Denkmaschinen

Was Computer für uns leisten können und sollten

Richard Kofler

1. Zahlensysteme, Bits, Bytes und Datenspeicher

Mit äußerst wenigen Ausnahmen haben die allermeisten Menschen jeweils zehn Finger und zehn Zehen. So ist es wohl nicht verwunderlich, dass sich in der Menschheit das Zehnersystem (Dezimalsystem) mit den Ziffern 0, 1, 2, ... 8 und 9 durchgesetzt hat, wo doch kleinere Zahlensysteme den Nachteil haben, dass größere Zahlen sehr schnell in ihrer Länge anwachsen und größere Zahlensysteme bedingen, dass man den Überblick über eine Unzahl von Ziffern bewahren muss.

So beinhaltet der Zahlenwert 1024 einen Tausender, null Hunderter, zwei Zehner und vier Einser oder in mathematischer Schreibweise:
$1024D = 1*1000 + 0*100 + 2*10 + 4*1 = 1*10^3 + 0*10^2 + 2*10^1 + 4*10^0$
(Sprich: 10 hoch 2, etc.; die Hochzahl (Potenz) bedeutet z.B., dass bei 10^3 die Zahl Zehn drei mal mit sich selbst multipliziert wird.)
(Die Reihenfolge geht immer von links (höchstwertige Ziffer) nach rechts (niederwertigste Ziffer).)

Ein Computer kennt aber als kleinste Speicherzellen nur Bits, die die Werte NULL (= nicht gesetzt = FALSE) und EINS (= gesetzt = TRUE) annehmen können.

Schreibt man jetzt $1024D = 10000000000B = 1*2^{10} + 0*2^9 + 0*2^8 + 0*2^7 + ...$ (Die Buchstaben „D“ und „B“ stehen hier für dezimal (Zehnersystem) und binär (Zweiersystem).), so wird die dargestellte Binärzahl schnell unübersichtlich.

So hat sich für maschinennahe Anwendungen am Computer die Hexadezimalschreibweise (Sechzehnersystem) durchgesetzt, wo jeweils vier Bits zu einem „Nibble“ (einer „Hexziffer“) zusammenfasst werden und zu den Dezimalziffern 0 bis 9 noch die Hexziffern A(10), B(11), C(12), D(13), E(14) und F(15) hinzukommen. Dabei merkt man sich in der deutschen Sprache am leichtesten, dass C und 12 mit „ts“ anfangen und D und 13 mit „d“.

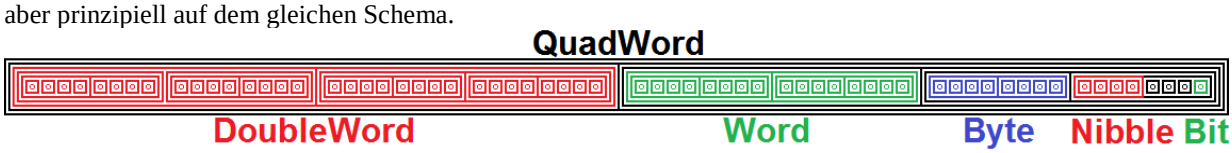
So gilt jetzt einfacher 10000000000B = 400H (sprich: „vier-null-null hex“).

Zwei solche „Nibbles“ werden heute auf nahezu aller Computerhardware zu einem „Byte“ zusammengefasst und man misst Speicherkapazitäten und Datenübertragungsraten praktisch immer in irgendwelchen Anzahlen von Bytes.

Ein Byte ist also acht Bit oder zwei Nibble und kann die (256) Werte 0D bis 255D oder 00H bis FFH annehmen.

Nun muss man wohl in den Computern auch vorzeichenbehaftete Zahlen darstellen können. Dafür verwendet man ganz einfach das höchstwertigste Bit – wenn dieses gesetzt ist, handelt es sich um eine negative Zahl und der Wertebereich des Bytes geht von -128D bis +127D.

Nun will man am Computer nicht nur Zahlen, sondern auch Buchstaben darstellen und zu diesem Zweck wurde der „ASCII-Code“ entwickelt. Dieser definiert ganz einfach bestimmte Zahlenwerte im Byte als geschriebene Großbuchstaben, Kleinbuchstaben, Ziffern, Sonderzeichen oder Steuerzeichen. So entspricht der Zahlenwert 35H = 53D der geschriebenen Ziffer „5“. Moderne Zeichensätze beinhalten viel mehr verschiedene Zeichen, beruhen aber prinzipiell auf dem gleichen Schema.



Wendet man nun irgendwelche Rechenoperationen auf Bytes an, hat man sehr bald den Wertebereich in negativer oder positiver Richtung verlassen, und daher entwickelten sich aus den allerersten 4-Bit-Prozessoren mit der Zeit bis jetzt 64-Bit-Prozessoren mit den folgenden Datentypen: (ganz in Analogie zur Evolution, wo aus den anfänglichen Bakterien so komplexe Lebewesen wie Menschen entstanden – wer weiß schon, was sich aus diesen Prozessorminiaturen noch alles entwickeln wird?)

Ein Word (einfaches Wort) besteht aus 2 Bytes, damit aus 16 Bits und hat den Wertebereich 0 bis 65535.

Ein DoubleWord (Doppelwort) besteht aus 2 Words (4 Bytes), damit aus 32 Bits und hat den Wertebereich 0 bis 4.294.967.295.

Ein QuadWord (Vierfachwort) besteht aus 2 DoubleWords (4 Words bzw. 8 Bytes), damit aus 64 Bits und hat den Wertebereich 0 bis 18.446.744.073.709.551.615. (Maximale Zahl etwa 18 ½ Trillionen)

Das QuadWord ist der Standard für die heute erzeugten und verkauften Mikroprozessoren - das sind diejenigen Einheiten, die in den Computern die eigentliche Denkarbeit leisten.

Bei derart riesigen Maschinenworten haben wir die Möglichkeit, neben den natürlichen und den ganzen Zahlen auch rationale und irrationale Zahlen in einer der Größe des Maschinenworts entsprechenden Genauigkeit darzustellen. Diese Darstellung erfolgt nach den Regeln der IEEE-(sprich: Ai-Tripl-I) -754-Norm. Diese reserviert in den Bits die folgenden Komponenten der binären Gleitkommazahl:

Ein Drittel der Bits für den vorzeichenbehafteten Binärexponenten.

Ein Bit für das Vorzeichen der kompletten Zahl.

Den Rest – also etwas weniger als zwei Drittel der Bits – für die Mantisse, also für die Stellen nach dem (binären) Komma. Um den zur Verfügung stehenden Platz optimal zu nutzen, wird die Zahl immer so normiert, dass vor dem binären Gleitkomma eine Eins steht, die im dargestellten Bitmuster gar nicht mehr angeführt werden muss, und in den Bits befinden sich dann nur mehr die Stellen nach dem Gleitkomma.

Trotz bestmöglicher Platznutzung gibt es nun dennoch Bitkombinationen, die keine derartigen binären Gleitkommazahlen ergeben. Diese werden für folgende spezielle Werte verwendet:

Haben alle Bits den Wert FALSE bzw. Null, so ist dies die tatsächliche Zahl Null (0,0). Diese kann entstehen durch direktes Setzen bzw. Speichern, durch Division einer sehr kleinen Zahl durch eine sehr große Zahl, wenn die resultierende Zahl nicht mehr darstellbar ist, etc. etc.

Es gibt ein reserviertes Bitmuster für die positive Unendlichkeit (positive infinity). Dieser Wert wird z.B. von der Maschine gesetzt, wenn eine positive Zahl durch Null dividiert wird.

Ganz analog dazu gibt es die negative Unendlichkeit (negative infinity). Dieser Wert wird z.B. erreicht, wenn eine sehr große positive Zahl durch eine sehr kleine negative Zahl dividiert wird, wenn das Resultat nach den oben genannten Regeln nicht mehr darstellbar ist.

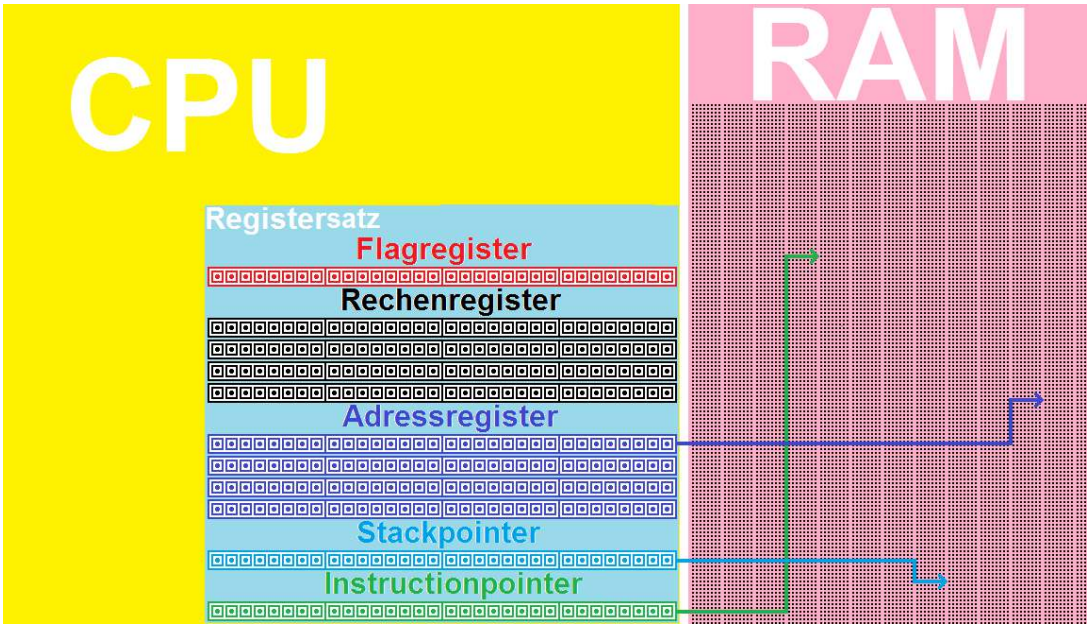
Zuletzt gibt es bei gewissen Rechenoperationen Ergebnisse, die gar keine Zahl mehr sind (not a number). Dazu das Beispiel der Division von Null durch Null: hier gilt die Gleichung Ergebnis = 0 / 0; multiplizieren wir diese Gleichung auf beiden Seiten mit Null, ergibt sich Ergebnis * 0 = 0, was ganz offensichtlich für absolut jedwedes Ergebnis gilt. Daher ist das Ergebnis die Menge aller bildbaren Mengen (und enthält damit auch das ganze Universum) und ist gar keine Zahl mehr.

2. CPU, RAM, Code und Daten

Unter der zentralen Prozessiereinheit (Central Processing Unit = CPU) versteht man üblicherweise keineswegs etwas blutdruckerhöhendes Juristisches, sondern denjenigen Bauteil (Chip), der im Computer die eigentliche Denkarbeit verrichtet.

Diese CPU ist über Adress- und Datenleitungen mit dem Hauptspeicher (Random Access Memory = RAM = zufällig zugreifbares Gedächtnis) verbunden. Jede dieser Leitungen repräsentiert genau ein Bit, und die CPU legt an die Adressleitungen ein Bitmuster (die Adresse) an, um dann an den Datenleitungen den Inhalt des Speichers an dieser Adresse zu erfahren.

Die Organisation dieses Speichers ist denkbar einfach: auf Adresse 0 liegt das nullte Byte, auf Adresse 1 das erste Byte, auf Adresse 2 das zweite Byte, und bei einem Speichervolumen von einem Terabyte auf Adresse 1.099.511.627.775 ganz einfach das 1.099.511.627.775te Byte. Freilich kann von so einer Byteadresse nicht nur ein Byte, sondern auch ein Wort, Doppelwort, etc. gelesen werden – dies betrifft dann halt auch die dem adressierten Byte folgenden Bytes.



In der CPU befinden sich zwei ganz stark zu unterscheidende Arten von „Registern“:

Diejenigen Register, mit denen gerechnet wurde und/oder wird und jene Register, mit denen im Hauptspeicher (RAM) adressiert wird. So kann z.B. zu einem Rechenregister der Inhalt addiert werden, auf den ein Adressregister (im RAM) zeigt. Das „Flagregister“ (Flaggenregister) beinhaltet einzelne Bits, von denen jedes eine besondere Bedeutung hat – bei dem genannten Befehl wird in einem „Flag“ angezeigt, ob das Ergebnis Null ist („Zero-Flag“), in einem anderen „Flag“ wird angezeigt, ob das Ergebnis einen Übertrag verursacht hat („Carry-Flag“).

Aber ein adressierendes „Zeigerregister“ kann noch auf ganz andere Dinge zeigen, als die Werte, die zu addieren, subtrahieren, multiplizieren, logarithmieren oder wer weiß noch wofür alles sonst zu verwenden sind.

Woher soll denn die CPU gerade wissen, was sie jetzt tun soll?

Der obengenannte Befehl würde in etwa lauten: Addiere zum Rechenregister 2 den Inhalt dessen, wo Adressregister 3 gerade hinzeigt. Jetzt muss die CPU den nächsten Befehl lesen. Zu diesem Zweck hat sie den „Instructionpointer“ (Instruktionszeiger), der auf den gerade auszuführenden Befehl zeigt, und der natürlich im Adressraum um die Größe der Instruktion weitergesetzt wird, wenn der zuletzt ausgeführte Befehl zu Ende gebracht wurde. In der nun gelesenen Instruktion kann zum Beispiel stehen, dass der Instructionpointer selbst um einen Betrag zu versetzen ist, wenn jetzt das Zero-Flag des Flagregisters gesetzt ist. Auf diese Art kann der Maschinencode bedingte Verzweigungen beinhalten.

Wir haben also nicht nur zu unterscheiden zwischen Datenregistern, die irgendeinen aktuellen Zustand einer Berechnung beinhalten, und Adressregistern, die irgendeine Byteadresse in sich tragen, die irgendwo im RAM liegt – nein! Wir müssen bei den adressierenden Registern auch unterscheiden, ob dort, wo sie hinzeigen, irgendwelche Instruktionen liegen, die die CPU abzuarbeiten hat, und den Registern, die dort hinzeigen, wo die Daten liegen, die gerade zu verarbeiten sind. So simpel ist der gravierende Unterschied zwischen Code und Daten zu verstehen.

Code ist das, was die CPU in ihrem Denkprozess auszuführen hat, und Daten sind die Inhalte der Speicherplätze, woher die Berechnungsdaten genommen werden, und die Zwischen- oder Endergebnisse der Berechnung abgelegt werden. Als Software bezeichnet man diesen Code, der abgearbeitet wird und als Hardware alle jene Bestandteile des Computers, die als real existierende physikalische Objekte vorliegen.

Jetzt fehlt uns zum totalen Verständnis des Denkvermögens einer CPU nur mehr dieser „Stackpointer“ (Stapelzeiger). Stellen Sie sich vor – ich muss über etwas nachdenken. Bei diesem Nachdenken muss ich zuvor ein Detailproblem lösen, dessen Ergebnis ich allerdringendst benötige zur Lösung des Gesamtproblems. Am Anfang habe ich ein Gesamtproblem und ich vertiefe mich in ein Detail, dieses Detail hat ein Unterproblem, das ich zuerst lösen muss, weil ich dessen Lösung für die Gesamtlösung meines Problems benötige. Und wenn ich dieses Detail gelöst habe, will ich zu meinem Hauptproblem zurückkehren. Und dieses Detail hat ein Unter-Unter-Problem, das ich lösen muss zur Bewältigung meines Unter-Problems. Und so geht es immer weiter vom Hauptproblem zum Unter-Problem, von dort zum Unter-Unter-Problem und immer weiter zum Unter-Unter-Unter-Unter-Unter-[...] Problem. Der Stackpointer zeigt immer auf den aktuellen Kontext – also auf die Zwischenergebnisse des aktuell zu bearbeitenden Objekts.

Mit dem Stackpointer finde ich das, was gerade aktuell ist.

Auf diese Architektur müssen alle Programmiersprachen abbilden aus dem einfachen Grund, weil wir keine bessere Architektur haben. Natürlich gibt es andere Architekturen als die von-Neumann-Maschine, aber diese haben alle die gemeinsame Eigenschaft reduzierter Gesamtfunktionalität bei aber u.U. schnellerer und/oder besserer Lösungsmöglichkeit für bestimmte Aufgaben.

Moderne CPUs haben u.U. mehrere Cores, mit denen sie gleichzeitig logische und arithmetische Aufgaben lösen können, aber so eine CPU kann natürlich immer nur für ein Core ins RAM greifen.

Religionsbetonte Menschen sind schockiert ob der Tatsache, dass es nun neben den gottkreierten Kreaturen sogar schon Maschinen gibt, die denken können, aber die Wesen, die in Wahrheit der Evolution entstammen, weisen gravierende Unterschiede zu unseren Denkmaschinen auf:

Die Nervenzellen unserer Gehirne arbeiten nicht digital (mit zwar sehr vielen, aber doch nur Bits), sondern analog – das (Zwischen-)Ergebnis einer Operation im Gehirn hat nicht nur einzelne gesetzte oder nicht gesetzte Bits, sondern eine Intensität, die die Werte gar nicht, ein wenig, sehr viel und enorm stufenlos darstellen kann.

Zum Zweiten sind die Nervenzellen in ein Lebewesen eingebettet, das sich in Interaktion mit seiner Umgebung befindet und so kompliziert innerhalb ihrer Peripherie angeordnet, dass es ab einer gewissen Komplexität und Größe des Gehirns zum Auftreten eines Bewusstseins und zur Empfindung einer Identität von Körper, Geist (Denken) und Seele (Fühlen) kommt.

In einem Belang sind sich jedoch diese beiden Daseinsformen sehr ähnlich: Im Gehirn des Subjekts Mensch wird sich nur das abspielen, wofür Eltern, Lehrer, Freunde und andere mehr oder weniger wohlgesonnene Zeitgenossen die Voraussetzungen geschaffen haben (so war es auch bei den größten Genies der Menschheitsgeschichte) – ebenso, wie das Objekt Denkmaschine auch beim Bearbeiten eines noch so ausgeklügelten Programms nur Lösungen finden wird innerhalb des Rahmens, den ein ganzes Heer von den Systemverantwortlichen bis zu den Programmierern sich ausgeheckt haben.

Irgendwelche mehr oder minder schwachsinnige Science-Fiction-Filme wollen uns weismachen, dass Computer ein Gefühlsleben hätten. Wir können derzeit nur feststellen, dass diese sehr wohl so tun können, als hätten sie eines (wenn die Programmierer dies so beabsichtigten) – in Wahrheit haben sie es zum jetzigen Zeitpunkt (noch) nicht. Ob es in Zukunft Roboter geben wird, die tatsächlich Empfindungen haben, wird nicht unwesentlich davon abhängen, ob die Menschheit unserer Tage so grenzenlos dumm sein wird, etwas von unserem Arsenal von dreissigtausend Atomwaffen zu aktivieren.

Wenn man am Computer den Strom abschaltet, sind CPU und RAM wieder leer. Daher gibt es unzählige Komponenten, die entweder im Computer selbst liegen oder außerhalb – sie dienen dem Transport von Code und/oder Daten in den Rechner hinein und/oder von diesem hinaus, um sie zu speichern. Dazu zählen etwa Bildschirme (hinaus), Tastatur und Maus, Scanner (hinein), Netzwerkskomponenten, Flash-Speicher, Festplatten (hinein und hinaus) und vieles mehr. Keines dieser Geräte bewirkt in der Denkmaschine irgendwelche Emotionen irgendeiner Art, während visuelle, taktile oder andere Sinneswahrnehmungen in der menschlichen Seele (das sind die im Gehirn ablaufenden Gefühle und nicht irgendwelche abstrusen Verbindungen zu einem virtuellen Weltschöpfer) enorme wohltuende oder abscheuliche Auswirkungen haben können. (Wie zum Beispiel Sex und Liebe in der einen und Glaubenskriege, Kindesmissbrauch und Genitalverstümmelung in der andern Richtung.)

Wir subsummieren die Grundaussage des Kapitels zwei: Computer sind dazu geschaffen, uns Denkarbeit abzunehmen in Analogie zum Geschirrspüler, der dazu geschaffen ist, uns die Abarbeitung der haushaltlichen Tätigkeiten zu erleichtern. Denkmaschinen sind zum Denken geschaffen und können dies auch – oft viel schneller und exakter als wir Menschen – aber sie können nicht fühlen, weil ihnen die Verbindung und Identität mit dem Körper fehlt. Sie denken, ohne sich dessen bewußt zu sein, wie Fliegen, deren Komplexität ihres Nervensystems (höchstwahrscheinlich) auch nicht zur Entwicklung eines Bewußtseins ausreicht.

Zum Schluss dieses Kapitels möchte ich noch den Cache-Speicher erwähnen, der sich auf modernen CPU's findet. Dieser ist ein CPU-internes Gedächtnis, das sich häufig adressierte RAM-Inhalte merkt, damit die CPU nicht immer wieder von den gleichen Adressen lesen muss, wenn sie z.B. gerade die gleiche Schleife millionenfach durchläuft und bewirkt dadurch im geeigneten Fall sehr sehr beträchtliche Zeiteinsparungen.

3. Vom Hexcode-Hacken zu C++

Die erste Rechenmaschine war wohl der Abakus. Beträchtlich später hatte man schon Denkmaschinen, deren Architektur der oben besprochenen ähnelte. Irgendwie muss man so einer Maschine jedoch die Software beibringen. So sagte man damals z.B.:

```
3F 01 01
56
7F 01 01
```

Dies bedeutete: Lade den Inhalt von Adresse 257 ins primäre Rechenregister.
Erhöhe das primäre Rechenregister um eins.
Schreibe das Ergebnis auf dieselbe Adresse 257 zurück.

Zum Einen ist hier anzumerken, dass die heute üblichen Maschinen diesen ganzen Vorgang in einer einzigen Instruktion erledigen, andererseits hat der Mangel an Komfort beim „Hexcode-Hacken" alsbald dazu geführt, dass die ersten Softwaremaschinen zum Softwareentwickeln geboren wurden und das waren die Assembler, die es dem Programmierer ermöglichten, vereinfacht zu schreiben:

```
DATA_SECTION
WORD meine_variable
CODE_SECTION
MOV AX, [meine_variable]
INC AX
MOV [meine_variable], AX
```

oder noch viel einfacher auf den heutigen Prozessoren:

```
INC [meine_variable]
```

(INC steht für „Inkrementieren": Erhöhen um eins)

Heute wird derartiger Code nur mehr von zwei Arten von Leuten „geschrieben": Zum Einen von den Boot-Sector-Autoren, die es ermöglichen, dass das Betriebssystem (dazu später) überhaupt mit dem Laden der eigenen Software beginnen kann und andererseits die Autoren der Codegeneratoren von Compilern. (Ein Compiler ist dasjenige Softwarewerkzeug, das aus den Programmzeilen, die in einer bestimmten Programmiersprache geschrieben wurden, den Maschinencode erzeugt und der Codegenerator ist diejenige Komponente des Compilers, die aus den Zwischenergebnissen der anderen Teile des Compilers diese wirklich auf die jeweilige Maschine abbildet.)

Nun war diese Sachlage den Programmierern wirklich zu ungemütlich – die Architektur und Instruktionen der Zielmaschine kennen zu müssen, wo man sich doch eigentlich um den Algorithmus (wie behandle ich welche Problemstellung) kümmern wollte, und endlich wurden die Programmiersprachen geboren.

Nun schreibt man ganz einfach:

```
X = X + 1
```

Diese mathematische Formel ist natürlich falsch! Der Mathematiker würde auch schreiben:

$X_{n+1} = X_n + 1$

Aber in den Programmiersprachen ist es üblich, ganz einfach auf die linke Seite des Gleichheitszeichens den Variablennamen zu schreiben, auf die rechte Seite die Formel mit der Rechenvorschrift für das abzuspeichernde Ergebnis.

In den intelligentesten aller Sprachen, den C-Sprachen, schreibt man ganz einfach:

```
++X;
```

Mit derlei Mitteln ausgestattet konnten natürlich immer komplexere Algorithmen bearbeitet werden, und in der Berechnung eines Autogetriebes auf Festigkeit befanden sich wahre Unmengen von Instruktionen wie:

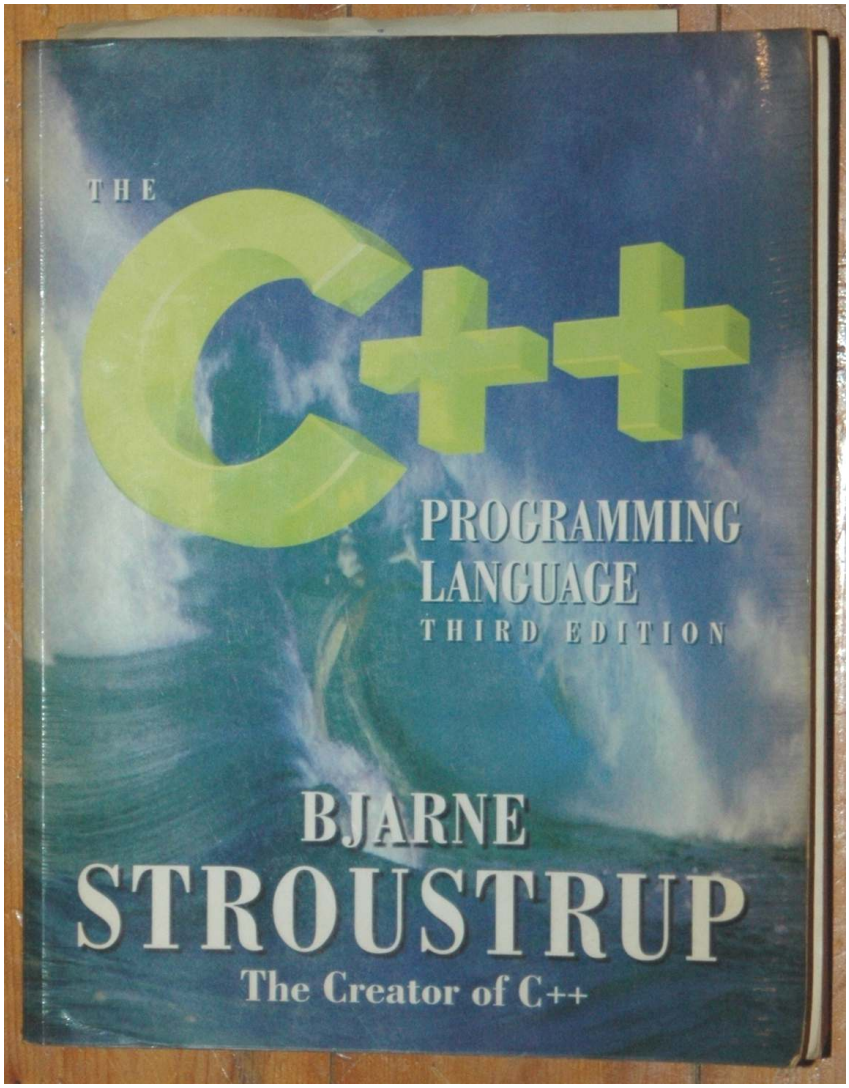
```
IF X > 123456 GOTO 4321
```

(wenn, dann gehe an die Stelle, wo steht `4321: CONTINUE`)

Auch das wurde den Programmierern schön langsam zu ungemütlich. Man nannte das „Spaghetticode“, weil sich niemand mehr dabei auskennen konnte bei den Myriaden von GOTOs – vor allem bei den komplexen Angelegenheiten – und die „strukturierten Programmiersprachen“, die als zweite Generation bezeichnet wurden, hielten Einzug. Also:

```
FOR (alle_anstehenden_Nachrichten())
{
IF (Fehler_in_Nachricht()) { bearbeite_Fehler(); CONTINUE; }
einsortieren_Nachricht();
}
```

Jetzt konnte man schon sehr komplexe Vorgangsweisen (Algorithmen) in Software realisieren. Aber alle Entwickler hatten dieselben Codestücke und Variablen. Das war die sogenannte Softwarekrise, als man feststellte, dass bei Projekten, die eine gewisse Komplexität überschreiten, niemand mehr den Überblick bewahren kann, wenn alle Beteiligten mit dem Allwissen ausgestatteten im realen Projekt über Alles Bescheid wissen müssen. Hier offenbart sich ein ganz wesentlicher offensichtlicher Unterschied in den Tätigkeiten: Ein Theologe hat möglichst beeindruckenden Fasel (z.B. über „interreligiösen Dialog“) von sich zu geben, dabei vielleicht mit Obrigkeiten zu kämpfen, die ihm widersprechen – der Softwareentwickler hat real existierende Problemstellungen möglichst wahrheitsgemäß zu analysieren und dabei Lösungsmodelle zu entwickeln, die miteinander verglichen werden können, und Umsetzungsmöglichkeiten parat zu halten, die mit geringstem Aufwand das bestmögliche Ergebnis zu erzielen imstande sind.



Es stellte sich heraus, dass bei umfangreichen Projekten auch der klügste und geeignetste Systemverantwortliche nicht mehr detailliert Bescheid wissen kann über alle Code- und Datenstücke und es hat die Geburtsstunde der dritten Generation – der objektorientierten Sprachen geschlagen. Die wesentlichen Eigenschaften dieser Objekte dieser Sprachen sind, dass sie unterschieden werden, ob sie privat (`private:`) sind und nur für den zuständigen Detailprogrammierer von Belang, oder öffentlich (`public:`) und für jeden Projektbeteiligten zugänglich. Das wesentlichste Genie in dieser Entwicklung war Bjarne Stroustrup, der die unvergleichlich potente Sprache C++ kreierte, in der heute nahezu alle Betriebssysteme, Compiler und umfangreichen und laufzeitabhängigen Anwendungen geschrieben sind.

Denken wir zum Beispiel an die monatlich oder gar in Echtzeit erforderliche (d.h.: alle Telefonnummern müssen immer im aktuellen Zustand verfügbar sein) Generierung eines Telefonbuchs. Hier braucht der Programmierer, der einen Teil benutzt, den ein anderer oder er selbst vor langer Zeit implementiert hat, nur mehr zu schreiben:

```
telefonbuch.insert (name, vorname, postleitzahl, ...);
```

Und er braucht sich keinerlei Gedanken darüber zu machen, wie diese Daten jetzt ins Telefonbuch eingebracht und dort intern dargestellt werden.

Entwicklungssysteme sind selbst wieder Software und stellen die Möglichkeiten zur Verfügung, die Programmtexte (sources) zu editieren und übersetzen (compilieren) und dann zum eigentlichen Programm zu verbinden (linken). Das wohl modernste der heute verfügbaren Entwicklungssysteme ist Qt, in dem es sogar möglich ist, Programme zu erstellen, ohne vorher festzulegen, auf welchen Maschinen und unter welchen Betriebssystemen das Programm eingesetzt werden soll. Haben wir mit:

```
bool operator < (const entry&telefonbucheintrag);
```

einmal festgelegt, wann ein Eintrag vor einem anderen zu liegen hat, brauche ich hier nurmehr zu sagen:

```
qSort(telefonbuch);
```

und mit diesem Aufruf einer (vom Entwicklungssystem zur Verfügung gestellten) Standardmethode liegt unser Telefonbuch bereits im alphabetisierten Zustand vor!

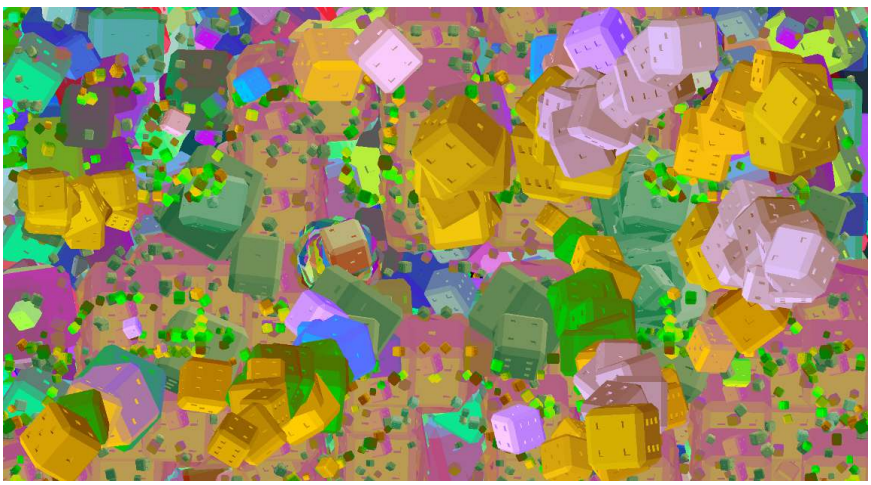
Welche Eigenschaften die Sprachen der vierten Generation ausmachen werden, wage ich nicht vorauszusagen – wo doch Prophezeiungen immer schwierig sind, vor allem, wenn es sich um die Zukunft handelt.

4. Betriebssystem, Programm, Prozess und Thread

Auch das beste Programm tut noch gar nichts, wenn es nicht ein Betriebssystem zur Verfügung hat. Dabei ist das Betriebssystem (operating system, OS) selbst nichts anderes als ein Programm, das automatisch geladen wird beim Hochfahren der Denkmaschine. Die Hauptaufgaben des Betriebssystems sind das geordnete Präsentieren der verfügbaren Programme für den Benutzer (user), das Ermöglichen der Interaktion zwischen Prozess(en) und User(n) und das zur Verfügung Stellen der geeigneten Schnittstellen zur Peripherie (Hardware) des Computers (Tastatur, Maus, Bildschirm, Festplatte, etc.).

Auch die Betriebssysteme haben (wie die Programmiersprachen) eine Historie durchlaufen, wo am Anfang zum Schockerlebnis der User sogar Computer ohne System verkauft wurden, aber im wesentlichen strickte jeder Computerhersteller sein eigenes System, bis ganz kluge Leute auf die Idee kamen, ein System zu designen, das dem User gegenüber immer das gleiche Auftreten hat und diese enorm schwierige Aufgabe erledigen konnte, jede beliebige Anzahl von Prozessen unterschiedlicher Priorität zugleich laufen zu lassen.

Wenn ein User vermittelst des Betriebssystems ein Programm startete, dann hat das System damals die Daten des Programms vom Magnetband gelesen – heute liest es zumeist von der Festplatte, legt diese Daten zum Teil als exekutierbaren Code in den Hauptspeicher, trägt dort noch irgendwelche Adressen ein, und startet das Programm, das dadurch zum Prozess wird. Die Prozesse haben unterschiedliche Wichtigkeiten (Dringlichkeiten, Prioritäten) und darum muss das System dazu in der Lage sein, unterschiedlichen Prozessen unterschiedliche Laufzeiten zur Verfügung zu stellen.



Das System gibt der CPU die Anweisung, zu regelmäßigen Zeiten den gerade laufenden Prozess zu unterbrechen und ins System zu springen, damit das System darüber entscheiden kann, welcher Prozess mit welcher Priorität jetzt an die Reihe kommen soll. Die Komponente des Systems, die dieses erledigt, nennt man Scheduler (sprich: skédjula) und die Eigenschaft, dass auch niedrigpriore Prozesse drankommen müssen, nennt man Realtime Multitasking (Echtzeitfähigkeit).

In den Siebzigerjahren des letzten Jahrhunderts des letzten Jahrtausends stellte sich heraus, dass der Scheduler des Betriebssystems UNIX der beste aller Scheduler war und (nahezu) alle Systemprogrammierer begannen unaufhaltsam, UNIX nachzubauen. Dass das bei WINDOWS nicht gemacht wurde ist die Begründung, warum besonders kritische Programmierer dem „Betriebsystem“ WINDOWS diesen Titel überhaupt aberkennen wollen. Aber das mag wohl daran liegen, dass Bill Gates dem wirtschaftlichen Erfolg seiner Firma höhere Bewertung zumaß, als den Qualitäten seiner Produkte. Die Firma Digital Research (in Konkurrenz mit dem Millionstelweich-(Microsoft-) Guru) hatte bereits das hervorragende realtime multitasking OS FlexOS – zu Zeiten, wo Bill Gates noch an seinem MSDOS herummurkste.

Heute sind die wertvollen Qualitäten ausgezeichneter Programmierer oftmals dazu verurteilt, gegen die Eigenschaften von WINDOOF zu kämpfen, anstatt sich ihren außerordentlich schwierigen algorithmischen Aufgaben zu widmen.

Man unterscheidet zwischen synchron (geradlinig von vorne nach hinten) und asynchron (gleichzeitig). Die Prozesse innerhalb des Betriebssystems laufen asynchron – aber die Sache wird zum Leidwesen der Programmierer noch komplizierter:

Nicht nur das System hat Aufgaben, die es „gleichzeitig“ (es gibt natürlich wahre Gleichzeitigkeit nur auf multicore CPUs) erledigen muss – auch die Applikationen (Prozesse) müssen oftmals Aufgaben asynchron bewältigen. Auch das hat das arme WINDOWS lernen müssen, und die Threads (Fäden) eines Prozesses bearbeiten quasigleichzeitig gemeinsame Aufgaben im gleichen Kontext des Prozesses.

Stellen Sie sich bitte vor – innerhalb eines Prozesses wartet die Komponente A darauf, dass die Komponente B fertig wird – und die Komponente B wartet darauf, dass die Komponente A fertig wird – dies ist ein Deadlock – und nicht so schlimm, wenn das innerhalb eines Userprozesses stattfindet – dann brauch ich ja nur den Prozess abzuschießen – schlimmer jedoch, wenn es sich um Systemkomponenten handelt. Daraus lässt sich schliessen, dass der Betriebssystembau wohl nicht die einfachste Sache der Welt ist.

Ich habe jetzt eine Bitte an Sie: Falls Sie beabsichtigen, irgendetwas zu steuern (einstellen) und/oder zu regeln (in Abhängigkeit vom aktuellen Zustand zu ändern), wovon Menschenleben abhängen – wie häufig in der Technik, es seien Verkehrsmittel, Staudämme, Atomkraftwerke und Atomwaffen genannt – bitte treffen Sie auch die Entscheidung über das Betriebssystem Ihrer Denkmaschinen gewissenhaft.

Die Programme bzw. Prozesse nennt man Applikationen (Anwendungen) und zwei Arten davon möchte ich noch kurz skizzieren, weil sie gar so deutlich veranschaulichen, dass von den heutigen Denkmaschinen Aufgaben bewältigt werden können, die man mit ganzen Heeren von denkenden Menschen beim besten Willen nicht in nichtastronomischen Zeitspannen erledigen könnte:

Zum einen die Finite Elemente Methode (FEM), bei der man Problemstellungen bearbeitet, bei denen die physikalischen Bedingungen als mathematische Gleichungen nicht lösbar sind oder erst gar nicht angeschrieben werden können. Hier erreicht man Berechnungsergebnisse, die über viele Dezimalen zuverlässig sind, indem man das physikalisch-geometrische Modell in Myriaden von winzigen Teilchen zerlegt, für diese die Gleichungen anschreibt, und von der Denkmaschine dann das entstehende schier unendlich riesig scheinende Gleichungssystem lösen lässt. Alle die erforderlichen Einzelschritte hierzu wären ohne Computer undenkbar.

Zum anderen die evolutionären Algorithmen, die auf beeindruckende Weise veranschaulichen, wie sinn- und wirkungsvoll man aus der Natur lernen kann: Hier werden nach ausgeklügelten Schemata kleine Mutationen (Modifikationen) etwa an einer Flugzeugtragfläche angebracht, die physikalischen Eigenschaften (Festigkeit und Strömung) des neuen Objekts werden nun mittels FEM ermittelt und wie in der evolutionären Selektion werden dann Verschlechterungen aussterben (gelassen) und Verbesserungen weiterverfolgt.

5. Conclusio

Die Denkmaschinen können uns eine Unmenge von Arbeit abnehmen, wenn sie richtig eingesetzt werden. Die Berechnung einer Tragfläche eines Verkehrsflugzeuges kann auch unter größtem Aufwand mit heute „zu Fuß“ rechnenden Personen nicht bewältigt werden.

Denkmaschinen haben keine Wochenenden, keine Feiertage, keine Arbeitsgesetze. Wollen wir die Klimakatastrophe eindämmen, müssen wir unsere Energiebilanzen analysieren und auch in der Nacht und in unserer Freizeit relevante Messdaten erfassen lassen, die uns dann in unserer Arbeitszeit in übersichtlicher Form veranschaulicht werden. Für solche Aufgaben sind die modernen Denkmaschinen vorzüglich geeignet.

Die Denkmaschinen können auch (z.B. mit Logikspielen) Spaß bereiten oder aber das menschliche Gehirn umprogrammieren: Das ständige Blutspritzen, die möglichst realitätsgemäße Darstellung des Zerreißens von menschlichen Organen trainiert unsere Denkmaschine auf den Einsatz im Religions- und Kulturkrieg. All diese Menschen (Männer, denn weibliche Wesen sind nicht so aufopferungswillig, dass sie sich solchen nebeligen und umnebelten Ideologien unterordnen würden, um ihr eigenes Leben aufs Spiel zu setzen), die solche Dinge in die Realität umsetzen, sind genitalverstümmelt (wie es die abrahamitische Religion fordert) und eines Tages töten sie wirklich einen anderen Menschen. Das ist genau die Programmierung, die das Christentum als Sterben für Gott, Kaiser und Vaterland über anderthalb Jahrtausende in unseren Gehirnen beschworen hat und die Attentäter von Paris haben

lediglich die Anweisungen des Koran in die Tat umgesetzt. Die Leute, die abstreiten, dass das etwas mit Religion zu tun habe, kennen ihre eigenen Programmschriften (Bibel, Koran) nicht.

Wir sind genauso programmierbar wie unsere Denkmaschinen. Wir müssen endlich damit aufhören, unseren Gehirnprogrammierern zu vertrauen. Diese haben unser Vertrauen über unzählige Generationen hinweg mißbraucht. Beginnen wir eigenständig zu denken und handeln nach dem, was unser Gewissen uns vorschreibt. (Kategorischer Imperativ!)

Weg mit den Theolügen von Kindergärten, Schulen und Universitäten!

Der Einsatz von Robotern und Denkmaschinen hat die Produktivleistung in unbeschreiblicher Form ansteigen lassen. Diese Sachlage hat jedoch die Profite der Kapitalisten und Spekulanten ins Unermessliche gesteigert und gleichzeitig Heere von Arbeitslosen und unter der Armutsgrenze Lebenden geschaffen, weil der Nutzen dieser Maschinen missbraucht wurde und wird. Wir brauchen pflichtbewußte Verantwortungsträger, die die von Computern bewirkten wirtschaftlichen Veränderungen zum Nutzen der benachteiligten Bevölkerung arbeiten lassen.

Wissenspfründe wie Wikipedia und WikiLeaks müssen gefördert werden – Religions- und Kulturkriegmanipulation gehört raus aus den Medien, aus dem Internet und aus den Computern.

Unsere Denkmaschinen müssen der Bildung und dem Lebensstandard aller dieser Bevölkerung unseres Erdballs zur Verfügung stehen und dienlich sein.

Die Bilder:
Seite 1: 15011502_machineWords.png ... Grafik by Richard Kofler
Seite 2: 15020404_cpu_ram.png ... Grafik by Richard Kofler
Seite 5: 15020700_c++-title.JPG … Foto by Richard Kofler
Seite 6: 15020500_application.png ... Screenshot vom Programm rižArtë;
beides by Richard Kofler