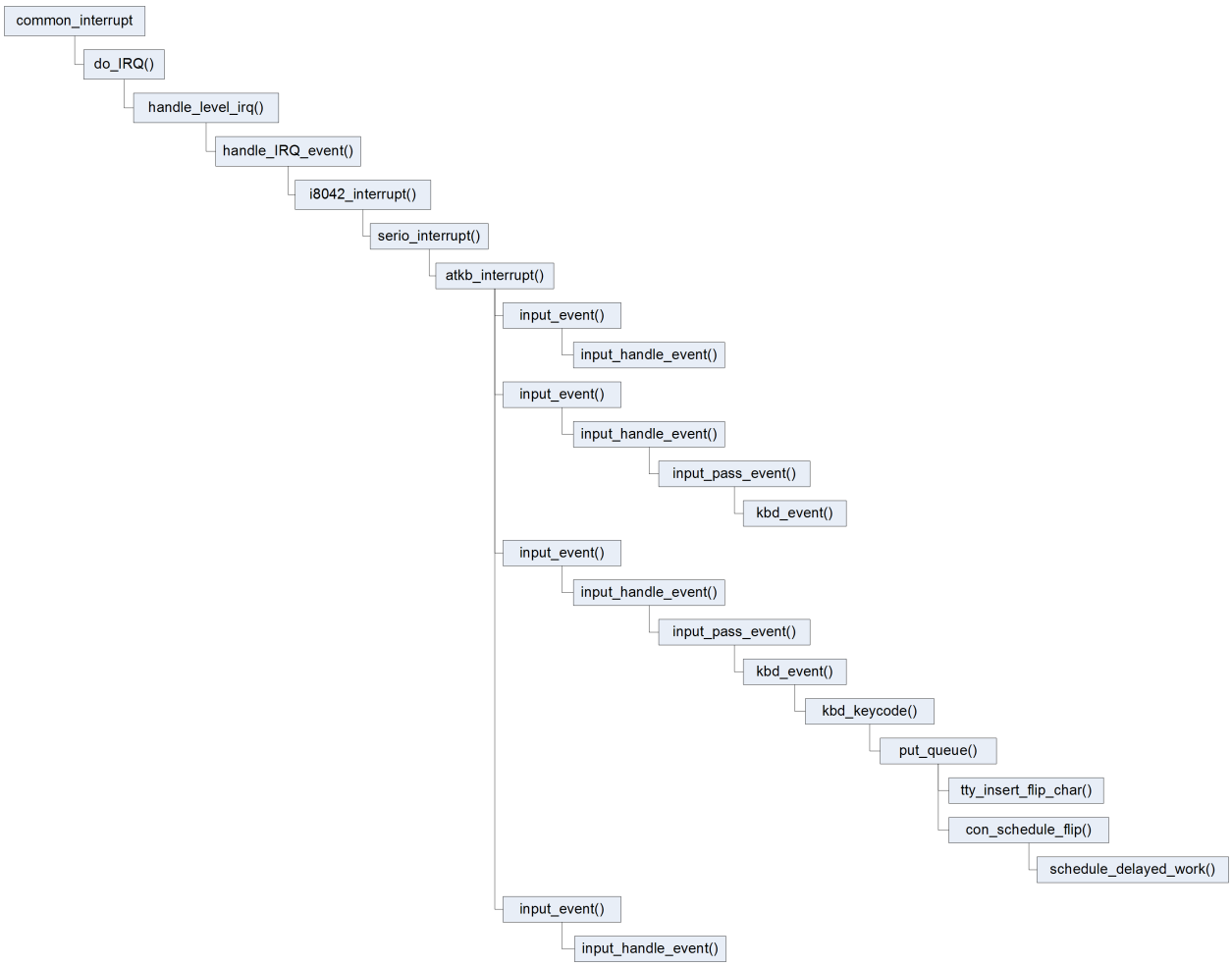


Abbildung 3: Der komplette Interrupt-Handler (Aufruf-Hierarchie)