Game & Emulation Development on GNU Linux

Written in LaTeXby Michael Durrer

[0] Last update on 1. Februar 2007

Inhaltsverzeichnis

Teil I

GNU Linux - Das fremde System

0.1 Vorwort

Sie haben dieses Buch gekauft, von meiner Seite als PDF runtergeladen oder sonstwie erhalten mit einem bestimmten Hintergedanken und einer Erwartungshaltung. Und vielleicht haben sich auch schon einmal eine der folgenden Fragen an den Kopf geworfen:

Wie programmiere ich ein Spiel oder eine Anwendung für Linux?
Wo(mit) fange ich überhaupt an?
Kann ich das überhaupt?

So ähnlich zumindest waren meine überlegungen, als ich vor vielen Jahren angefangen habe, mich mit der Interna von Rechnern und der Digitaltechnik herumzuschlagen. Seien Sie also nicht beunruhigt, jeder ist sich anfangs unsicher. Und in Erinnerung an die Problematik dieses Beginns (der heute in der riesigen Fülle an Informationen im Internet immerhin erleichtert worden ist), möchte ich dieses Buch all denen zur Verfügung stellen, die sich gerade inmitten jener Phase befinden und händeringend brauchbare Informationen suchen oder bereits diverse Grundlagen besitzen und darauf aufbauen möchten.
Vielleicht haben Sie bereits früher einmal versucht ein Spiel zu programmieren und es gelang Ihnen nicht, die Applikation fertigzustellen (übrigens eine der häufigsten Ursachen für den Abbruch eines Projektes: *Motivationsverlust*). Möglicherweise sind Sie auch schlichtweg an einer Stelle in Ihrem Workflow angeeckt und haben keine Lösung für das Problem gefunden. Gerade in solchen Fällen ist es mein Ziel, Ihnen mit diesem Buch lückenlos jede Stufe der praktischen Umsetzung von der Idee zur Applikation aufzuzeigen oder zumindest als Stütze dienen zu können. Aus den verschiedenen Beweggründen und Ursachen, die jemanden zu diesem Buch geführt haben, lassen sich verschiedene Typen herauskristallisieren und auf einen gemeinsamen Nenner bringen: So mögen Einige bereits programmieren können in C/C++, Andere haben gar noch nie etwas entwickelt aber würden gerne den Einstieg finden und wieder Andere wollen sich schlichtweg weiterbilden, zum Beispiel über SDL und Emulatoren. Aus diesem Grund habe ich dieses Buch in mehrere Parts unterteilt, welche sich an verschiedene Niveaustufen richten und Neueinsteigern wie Quereinsteigern einen Einstieg in die Thematik bieten kann. Diese Parts sind:

- Hardware-Grundkenntnisse und Wissenswertes

- GNU Linux - Das fremde System
- SDL - Der Simple DirectMedia Layer
- ANSI-C-Kurs für Anfänger
- C++ Kurs (aufbauend auf den ANSI-C Kurs)
- Die Grundlagen von Python
- Emulation - Fremde Systeme simulieren

Dabei ist der Schwerpunkt auf Linux gesetzt, doch habe ich auch durchgehend Wert auf Portabilität gelegt. Alle Programmbeispiele sollten auch unter Windows 2000 und höher laufen.
Speziell in der Spielewelt wäre eine häufigere Verwendung der SDL angebracht, ist sie doch portabel auf alle gängigen Plattformen wie Windows Vista/XP, Linux oder Mac OS X.
Portabel heisst in diesem Sinne, dass durch Verwendung von SDL für Steuer-, Eingabe-, Audio- und Videogeräte diese universal programmierbar werden. Eine einheitliche API (*Application Programming Interface*) steuert die Geräte nun an, der Programmierer sieht nur noch die API und deren Dokumentation. Spezifische Eigenheiten für jeweilige Geräte und Betriebssysteme, z.B. verschiedene Grafikkarten, übernimmt nun die SDL-Schicht komplett und übersetzt sie in die Sprache der jeweiligen Betriebssysteme und Hardware um.

Zusätzlich habe ich mich für dieses Buch auf 3 Programmiersprachen festgelegt: **C, C++ & Python**.
C/C++ ist quasi ein Industrie-Standard und wird seit Jahren in der Applikationsentwicklung da eingesetzt, wo man schnelle und übersichtliche Anwendungen benötigt, zu diesen gehören auch grafiklastige Applikationen. Seit einigen Jahren sind aber die Prozessoren mittlerweile an einem Punkt angelangt, wo sich die Entwicklung nicht mehr so stark an Geschwindigkeit multipliziert wie früher.

Die meisten Rechner sind heutzutage schon bei einer Geschwindigkeit angelangt, mit der die meisten Spiele auf dem Markt lauffähig sind, zudem wird immer mehr Rechenleistung auf die enorm leistugnsfähigen Grafikkarten ausgelagert, die auch auf langsameren Rechnern eine gute Grafikleistung hervorzaubern können und nicht mehr zwingend eine 'schnelle Programmiersprache' benötigen.

An dieser Stelle springt Python ein: Python ist eine skriptbasierte Sprache, die ebenfalls objektorientiert aufgebaut ist und somit eine exzellente übersicht

bietet. Desweiteren ist sie sehr einfach zu erlernen und kann ebenfalls mit SDL umgehen, es richtet sich daher eher an die Programmieranfänger, was jedoch nicht heissen soll, dass man damit nicht genauso komplexe Applikationen bauen könnte.
Wie auch immer Ihre Fähigkeiten und Ihr Wissen derzeit liegen, Ich hoffe Sie finden mit diesem Buch ein Themengebiet, dass Sie weiterbringt und Ihnen einen einfachen Einstieg in die Thematik ermöglicht. Beachten Sie jedoch bitte, dass ich Ihnen nahelege, gute Vorkenntnisse in ANSI-C oder C++ mitzubringen, da ich den Grossteil dieses Buches doch darauf stütze und auf jeden Fall früher oder später für professionelle Anwendungsentwicklung gelernt werden muss.

An den Teil über *Emulation*, sollten sich nur echte Profis ranwagen, welche C/C++ bereits in- und auswändig kennen. Auch wenn vielleicht nicht alles beim ersten Durchlesen verständlich sein mag für Anfänger, interessant ist es dennoch auf alle Fälle. Gleichzeiti wird Fach- und Hintergrundwissen vermittelt, welches unterstützend mitwirken kann beim Verständnis Abläufe in der Programmierung. Denn nur wer genau versteht, was sein Gerät spricht und tut, kann ihm genau befehligen, was und wie das Gerät es zu tun hat.

Einen Schwerpunkt habe ich beim Schreiben des Codes auf Übersichtlichkeit und einfachen Code, für Anfänger verständlich, gelegt, da es mir sehr wichtig war, Anfängern bereits die Möglichkeit zu bieten, komplexere Vorgänge wie Video- und Audioprogrammierung zwar nicht zwingend selber programmieren zu können, doch zumindest das Prinzip zu verstehen.

Neben dieser geballten Ladung an Information und Wissen, dass Sie in gebundener Form vor sich liegen haben, sollten Sie jedoch ein Punkt nie aus den Augen verlieren: Programmieren ist nicht nur lernen und arbeiten, sondern bereitet auch durchaus Entspannung und Spass! Und eben diese wünsche ich Ihnen nun mit diesem Buch und entlasse Sie in die spannende Welt der Linuxprogrammierung...

Ihr Michael Durrer

Kapitel 1

Hardware-Grundkenntnisse und Wissenswertes

In diesem Kapitel möchte ich auf einige Themen eingehen, die jeder (*Linux-*)Programmierer kennen oder zumindest einmal davon gehört haben sollte. Darunter fallen Sachen wie *Aufbau von Bits & Bytes, Berechnungen mit ODER-/UND-/EXODER-Tabellen, Ablauf von Programmen, Aufbau von Speicher und Hardware in einem PC-System* und vieles Weiteres. Ganz besonderen Wert lege ich auf emulationsrelevante Themen wie die **Struktur eines Mikroprozessors** oder die Wichtigkeit bei der Beachtung der *Byte-Reihenfolge* beim Ablegen/Auslesen von Adressen und Daten im Speicher auf Systemen mit **Big- und Little-Endian**.

Zwar möchte ich nicht auf alle Grundthemen eingehen, da dies nun wirklich den Rahmen des Buches sprengen würde, doch zumindest einige wichtige Grundbegriffe und Theorien sollten in diesem Kapitel schon vermittelt werden.

1.1 Bits and Bytes - Wie der Computer Zahlen liest und speichert

Wir wir wissen, ist der Speicher bei Computern in verschiedene Einheiten unterteilt, wovon die kleinste als **Byte** bezeichnet wird, welche wiederum aus 8 **Bits**, welche jeweils den Zustand *Wahr/True* oder *Unwahr/False*, also 1 und 0, besitzen können.

Ein Byte verwendet das binäre Zahlensystem und kann mit 8 Bits/Stellen insgesamt 256 verschiedene Zustände und davon nur einen gleichzeitig besitzen, d.h. von 0-255 zählen. Nach 255 (binär: 1111 1111) springt die Zahl wieder auf 0 (binär: 0000 0000).

Binärzahlen kann man zu einer Dezimalzahl umrechnen, indem man aus untenstehender Tabelle die Zahlen oberhalb der Spalte, wo unten eine 1 steht, alle zusammenzählt. In der linken Spalte stehen einige beliebige Zahlen. Versuchen Sie es selber mit einigen Zahlen, indem Sie von einer Binärzahl zu Dezimal und von einer Dezimalzahl zu Binär umrechnen.

Tabelle 1.1: Eine Rechentabelle für die Umrechnung von Binär- zu Hexadezimalzahlen

Zahl	128	64	32	16	8	4	2	1
255	1	1	1	1	1	1	1	1
127	0	1	1	1	1	1	1	1
56	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0

Wie wir in der Tabelle sehen, hat ein 1 je weiter es links steht einen umso höheren Wert. Dies geht immer so weiter, denn wenn wir beispielsweise eine Zahl speichern wollen, die höher als 255 ist oder mehr als 256 Zustände benötigt, dann brauchen wir bereits 2 Bytes. Mit 2 Bytes lassen sich bereits Zahlen
bis zu (2^{16} =) 65536 Zustände darstellen, respektive ($2^{16} - 1$ =) 65535 Zahlen darstellen.

Nehmen wir nun zum Beispiel die Zahl 65535 in hexadezimaler Kodierung: $FFFF Jedes F-Doppel steht für ein Byte; FF ist die höchste Zahl, die ein Byte annehmen kann, bzw. in hexadezimaler Kodierung mit einem Byte darstellbar ist: 255. Da die beiden Bytes nun aneinandergehängt wurden, haben die Bits des Bytes, welches die Bits 8-15 darstellt, eine höhere Wertung und es lassen sich so Zahlen bis 65535 darstellen.

Tabelle 1.2: Eine Rechentabelle für 2 Byte-Zahlen

Bit Nr.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zahl	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
65535	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
65534	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0
32768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
65280	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

1.2 Big- and Little-Endianness - Byte-Strukturierung im Speicher

Nun könnte man eigentlich annehmen, dass die Zahl $FF F0 = 65520 = 11111111 11110000 eigentlich folgendermassen irgendwo im Speicher stehen müsste:

Speicheradresse	$2000	$2001
$FFF0 im Speicher	FF	F0

Leider ist dem jedoch nicht (immer!) so. In den 70'er Jahren gab es bei der Entwicklung der ersten Mikroprozessoren verschiedene Hersteller, die die Mikroprozessoren auf unterschiedliche Art & Weise bauten, was die Assembler-Sprache anging als auch die Methode des Ein- und Auslesens von Adressen und Daten im Speicher.
So ergab es sich, dass Intel seit dem 8086er bis heute zu den x86er Modellen aufwärts den **Little-Endian-Standard** verwendet, während viele andere Prozessorhersteller wie z.B. *Motorola* auf den **Big-Endian-Standard** setzten. Gute Beispiele dafür sind die *PowerPC-Prozessoren* in den *Apple-Rechnern* oder in den *Amigas*.

Worin unterscheiden sich nun diese beiden Ansätze? Solange der Prozessor von einer Speicherstelle nur ein Byte auslesen muss, ist das noch nicht problematisch, da ein Byte immer auf dieselbe Art und Weise interpretiert wird, die Bits selber wechseln die Position ja nicht. Setzt man jedoch die Reihenfolge der Bytes um, so kehren sich natürlich die Zahlen um und dies führt natürlich zu enormen Problemen, wenn dies nicht rechtzeitig erkannt wird und der Prozessor richtig programmiert wird, im entsprechenden Endian-Stil.

Um die Unterschiede zu verdeutlichen, habe ich ein kleines Beispiel vorbereitet:

Speicheradresse	$2000	$2001
$FFF0 im Speicher, wie es bei Little-Endian abgelegt ist	F0	FF
$FFF0 im Speicher,wie es bei Big-Endian abgelegt ist	FF	F0

Wir sehen, dass bei Little-Endian der Wert nun umgekehrt als wie wir ihn von links nach rechts lesen würden im Speicher steht. Darauf weist auch der Name *Little-Endian* hin: Die niederwertigeren Bits zuerst (also links) nach rechts, wo am Ende das höchstwertige Bit steht. Bei *Big-Endian* ist es genau umgekehrt: Das Bit mit der grössten Wertigkeit kommt zuerst und liest sich, von links nach rechts genau gleich wie die Adresse $FFF0, wie wir es normalerweise lesen oder schreiben würden.

1.2.1 Anwendungszwecke

Vielleicht fragen Sie sich, weswegen wir das wissen müssen, da wir in C/C++ ja bei normalen Programmen sowieso nie damit konfrontiert werden, da die Hinterlegung im Speicher ja C/C++, bzw. der Compiler für uns übernimmt. Dies hat einen einfachen Grund: Zum Beispiel bei der Netzwerkprogrammierung, werden die Datenpakete auch teilweise im Big-Endian-Format umhergeschickt, so dass wir natürlich wissen müssen am Ziel-Rechner, ob unser Computer met dem Netzwerk-Endian-Format kompatibel ist oder nicht und wie wir diese Datenpakete auspacken und wieder zusammenbauen müssen. Ein weiterer Grund ist die Emulation von fremden Systemen/Mikroprozessoren, wie ich es im letzten Teil dieses Buches beschreibe. Dort werde ich einen Little-Endian-Rechner, den Intel 8080, emulieren und muss dementsprechend wissen, wie im emulierten Speicher der Prozessor seine Adressen hinterlegt, damit er die Daten richtig interpretieren kann.

Kapitel 2

Programmieren unter Linux

Teil II

SDL - Der Simple DirectMedia Layer

Kapitel 3

SDL-Initialisierung

3.1 Einige Hintergrundinformationen...

SDL steht, wie schon im Titel erwähnt, für die Simple DirectMedia Layer Library. Der Name sagt uns schon, dass uns SDL einen *simplen* und *direkten* Zugriff auf Medien, bzw. Multimedia-Hardware bieten soll.

Sie wurde entwickelt von Sam Latinga, während er bei Loki Software, bekannt für ihre Linux-Portierungen von Windows-Spielen, als leitender Programmierer angestellt war. SDL diente als Basis für die Portierung vieler Windows-Spiele, darunter *Civilization: Call to Power*und *Descent 3*, um nur Einige zu nennen.
Dank der vielen Fähigkeiten der Bibliothek und ihren OpenGL-Erweiterungen hat, vorallem aber auch dank der LGPL-Lizenzierung (mehr dazu im Kapitel **Open Source Lizenzen - Was ist das eigentlich genau?**), haben zu einer enormen Verbreitung dieser Entwicklungsbibliothek geführt. Mittlerweile entwickeln ein ganzer Haufen an professionellen als auch private Entwickler an SDL weiter und sorgen dafür, dass auch weiterhin aktuelle Technologien leicht ansprechbar bleiben über ein vereinfachtes Interface.

Durch die grosse Verbreitung ergibt sich noch ein besonders interessanter Vorteil: Die Plattformunabhängigkeit. Mittlerweile unterstützt SDL mehr als nur alle gängigen Betriebssysteme wie Linux, Windows und Mac OS X sowie die meisten Hochsprachen wie C/C++ sondern auch viele Nischen-Betriebssysteme und -Plattformen. Unter Anderem AmigaOS, SEGA Dreamcast, Microsoft XBox, Sony Playstation u.v.m.

Ganz im Gegensatz zu DirectX von Microsoft, welches ja alles Andere als cross-platform-fähig ist...

Unter Windows hat SDL zudem einen kleinen (selbst unverschuldeten) Haken: Microsoft lässt den Zugriff auf die Multimedia-Hardware, insbesondere Grafikkarten, nur über ihr eigenes Entwicklungs-Kit zu, nämlich DirectX. Somit kann auch die SDL-Schicht nur auf die DirectX-Schicht aufbauen und ist somit gezwungenermassen leicht langsamer unter Windows, wenn man das Programm mit einem Kompilat auf dem gleichen Rechner, jedoch unter einem anderen Betriebssystem testet. Dieser fällt jedoch nicht allzustark ins Gewicht und seien wir doch ehrlich:
Uns ist die einfache Programmierung und das Erreichen eines grösseren Zielpublikums durch Cross-Platform-Kompatibilität ein paar Frames pro Sekunde wert, oder?

3.2 Installation

Zuerst brauchen wir natürlich eine saubere SDL-Installation für unseren Compiler (in unserem Fall GCC).

3.2.1 Header-Dateien

Um SDL benutzen zu können, müssen wir natürlich in unserer Hauptdatei (z.B. *main.c* oder *main.cpp*) die SDL.h Header-Datei inkludieren:

```
#include <SDL.h>
```

Nun stehen uns alle Funktionen und Prozeduren der SDL-Welt zur Verfügung! Willkommen in SDL! Als nächsten Schritt müssen wir einen Modus initialisieren, zum Beispiel um Video-spezifische Sachen darzustellen den Video-Modus.

SDL-Erweiterungen

Es gibt für SDL noch eine ganze Reihe an Erweiterungen, die allesamt auf der normalen SDL-Bibliothek aufbauen. Eine kleine Auswahl:

Tabelle 3.1: Verschiedene hilfreiche SDL-Erweiterungen und Variationen

SDL-Erweiterung	***Beschreibung***	***Hinweise***
SDL_Image	Bild-Manipulation und diverse Grafik-Funktionen, liest versch. Bildformate ein	Win/Linux/OSX
SDL_Mixer	Audio-Kanäle mischen und Musikdateien abspielen (MP3, MIDI, MOD,...)	
SDL_Net	Netzwerk-Support für SDL, baut TCP/IP-Verbindungen auf, ideal für Spiele	
SDL_TTF	TrueType Font-Unterstützung	

Zu beachten ist, dass alle diese Erweiterungen SDL bereits voraussetzen. Besonders empfehlenswert, worauf ich auch in diesem Teil des Buches noch eingehen werde, ist die *SDL_Mixer Bibliothek*, diese gewährleistet Zugriff auf alle möglichen Grafikdateien und hilft uns, sie zu laden und zu schreiben.
Erhältlich sind die Bibliotheken (und viele andere Sachen) allesamt unter:

`http://www.libsdl.org`

3.3 SDL Modi initialisieren

SDL kann in mehreren Modi initialisiert werden, jedoch werden nicht immer alle benötigt. Dafür kann man mehrere Modi kreuzen. So braucht man beispielsweise in einigen Programmen gar keinen Audio-Modus oder es wird bei selbstablaufenden Programmen nichteinmal Input/Eingabe benötigt.

Deswegen kann man Ressourcen (und damit Leistung!) sparen, indem wir nur das initialisieren, was wir auch wirklich benötigen.

Als Allererstes müssen wir SDL selber initialisieren, wir nehmen ersteinmal den Video-Modus. Dafür benötigen wir den Befehl *SDL_Init()*:

```
int SDL_Init(Uint32 flags);
```

Wir ersehen daraus, dass wir sogenannte Flags übergeben, hier nehmen wir den Video-Modus:

```
SDL_Init(SDL_INIT_VIDEO);
```

Nun haben wir den Video-Modus initialisiert und können bereits andere Video-Modi initialisieren. Es gibt einige solcher Flags, hier die komplette Liste:

SDL_INIT_EVERYTHING	Alle Subsysteme gleichzeitig starten
SDL_INIT_INIT	Timer initialisieren
SDL_INIT_VIDEO	Video Subsystem initialisieren
SDL_INIT_INPUT	Eingabegeräte (Joystick, Maus, Tastatur) initialisieren
SDL_INIT_CDROM	CD-/DVD-ROM Subsystem initialisieren
SDL_INIT_JOYSTICK	Joystick Subsystem initialisieren
SDL_INIT_NOPARACHUTE	SDL fängt keine fatalen Signale mehr ab
SDL_INIT_EVENTTHREAD	Event Manager wird in einem separaten Thread gestartet

Besonders hinzuweisen ist noch auf *SDL_INIT_NOPARACHUTE*. Der SDL-Parachute („Fallschirm„) schützt vorzusagen vor einem Absturz, einer 'unsanften' Bruchlandung. Er fängt zuverlässig

3.3.1 Die ODER-Tabelle

Doch zuvor wollen wir noch sehen, wie man mehrere Modi miteinander kombiniert, wir möchten den Input-Modus zusätzlich! Dies bewerkstelligen wir, indem wir die Werte bitweise miteinander in einer ODER-Tabelle verknüpfen. Die ODER-Tabelle verknüpft Bits nach folgendem Schema:

Tabelle 3.2: ODER-Tabelle mit Eingangsvariablen A + B und Resultat C.

A ODER B	*C*
0 ODER 0	0
0 ODER 1	1
1 ODER 0	1
1 ODER 1	1

Das heisst, wenn mindestens eine der beiden Werte, bzw. deren jeweilige Bits die miteinander verknüpft werden, 1 ist, ist das Ergebnis (C) 1. So kann man gut Werte miteinander verknüpfen, bzw. fehlende Bits auffüllen.
Dazu müssen wir nun jedes Bit eines Wertes dem dazugehörigen Bit des anderen Wertes ODER-verknüpfen. Das ODER-Zeichen ist in C & C++ die Pipe: |

Ein kleines Beispiel:
Wir wollen Wert A mit Wert B verknüpfen; in der Tabelle steht das neue Resultat, dass wir in C dann erhalten.
In der Tabelle sind alle ODER-Fälle enthalten, es ist also transparent, wie der Prozess abläuft.

Zahlensystem	Binär	Dezimal	Hexadezimal
Wert A	0111 0001	71	113
Wert B	0000 1111	0F	15
Wert C	0111 1111	80	128

Teil III

ANSI-C-Kurs für Anfänger

Kapitel 4

Hintergrundwissen

4.1 Entwicklung und Geschichte von C

C gehört zu den sogenannten Hochsprachen und wurde in den 1970'er Jahren entwickelt von *Ken Thompson* und *Dennis Ritchie* in den Bell Labs. Ursprünglich, als eine der ersten Hochsprachen, gab es *ALGOL 60* die etwa 1960 entwickelt worden war von einem internationalen Komitee.

Daraus wurde dann *CPL* (**C**ombined **P**rogramming **L**anguage) geschaffen, woraus dann wiederum *BCPL* (für **B**asic **CPL**) entstand. 1970 entstand dann einer der direkten Vorgänger von dem, was später als (ANSI-)C zu weltweiter Verbreitung finden sollte: Simpel und schlicht **B** wurde es genannt.

Vielen ist bekannt, dass wir die Entwicklung der Programmiersprache *C* den Bell Laboratories zu verdanken haben, welche im Zuge Ihrer Planung und Entwicklung von *UNIX* das System später sogar in C schrieben. Denn zunächst entstand durch die beiden Entwickler *Ken Thompson* und *Dennis Ritchie* am *MIT* das *Betriebssystem UNIX*, und zwar noch in **Assembler**.

Da Assembler nicht gerade leicht gerade gut lesbar und verständlich ist aufgrund ihrer Struktur und der verwendeten Zahlensysteme, reifte in den beiden Entwicklern der Wunsch, eine einfache wie schnelle Programmiersprache zu entwickeln, welche nicht bloss auf einem Rechner lauffähig sein sollte, sondern auf jedem Rechner lauffähig sein sollte ohne grosse Anpassungen am Quelltext vornehmen zu müssen.

[0] Maschinensprache: Für den Computer direkt verständliche Codes (*Mnemonics*).

Der grosse Nachteil in Assembler lag nämlich besonders darin, dass für nahezu jeden Prozessor wieder ein teilweise komplett anderer Befehlssatz offeriert wurde, hinzu kamen andere Eigenheiten wie verschiedene *Byte-Reihenfolgen*

Da nun *UNIX* eben aus oben genannten Gründen nicht auf allen Systemen lauffähig war, beschloss der junge Wissenschaftler *Dennis Ritchie*, eine Sprache zu entwickeln, die für solche performantenwie portablen Applikationen, bzw. deren Entwicklung, geeignet war und schuf auf einer *PDP-11*, ein damals weitverbreiteter Grossrechner, im Jahr 1971 auf Basis der Programmiersprachen **BCPL** und **B** endlich die erste Version von ***C***!

Schon bald darauf folgte die Feuertaufe im Jahre 1972 für **C**: **UNIX** wurde nun von ***Assembler*** in ***C*** umgeschrieben und für den **PDP-11** veröffentlicht. In den nächsten Jahrzehnten sollte es sich auf der ganzen Welt rasant verbreiten und weiterentwickeln. Ursprünglich nur für Grossrechner und die Industrie gedacht, verbreitete es sich in den 80'er Jahren ebenfalls dank der ersten Umsetzungen für Intel-Rechner auf Home-Computer von Privatleuten, welche dank **UNIX** die Kapazität ihrer Heimrechner immer mehr ausnutzen wollten und erstmals konnten. Eine der erfolgreichsten, *unixoiden* (sprich: ***UNIX*** - ähnlichen) Betriebssysteme hat weltweite Verbreitung auf alle möglichen Arten von Prozessoren und Rechnern gefunden und ist uns heute bekannt unter dem Namen **Linux**!

Ganz im Sinne der Entwickler haben sich ***UNIX***, bzw. heute vermehrt auch ***Linux*** weltweit überwiegend auf Servern, aber auch auf Client-Rechnern, verbreitet und wurden auf alle möglichen Plattformen portiert. Nicht zuletzt verdankt man diesen grossen Erfolg der **GPL**-Lizenzierung [1], die einen grossen Teil zur Popularität und öffentlichen Beteiligung beigetragen hat durch Offenlegung der Quelltexte.

Und genau wie sein Urahn UNIX wurde Linux zu einem überwiegenden Teil in *C* geschrieben. Die meisten Distributionen[2] liefern heute standardmässig einen C-Compiler (üblicherweise den ***gcc*** → ***GNU C Compiler***) und den Quelltext zur Distribution mit; so wird dem Benutzer gleich von Anfang an eine Entwicklungsumgebung zur Verfügung gestellt, die ihm erlaubt, das System nach seinen Wünschen anzupassen oder zu verändern, bzw. sogar Fehler

[0] Lesen Sie dazu bitte das Kapitel →*Little Endian vs. Big Endian - Byte-Reihenfolgen und ihre Auswirkungen* und systemspezifische Arten der Speichernutzung.

[1] Mehr Informationen und Hinweise hierzu finden Sie im Kapitel "Programmieren unter Linux„

[2] Distributionen sind Zusammenstellungen von Linux-Software mit Linux als Kern(el). Sie werden von kommerziellen Betreibern und der privaten Linux-Community gepflegt und gewartet.

zu berichtigen und Features zu programmieren, welche er in die Distribution zurückfliessen lässt und somit zur ständigen Weiterentwicklung beiträgt.

Um nocheinmal zu C zurückzukommen: Durch die Vervielfältigung des sich ständig in Wandlung befindlichen Betriebssystems UNIX / Linux, gab es während den 70'er und 80'er Jahren keine eindeutige Definition, welchen Sprachumfang C nun genau umfasst.

So besitzt C zum Beispiel kaum eigene Funktionen, liefert jedoch im Standardumfang div. Funktionen wie z.B. **printf()** zur flexiblen Textausgabe mit, welches wohl jedem von "Hello World„-Programmen bekannt ist. Um einer weiteren wilden Verbreitung Einhalt zu gebieten, wurde vom **ANSI** (***A**merican **N**ational **S**tandard **I**nstitute*), einem Komitee zur Festlegung und Festigung von Standards in vielen verschiedenen Gegenständen und Technologien des alltäglichen Gebrauchs. Diese dienen zur besseren Interkommunikation und Zusammenarbeit zwischen Institutionen, Firmen und letztendlich Menschen und optimieren diese.

1988 gab es somit die erste Sprachbeschreibung des ANSI und ein Jahr später erreichte eben diese Sprachbeschreibung den offiziellen Status eines ANSI-Standards und wird seither als ***ANSI-C*** bezeichnet [wie in diesem Buch natürlich auch.]

4.2 Über ANSI-C und C++

Wie schon früher angedeutet und hingewiesen, ist **ANSI-C** eine flexible, portable Programmiersprache, die auf fast allen gängigen aber auch selteneren Betriebssystemen zur Verfügung steht und untereinander weitestgehend kompatibel sind, da die Sprache den *ANSI-Standard* von C hält und somit keine allzugrossen Differenzen bei der Funktionalität entstehen. Mit dem Erlernen von C investieren Sie in ein zukunftssicheres Wissen, welches auch in 20 Jahren in den Prinzipien noch seine Gültigkeit haben wird; desweiteren wird man ihre heute in *ANSI-C* entwickelten Programme ebenfalls noch viele Jahre lang lauffähig übersetzen können.

Durch die hohe Verbreitung von *C und C++* ist dieses Wissen in kommerzieller als auch, sogar insbesondere, in non-kommerzieller (*Open-Source, Hobby-Entwickler*) Hinsicht hochgefragt.

Was mit **Linux**, **UNIX** und **Mac OS X** (***Objective-C***), eignen sich diese Sprachen perfekt zur Systemprogrammierung. Darüberhinaus werden die C-Sprachen auch noch eingesetzt für Treiberentwicklung und für die Anwendungs-

entwicklung auf eingebetteten Systemen/Mikrocontrollern.

Die meisten der heutigen kommerziellen Anwendungen werden in ***C oder C++*** geschrieben. Vermehrt kommt seit einigen Jahren auch *Java* zum Einsatz, da im Gegensatz zu **C**/**C++** die Quelltexte von **Java** nur ein einziges einmal kompiliert werden müssen und Anschluss auf jedem System, welches über eine **JVM**[3] verfügt, in der Lage ist, die Programme laufen zu lassen.

Es ist heutzutage offensichtlich, dass immer mehr Wert darauf gelegt wird, dass Programme leicht auf mehreren Plattformen lauffähig sind, und, wie schon erwähnt, bieten **C** und **C++** genau diese Möglichkeit, den Quelltext ohne (grössere) Änderungen auf jeder Plattform zu übersetzen. Einer der weiteren Vorteile ist die geringe Speichernutzung, bzw. das optimale Speichermanagement, worin besonders C++ punkten kann. Nach C/C++ sind meistens nur noch Assembler-Anwendungen kleiner, welche jedoch nicht annähernd über dieselbe Flexibilität verfügen.

Um das Augenmerk auch noch ein wenig auf *C++* zu lenken: C++ verfügt gegenüber C viele Vorteile. C++ selber ist eine *Obermenge* von C, das heisst: Alle Elemente von C sind in C++ auch enthalten; darüberhinaus besitzt C++ jedoch noch viele weitere Fähigkeiten, die in der modernen Software-Entwicklung nicht mehr wegzudenken wären.

Dazu gehören unter Anderem:

- Objektorientierung: Bietet eine bessere Ordnung und Verwaltung von Projekten.
- Die Speicherverwaltung ist mit *new* und *delete* weitaus absturzsicherer, einfacher und sauberer geworden. Vorallem ist die Allokation von Speicher nun Hauptbestandteil von C++ selber.
- Die Objektorientierung bringt *Vererbung* mit sich; das bedeutet, dass Klassen nun (in C bekannt als Funktionen:)Methoden und (Variablen:)Attribute von anderen Klassen erben können, während man in C noch komplett neue Strukturen und Funktionen schreiben musste.
- Das Stream-Konzept: Daten werden in Form von Datenströmen, Streams, gelesen und geschrieben

[3] Java Virtual Machine - Eine virtuelle System-Umgebung, in der Java-Anwendungen ablaufen

- Konstruktoren und Destruktoren: Beim Erzeugen und Löschen einer Instanz einer Klasse, werden selbstdefinierte Funktionen, den Konstruktor und den Destruktor, ausgeführt. Ideal um z.B. dazugehörigen Speicher gleich mitzulöschen, damit er nicht im Speicher hängenbleibt.

- Polymosphirmus: Darauf gehe ich in C++-Kurs näher ein, da das hier sonst den Rahmen sprengt.

4.2.1 C-Standards und dieses Buch

Ich habe in diesem Buch bewusst den ANSI-C-Standard **C99** durchgehend benutzt; dabei muss man weiterhin unterscheiden zwischen verschiedenen Revisionen des Standards. So gab es neben dem ursprünglichen (nicht-ANSI-)**K&R-C** (logischerweise *Kernighan & Ritchie-C*) von **1978** die ANSI-Varianten **C89, C90** und **C99l**; die Zahl steht dabei stellvertretend für die Jahreszahl der Veröffentlichung
Dabei profitierten die späteren Revisionen von den Neuerungen seines weiterentwickelten Bruders **C++**, von dem viele Funktionen und Definitionen zurückflossen nach **C**.

Einige kleine Beispiele von **C99** im Vergleich **K&R-C**:

- Der zeilenweise Kommentar wurde ergänzend zum altbekannten "**/* ... */**"von **C++** übernommen.

- Die Variable **void** wurdei ns Repertoire der Sprache aufgenommen.

4.3 Datentypen

Wenn wir programmieren, müssen wir temporär Daten in Form von Bytes im Speicher lagern. Diese bekommen einen Platz, **Variable**, zur Aufnahme zugewiesen. Dabei müssen wir beachten, *wie* wir sie ablagern möchten und wie der Computer sie zu verstehen hat. Deswegen müssen die Variablen in bestimmten Datentypen deklariert werden. Zudem brauchen auch Funktionen[4] einen Datentyp, denn eine Funktion gibt nachdem sie aufgerufen wurde einen Wert zurück, nämlich das erwartete Resultat der Funktion.

[4]Sequenzen von Programmcode irgendwo im Speicher, die man immer wieder aufrufen kann.

Wenn wir zum Beispiel eine Funktion haben, die uns eine Zahl multplizieren soll, dann wissen wir, dass wir eine Ganzzahl (Integer) zurückbekommen möchten, wenn wir normale Dezimalzahlen verwenden. Wichtig zu wissen ist für uns auch, wieviele Bytes eine Variable zugewiesen bekommt im Speicher, damit wir wissen, wieviel Speicher/Variablen wir eines Typs benötigen um z.B. eine Datei komplett in den Speicher laden zu können.

C verfügt über eine ganze Reihe von Datentypen, die in verschiedener Konstellation mit zusätzlichen Attributen deklariert werden können. In den folgenden Unterkapiteln gehe ich auf jeden enizelnen Datentyp und seine Verwendung genauer ein anhand eines analytischen Beispiels. Um Ihnen zuvor einen besseren, groben Überblick zu verschaffen, habe ich die Datentypen in einer Tabelle zusammengefasst:

Tabelle 4.1: Eine Tabelle mit C-Datentypen und ihrer Anwendung.

Datentyp	Schlüsselwort	Bytes	Verwendung	Zahlenrau
Buchstabe(Character) mit Vorzeichen	signed char	1	Buchstaben und Sätze	-128..1
Buchstabe(Character) ohne Vorzeichen	unsigned char	1	0..255	
Ganzzahl(Integer) mit Vorzeichen	long int	8		
Ganzzahl(Integer) mit Vorzeichen	int	4		
Ganzzahl(Integer) ohne Vorzeichen	short int	4		
Fliesspunktzahl(Floating Point) nur mit Vorzeichen	float	4		
Fliesspunktzahl(Floating Point) nur mit Vorzeichen	double	8		

4.4 Das erste Programm

Für Ihr erstes C-Programm, können Sie untenstehenden Quelltext verwenden; diese Anwendung liest über die **Standardeingabe (STDIN)**, in Ihrem Fall die Tastatur, einen String ein. Jener String darf maximal 15 Zeichen betragen, da nicht mehr Speicher alloziiert worden ist, und wird nach der Eingabe mit Return in das Programm übergeben, welches diesen Satz anschliessend auf der **Standardausgabe (STDOUT)**, dem Bildschirm, ausgegeben.

```
/* ReadString.c - Ein Programm, welches einen String einliest und über den Bildschirm ausgibt.
*/
\newlabel

#include <stdio.h> // Einbinden der Standard-Ein-/Ausgabe-Funktionen.

int main (int argc, char *argv[])
{
char stringPhrase[16];  // String aus 15 Zeichen + 1 Zeichen für das Terminierungssymbol ('\0)
scanf("%s",&stringPhrase[0]); // Einlesen des Strings von Tastatur
printf("Ihre Eingabe lautete: %s",stringPhrase); // Eingabe ausgeben auf Bildschirm
return(0); // Programm erfolgreich ausgeührt, Gebe Wert zurück
}
```

Teil IV

Der C++-Kurs

Teil V

Emulation - Fremde Systeme simulieren

Kapitel 5

Von Emulatoren und Nostalgikern

5.1 Definition

Bestimmt haben Sie schonmal von **Emulatoren** gehört, jenen Anwendungen, die auf Ihren PC alte Konsolen oder Computer, realitätsgetreu in Ihrer (Re)Aktion, herbeizaubern und es Ihnen ermöglichen, alte Applikationen und Spielanwendungen wieder zu benutzen und zu erleben. Doch was sind Emulatoren genau per Definition? Und was passiert da genau, was macht ein Emulator?

Ersteinmal möchten wir dem Wort auf den Grund gehen: *Emulation]* leitet sich vom lateinischen Wort für nachahmen ab, welches *aemulari* lautet. In der EDV wird das funktionelle Nachbilden eines Systems auf / durch einem/eines Anderen bezeichnet. Ein Emulator ist folglich ein System, welches ein anderes System nachahmt und komplett gleich reagiert auf Eingabe & Verarbeitung der eingeführten Daten.

Das nachbildende System erhält dieselben Daten, führt dieselben Programme und `muss` die gleichen Ergebnisse erhalten wie das Originalsystem. Durch die sich immer schneller entwickelnde EDV-Technik der letzten Jahrzehnte kamen mitte der 90'er die ersten Software-Emulatoren. Moment mal, Software-Emulatoren? Wenn es Software-Emulatoren gibt, muss es auch Hardware-Emulatoren geben, und jene gab es natürlich: Zunächst in den 60'ern und 70'ern. Damals kamen Geräte in den Handel, die zum Beispiel kompatibel waren mit dem damals

sehr populären Atari 2600, sie bildeten sein Verhalten elektronisch nach mittels anderer Hardware. Damals war die Hardware noch zu langsam, um soetwas über die Software zu realisieren.

Ein Nachteil dieser ganzen Geschichte liegt darin, dass ein Emulator nie 100% gleich agieren kann wie das Original, auch wenn die Abweichungen nahezu irrelevant und nur minim sind. Somit bleiben gewisse Anwendungen / Spiele Nostalgikern und Spiele-Freaks heutzutage immer noch verschlossen vor der Verwendung auf einem Fremdsystem. Zugegebenermassen hat aber die Qualität bei Emulatoren mittlerweile dermassen zugenommen, dass man gute Chancen hat, für seine bevorzugte Applikation ein passendes Programm zu finden, welches fähig ist, die Applikation fehlerfrei auszuführen.

5.2 Anwendungszwecke von Emulatoren

Für Emulatoren gibt es haufenweise Anwendungszwecke; bei einigen mag sich der Sinn zwar nicht unbedingt allen erschliessen, doch sind sie nicht umsonst sehr begehrt für jedes mögliche zu emulierende System. So verwendet man Emulatoren unter Anderem für folgende Anwendungszwecke:

- Weiterverwenden von Software für ein anderes (Betriebs-)System nach der Migration in eine neue Umgebung.
- Softwareentwicklung - Ein Programm kann ohne Gefahren für die eigene Hardware nach Belieben getestet und entwickelt werden im geschützten Rahmen der Emulation.
- Freizeitvergnügen - Viele Leute spielen gerne im Emulator die Spiele von einst auf ihren modernen PCs.
- Ergonomisches Arbeiten - Man kann seinen Arbeitsplatz an ein System verlagern, welches ergonomische Hardware bietet und das Zielsystem emulieren kann, so dass man nun ergonomisch arbeiten kann.
- Applikationen dank einiger Zusatzfunktionen schneller ausführen lassen im Emulator, als wenn Sie auf dem echten Gerät laufen würde.

5.3 Aufbau dieses Parts

Sie sehen also, es gibt durchaus einige vernünftige und sinnvolle Zwecke zur Entwicklung und Verwendung eins Emulators. Nicht zuletzt kann es eine sehr lehrreiche und interessante Sache sein, einen Emulator selbst zu entwickeln und zu verwenden. Ein Emulator ist rein vom programmiererischen Aspekt her eine sehr anspruchsvolle Arbeit und erfordert viel Verständnis für Aufbau und Abläufe eines Mikroprozessors und sonstiger Hardware. Daher habe ich auch entschlossen, diesen Part in meinem Buch zuletzt zu platzieren, da hier alles Vorwissen aus jedem vorherigen Part hier zusammenläuft und praktische Verwendung findet. Ohne tiefgreifende Kenntnisse der zu emulierenden Hard- und Software wird man jedoch seine Mühe haben, hier auch nur ansatzweise zum Erfolg zu kommen.

Diesem Buch beiliegend finden Sie eine komplette Dokumentation inkl. Quelltexte zu meinem Intel 8080 Emulator. In diesem Teil des Buches wird auch umfassend beschrieben, welche Entscheidungen und Design-Wege ich getroffen und gegangen bin, um den Emulator möglichst akkurat zu gestalten. Jeder Schritt soll für den Leser nachvollziehbar und für andere Systeme vom Prinzip her anwendbar sein.
Es empfiehlt sich daher sattelfest in C++ und SDL zu sein und ein gutes Vorwissen mitzubringen im Bezug auf Emulation, Mikroprozessoren (Assembler), Aufbau des Arbeitsspeichers U.Ä.