# MTX User-Club Deutschland

**Info 45**
17.01.1992

**Zweck:** Zusammentragen und Austausch von Tips & Tricks u.s.w., Hilfestellung bei allen möglichen Problemen, Aufbau einer Programmbibliothek und Basteln von Hardware-Erweiterungen.

**Programme** (nur **Selbstgeschriebene**): Tausch von kurzen und einfachen Routinen. Gute Programme (mit Dokumentation) können über den Club an Mitglieder verkauft oder auf Public-Domain werden. Wer solche Programme an uns schickt erhält ggf. Verbesserungshinweise und eine Besprechung im Info.

**Mitglied** kann jede(r) werden! Keine Beitragsgebühr! Anmeldung kostet DM 2.-.

**Verpflichtungen:** Einsendung unseres Anmeldeformulars.

**Bitte:** Einsendung von Tips & Tricks, Fragen, Antworten, Routinen, Programmen, Beiträgen und Anregungen zum Info, Hinweisen auf preiswerte Hard- und Software, und was so zusammenkommt und andere interessieren könnte.

**Club-Info**, unser Blatt, verschicken wir ca. 8-wöchentlich. Inhalt ist alles was uns über den MTX/FDX (ohne Copyright) in die Hände fällt. Es kostet nicht über DM 12.- je Exemplar. Jeder kann dazu Beiträge liefern und gratis Kleinanzeigen veröffentlichen.

**Kosten:** Wir berechnen ausschließlich Selbstkosten und verschicken **nichts**, wenn Ihr persönliches Guthaben nicht reicht! (s.u.)
Schüler, Studenten, Auszubildende, Grundwehrdienstleistende, Rentner und Arbeitslose erhalten einen Nachlaß von 40% auf die zukünftigen Infos nach Einsendung einer entsprechenden Bescheinigung für deren Gültigkeitzeitraum.

**Geld/Konto:** Für jedes Mitglied führt Herbert zur Nedden ein Konto, von dem die jeweils enstehenden Kosten abgehen. Der Kontostand wird bei **jeder** Sendung mitgeteilt (**er steht über der Anschrift**) und kann selbstverständlich jederzeit erfragt werden!

Einzahlungen bitte auf's Club-Konto: (oder V-Scheck)
(**Absender!** incl Name **und** Anschrift bitte nicht vergessen!)
**Postgiroamt Hamburg, BLZ 200 100 20,**
**Herbert zur Nedden, Sonderkonto C, Nr. 3480 00-200**

**Kontaktadressen:**


Herbert zur Nedden
Alte Landstraße 21
2071 Siek
(04107) 99 00

Hans Gras
Statenhoek 49
NL 1506 VM Zaandam
(0031-75) 17 49 91


**Telefon-Sprechzeiten**
Herbert zur Nedden: Do 18 - 21 Uhr, Sa 9 - 12 Uhr
(Etwas klingeln lassen oder nochmal versuchen!)

MTX-Info – Copyright (C) 1983-1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

I n h a l t s v e r z e i c h n i s



**Preis für dieses Info: DM** 11.70

**Meine Sprechstunden** (Herbert zur Nedden, 2071)
Samstags nur noch bis 12 Uhr.

**Clubtreffen** (Herbert zur Nedden, 2071)
Hast Du Interesse? Ort: südlich von Hannover, Termin: April-Juni. Wenn ja, laß es mich bitte wissen. Bei zu wenigen positiven Rückmeldungen bis Mitte März findet dieses Jahr kein Treffen statt.

Kontostand (Herbert zur Nedden, 2071)
Eine rote Markierung auf dem Umschlag bedeutet, daß er zu niedrig ist.

MTX-Info – Copyright (C) 1983-1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

Moin, moin!

Ich hoffe, Ihr seid gut ins neue Jahr gekommen! Ich habe den Jahreswechsel im Rahmen des größten Teils meines Jahresurlaubs genossen.

Ja, es ist da: RAM 6.2! Es ist in seinem Inneren recht stark überarbeitet worden (sprich: hat mich in meinem Urlaub gut beschäftigt) und bietet daher eine Reihe von Neuerungen wie z.B. der Möglichkeit, die Systemgröße, gemessen am BDOS auf 63.75 kB zu bringen - und zwar bei allen Boot-EPROMs. Und noch einiges mehr, aber das kannst Du weiter unten lesen.

Thema Clubtreffen: Wer hat Interesse an einem Clubtreffen im April bis Juni 1992 - natürlich wieder in Ronneberg südlich von Hannover von Samstag ca. 10-11 Uhr bis Sonntags. Wenn Du Interesse hast, laß es mich bitte incl. der Angabe, welche Termine Dir (nicht) genehm wären wissen. Je nach der Menge der Antworten, die ich bis Mitte März 1992 erhalte, findet es statt oder nicht.

Eine erfreulich rasche und auch große Resonanz: Das Gesamtinhaltsverzeichnes der Infos 1 bis 46 wird im kommenden Info erscheinen.

Ich habe vor, mir nächsten Monat einen zweiten Rechner zuzulegen. Natürlich wieder einen englischen Exoten mit einem BASIC drin, welches Assembler-Routinen im Programm enthalten kann - das erinnert doch recht stark an den Memotech, nicht wahr? Der neue ist ein Archimedes A5000 von der Firma Acorn. Das Teil hat eine von Acorn entwickelte RISC-CPU, genannt ARM 3. Die CPU ist in der Tat faszinierend! Jeder Befehl kann in Abhängigkeit der Flags ausgeführt werden, die meisten können je nach Wunsch die Flags verändern oder auch nicht. Dieses Konzept macht die meisten relativen Sprünge überflüssig, so daß die Befehlspipeline nicht abreißt. In einem Artikel habe ich gelesen, daß ein 8 MHz-ARM in Sachen Leistung mit einem 25MHz-80386 vergleichbar ist - und der A5000 hat 25MHz CPU-Takt. Als Betriebssystem hat die Kiste Acorns RISC-OS Version 3.0: Ein Multitasking-System mit Window-Oberfläche, welches ähnlich wie Apple's Betriebssystem auf Moduln basiert. Stecken tut das Betriebssystem, das BASIC sowie einige Anwendungen in den 2 MB EPROM im A5000. Für einfache Anwendungen wird übrigens das BASIC gerne verwendet, da es keine Runtime-Library benötigt: alles, was es braucht steckt in den EPROMs und schnell ist es obendrein.

Die erste Anwendung, die ich zusammen mit Olaf Krumnow schreiben werde (er hat schon einen Archimedes) ist die, daß der MTX als eine Task unter RISC-OS läuft: Die 80Zeichen-Karte des MTX wird auf dem Archie als Fenster zu sehen sein und die Tastatureingaben in dem Fenster wandern an den MTX. Sprich er wird remote betrieben! In den nächsten Stufen werden die beiden Rechner dann auch Datenaustausch und anderes können.

Das bedeutet jedoch nicht, daß ich dem MTX User-Club Deutschland ade sagen werde - der wird auch weiterhin bestehen. Allerdings werde ich evtl. etwas weniger Zeit für den Memotech übrig haben. Ob sich das jedoch so bemerkbar machen wird, muß sich zeigen, da zur Zeit die Aktivitäten im MTX User-Club Deutschland eh recht zurückhaltend sind.

Euer

Herbert

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

C l u b: Fragen/Antworten

## Fragen und Antworten

F: (Christian Schinko, 8314)
Wer baut mir einen Omti-Controller an meine SDX?

F: (Peter Würfel, 7262)
Wer ist an einer HGR-80-Zeichen-Karte interessiert. Ich hätte eine zum Tausch gegen eine normal einfach aufgerüstete (96x26 Zeichen) anzubieten.

F: (Andreas Fischer, CH-4303)
Es gibt ein Buch von Klaus Hempe "Sterne im Computer". Darin sind sehr interessante Programme in Turbo-Pascal aufgelistet. Hat jemand die Programme auf Diskette. Andreas hätte sie sehr gerne auf 3 1/3"-Diskette im Format 07.
**Anm.d.HzN:** Ich kann das Konvertieren von 5 1/4"-Disk auf 3 1/2" gerne machen.

## Gelatcht oder ungelatcht (Herbert zur Nedden, 2071)

Kürzlich in der c't in zwei Leserbriefen entdeckte ich folgendes:

F: (Im Leserbrief Nr. 1)
Außerdem würde ich gern wissen, worin der Unterschied zwischen dme gelatchten und ungelatchten Betrieb eines Plattencontrollers besteht. Ich verwende einen WD 1006 SR2 an einer Segate 277R. Außer daß im gelatchten Betrieb die Kontrollampe der Platte ständig brennt, kann ich keinen Unterschied zwischen den Modi feststellen.
A: (von c't)
Zum latched mode: genau das (und nur das), was Sie beobachtet haben, nämlich eine ständig leuchtende Harddiks-LED, bewirkt diese recht überflüssige Option.

F: (Im Leserbrief Nr. 2)
Nachdem ich ein Verzeichnis auf meiner Festplatte mit DEL *.* geleert hatte, wurden nach einem DIR immer noch zwei Dateien mit der Bezeichnung '.' und '..' angezeigt. Als ich versuchte, mit DEL .. die eine davon zu löschen, bekam ich die Rückfrage, ob ich auch wirklich alle Dateien im Verzeichnis löschen wollte; genau das wollte ich natürlich. An der Anzeige mittels DIR hatte sich jedoch nichts geändert; als ich etwas später neu booten wollte, blieb der Rechner mit der Meldung 'Falscher oder fehlender COMMAND.COM' hängen. Was ist denn nun eigentlich passiert.
A: (von c't - gekürzt)
Die beiden Directory-Einträge '.' und '..' sind zwei Verzeichnisse. '.' ist das aktuelle und '..' das Verzeichnis eine Ebene höher. Daher löscht DEL . alle Dateien des aktuellen Verzeichnisses, ist also wirkungsgleich mit DEL *.*. DEL .. hingegen löscht die Dateien des nächst höheren Verzeichnisses - und so haben Sie etwas gelöscht, was sie garnicht löschen wollten.

**Anm.d.HzN:** Es ist doch immer wieder interessant zu sehen, mit welchen Problemen und Fallstricken sich MsDos-Anwender herumschlagen müssen.
Allzusehr sollte ich aber nicht unken, da mein neuer Rechner einen PC-Emulator hat - wer weiß, was mir da unterkommt, wenn ich mal das eine oder andere PC-Spiel ausprobieren will.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: KLIX-Programmierung

## KLIX-Programmierung mit DSEG

(Herbert zur Nedden, 2071)

Wie sieht ein KLIX-Overlay aus?

KLIX-Header
Programm mit seinen Daten
Evtl. Librarymodule
Pufferbereich

Der Pufferbereich ist kein Bestandteil des .KLX selbst sondern wird durch den Loader beim Laden des KLIX-Overlays direkt hinter dem Programm auf dem Heap reserviert. Das funktioniert so, daß der Loader weiß, wie lang die ersten drei o.g. Teile zusammen sind und einfach zusätzlich zu dem dafür erforderlichen Platz den geünschten Pufferplatz auf dem Heap mit reserviert.

Wozu? Gelegentlich schreibt man ja KLIX-Overlays, die einiges an Puffer für Ihre Daten benötigen. Als Beispiele wären da u.a. DiJey (Platz für das Directory und die Dateipuffer) und MTX-Edit (Platz für den Text). Dadurch, daß der Loader diesen Puffer auf Anforderung mit reserviert muß er nicht im Programm selbst definiert und damit nicht im .KLX enthalten sein. Das spart Platz auf der Disk.

Wieviel Puffer gewünscht wird und wieviel tatsächlich reserviert wurde, wird im KLIX-Header angegeben. Hier der KLIX-Header:

```
$MEMRY::        dw      0

ProgStart:      jp      Start
                db      'NameName'      ; Programmname
                dw      0AA1AH          ; Loader-Identifikation

BuffLen:        dw      PufferLen       ; I: Pufferanforderung
                                        ; O: Pufferlänge
ProgLen:        db      0ffh            ; I: Lösch 0ffh=Ja, 00h=Nein
                db      0               ; I: Funktionstaste
                                        ; O: ProgrammLänge ab ProgStart
                db      61h             ; Mindest-Version RAM
                dw      KlxInit         ; Programminitialisierung
                dw      KlxUnit         ; Programmuninitialisierung
                db      0               ; Bit 6 (40h) = Löschschutz
                                        ; Bit 5 (20h) = reserviert
                                        ; Bit 4 (10h) = Warmboot-Lösch
                ds      6,0             ; reserviert
```

In $MEMRY trägt der Linker (L80, SLRNK, ...) automatisch die Gesamtlänge des Codes, d.h. KLIX-Header, Programm mit Daten und der dazugelinkten Library-Module ein. In BuffLen muß die Länge des gewünschten Pufferbereichs stehen; nach dem Laden trägt der Loader die Länge des tatsächlich reservierten Pufferbereichs ein. Dies ist erforderlich, da Du als Länge auch 0FFFFh angeben kannst - dann wird ein möglichst großer Pufferbereich reserviert, dessen Länge das KLIX-Overlay so erfährt. Weiterhin stellt der Loader an ProgLen die Länge des Programmes ein; er überschreibt folglich die Informationen, die dort zuvor gestanden haben. Den Wert ProgLen benötigst Du, um ausrechnen zu können, wo der Puffer beginnt.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: KLIX-Programmierung

Will ich in einem KLIX-Overlay mit so einem Pufferbereich arbeiten, habe ihn also auch im KLIX-Header angefordert, muß ich mir dessen Adresse im laufenden KLIX-Overlay errechnen:

```
ld      hl,ProgStart    ; hl = Programmanfang
ld      de,(ProgLen)    ; de = Programmlänge
add     hl,de           ; hl = Pufferanfang
```

Will ich viele einzelne Datenfelder in dem Puffer bearbeiten, muß ich - äh nein, das KLIX-Overlay - entsprechend all deren Adressen errechnen . Das ist recht unpraktisch. Daher werden kleinere Datenbereiche in KLIX-Overlays meist im Programm definiert, da es dann leichter ist damit zu arbeiten; aber das kostet Platz auf der Disk, da im .KLX der gesamte Programmbereich enthalten ist.

Bei der Programmierung normaler .COMs mit Assembler gibt es einen schnuckeligen Weg, das .COM klein zu halten und trotzdem die Datenbereiche bequem zu bearbeiten:

```
CSEG
Hier kommt das Programm und die Datenbereiche hin,
die Konstanten beinhalten.
DSEG
Hier kommen die anderen Datenbereiche hin, also ins-
besondere größere Datenpuffer usw.
(Diese müssen alle mit DS ohne Defaultwert definiert werden.)
END
```

Beim Linken werden zwar alle Labels des DSEG korrekt errechnet, aber das DSEG selbst nicht in das .COM mit abgespeichert, wenn man dem Linker sagt, daß das DSEG hinter dem CSEG stehen soll (dabei muß das DSEG nicht unbedingt direkt hinter dem CSEG landen) und im DSEG die Datenbereiche mit DS ohne Defaultwert definiert wurden. So kann ich leicht mit verschiedenen Puffern (z.B. Lese- und Schreibpuffer, usw.) arbeiten, ohne deren Adresse umständlich errechnen oder mit EQUs definieren zu müssen.

Was macht der Linker mit obigem Beispiel-Programm, so man ihn dazu ermächtigt? Er legt die verschiedenen Bereiche in folgender Reihenfolge an:

1. CSEG-Bereich
2. Dazugelinkte Library-CSEG-Bereich(e)
3. DSEG-Bereich
4. Dazugelinkte Library-DSEG-Bereich(e)

Und nur die CSEG-Bereich(e), also 1. und 2. werden in das .COM gesteckt. Damit ist das .COM schön klein, die Datenbereiche jedoch leicht anzusprechen.

Und warum machen wir nicht das selbe mit KLIX-Overlays?

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: KLIX-Programmierung

Bei der Programmierung von Termin (meinem Terminkalender im KLICK) habe ich genau das getan! Die einzige Einschränkung ist, daß keine Library-DSEG-Bereiche vorkommen dürfen. Hier mein Programm samt KLIX-Header:

```
                CSEG

                dw      BegOfDSEG       ; Hier steht das Programmende

ProgStart:      jp      Start
                db      'Termin 1'      ; Programmname
                dw      0AA1AH          ; Loader-Identifikation
BuffLen:        dw      LenOfDSEG       ; I: Pufferanforderung
                                        ; O: Pufferlänge
ProgLen:        db      0ffh            ; I: Lösch 0ffh=Ja, 00h=Nein
                db      S_F7            ; I: Funktionstaste
                                        ; O: ProgrammLänge ab ProgStart
                db      61h             ; Mindest-Version RAM
                dw      KlxInit         ; Programminitialisierung
                dw      KlxUnit         ; Programmuninitialisierung
                db      0               ; Bit 6 (40h) = Löschschutz
                                        ; Bit 5 (20h) = reserviert
                                        ; Bit 4 (10h) = Warmboot-Lösch
                ds      6,0             ; reserviert

                Hier nun das Programm

                DSEG
BegOfDSEG:
                Hier die Datenbereiche
LenOfDSEG       equ     $-BegOfDSEG

                END
```

Wann, warum und wie(so) funktioiert das?

Der Linker erzeugt zusammen mit GETREL als .KLX folgendes

1. CSEG-Bereich
2. Dazugelinkte Library-CSEG-Bereich(e)
3. DSEG-Bereich

wobei die DSEG-Bereiche nicht in das .KLX hineinkommen, da der Linker diese auf meinen Wunsch hin raus lies. Damit habe ich mein Ziel erreicht! Meine Datenbereiche kann ich bequem definieren und ansprechen, ohne sie im .KLX drin zu haben - immerhin sind es einige.

Der Loader findet als Programmende (d.h. dort, wo sonst $MEMRY steht) den Anfang des DSEG-Bereiches, also genau das Ende des im .KLX wirklich enthaltenen Codes! Denke ich mir die DSEG-Bereiche weg, so entspricht das genau dem Ende des Ganzen, also dem, was der Linker sonst an $MEMRY schreibt.

Die Pufferanforderung ist die Länge des DSEG-Bereiches, damit der Loader den Bereich, den das DSEG belegt auf dem KLIX-Heap mit reserviert und ich so auch wirklich mit den Datenbereichen arbeiten kann.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: KLIX-Programmierung

Eine kleine Tücke bleibt leider bestehen! Wie erreiche ich es, daß der Linker den DSEG-Kram wirklich direkt hinter das Programm packt; schließlich erhält er als Eingabe folgendes:

1. CSEG-Bereich
2. DSEG-Bereich
3. Dazugelinkte Library-CSEG-Bereich(e)

Und das soll er umsortieren! Das macht man dadurch, daß man dem Linker sagt, wo er CSEG und DSEG hinstecken soll. Nur zu dumm, wenn man nicht weiß, wie lang die CSEG-Bereiche zusammen sind, um dem Linker sagen zu können, wo das DSEG hin soll, oder?

Üblicherweise werden KLIX-Overlays wie folgt erstellt:

1. Assemblieren als .REL
2. Linken an Adresse 100h in QUELLE1.COM
3. Linken an Adresse 200h in QUELLE2.COM bzw. QUELLE2.CIM
4. Mit GETREL daraus das .KLX erstellen

Das Linken muß anders werden. Original (SRLNK ist der SLR-Linker, für den L80 ist der Befehl entsprechend - Du findest Sie in Olaf's GETREL.SUB-Dateien):

2. SRLNK /A:100,QUELLE1/N,Programm,/E
3. SRLNK /A:200,QUELLE2/N,Programm,/E

Neue Variante:

2. SRLNK /A:100,/D:????,QUELLE1/N,Programm,/E
3. SRLNK /A:200,/D:????,QUELLE2/N,Programm,/E

Die /D:-Angabe legt fest, wohin das DSEG soll. Und es gehört bei KILX-Overlays direkt hinter das Programmende. Dazu mußt Du es erst einmal an irgend einen hohen Wert, z.B. /D:C000 bzw. /D:C100 (beim 2. Linken muß die /D:-Angabe um 100 höher als beim ersten sein) linken. Dann notierst Du Dir, bis wo die CSEGs jeweils wirklich gingen und trägst diese Werte dann beim eigentlichen Linken als DSEG-Adressen ein.

Der SLRNK+ kann das übrigens für Dich mit der Option /J übernehmen, die das DSEG automatisch direkt hinter das CSEG positioniert (oh wie praktisch):

2. SLRNK+ /A:100,/J,QUELLE1/N,Programm,/E
2. SLRNK+ /A:200,/J,QUELLE2/N,Programm,/E

Übrigens, wenn Du für Dein Programm zusätzlich zu den Datenbereichen im DSEG noch einen weiteren Puffer benötigst, dessen Länge sich der Anwender selbst aussuchen darf - wie z.B. die Größe des Textbereiches bei MTX-Edit, dann mußt im KLX-Header an Stelle von LenOfDSEG ein entsprechend größerer Wert angegeben werden.

Im Rahmen seiner Initialisierung muß das KLIX-Overlay weiterhin als erstes prüfen, ob der reservierte Pufferbereich mindestens so lang wie LenOfDSEG ist, da das KLIX-Overlay sonst Datenbereiche außerhalb seines Heap-Bereiches vermutet und anspricht.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: Trace-Macros

## Trace-Macros

(Herbert zur Nedden, 2071)

Gelegentlich kommt es sicherlich vor, daß man beim Schreiben eines Assemblerprogrammes gerne wissen möchte: 'Ja, wo laufen sie denn?'. Mir ging das bei RAM 6.2 so: Nach dem Einbau einiger umfangreicher Änderungen wollte es erst einmal nicht so recht in die Hufe kommen. Unter MONI kann ich es nicht testen, da es ja alles auf den Kopf stellt und folglich das laufende System killt. Natülich kann man einfach in das Programm an diversen Stellen irgendwelche Druckerausgaben einbauen: Hier ein 'A' ausgeben, dort ein 'B' und da ein 'C' usw. Das geht sicherlich gut, nur ist das, was auf dem Drucker rauskommt so kurz gehalten, daß man irgendwann beschließt aussagekräftigere Texte auszugeben. Und genau das macht das Macro TCALL zu dem ich jetzt komme - und zwar recht bequem in Sachen Nutzung:

Die Idee zu TCALL ist, daß es sich anbot, den Aufruf von bestimmten Unterprogrammen via CALL ausdrucken zu lassen. Und am besten druckt man einfach den Namen des aufgerufenen Upros aus. Der Aufruf

```
   TCALL  PeterUndPaul
```

liefert auf dem Drucker eine Zeile mit dem Text 'PeterUndPaul' und ruft natürlich auch noch PeterUndPaul auf. Alles was ich also in meinem Source machen muß, ist die interessanten CALL-Befehle durch TCALL zu ersetzen und schon erhalte ich das gewünschte Protokoll auf den Drucker. Hier TCALL

```
TCALL           MACRO   Was                     ; Macro hat einen Parameter
                local   print,msg,weiter        ; und drei lokale Labes
                push    hl                      ; Rette ein paar Register
                push    af
                ld      hl,msg                  ; HL = Meldung
print:          in      a,(4)                   ; Warten, bis Drucker
                and     1                       ; bereit (nicht Busy)
                jr      nz,print                ; und warten und warten
                ld      a,(hl)                  ; A = Nächstes Zeichen
                inc     hl                      ; Meldungs-Pointer +1
                or      a                       ; Ende der Meldung ?
                jr      z,weiter                ; Ja, dann weiter
                out     (4),a                   ; Zeichen an Drucker
                di                              ; und nach Memotech-Manier
                in      a,(0)                   ; den Strobe schön zu
                in      a,(4)                   ; Fuß erzeugen.
                ei                              ; Siehe Text zum EI!
                jr      print                   ; Weiter mit nächstem Zeichen
msg:            db      '&Was',13,10,0          ; auszugebener Text
weiter:         pop     af                      ; Register wieder her
                pop     hl
                call    Was                     ; und den eigentl. CALL
                ENDM
```

Um den Trace abzuschalten tut's das Macro so, statt aus TCALLs CALLs zu machen:

```
TCALL           MACRO   Was
                call    Was                     ; nur den eigentl. CALL
                ENDM
```

Sicherlich hat der eine oder andere jetzt ein paar Fragen auf der Pfanne. Zuvor möchte ich jedoch auf den Befehl 'EI' im Macro eingehen: Wenn Du dieses Macro in einer Routine einsetzt, die mit gesperrten Interrupts läuft wie z.B. einer Interrupt-Routine selbst, dann muß der 'EI' raus!

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

A s s e m b l e r: Trace-Macros

Jetzt die Fragen und Antworten:

F: Warum ausgabe auf den Drucker und nicht auf den Bildschirm?
A: Auf den Drucker kann ich mit obigen Befehlen etwas DIREKT ausgeben, benötige folglich nichts vom Betriebssystem und kann das alles machen, ohne mit Interrupts in Kollision zu geraten. Die Bildschirmausgabe wäre bei Interrupt-Routinen Mist, zermüllt die Glotze, ist vermutlich bei längeren Traces gerade, wenn es interessant wird weg, da aus irgendwelchen Gründen die Glotze gelöscht oder der Text überschrieben wurde.
F: Warum ist die ganze Routine zum Drucken im Macro drin und nicht irgendwo einmal im Programm abgelegt?
A: Die o.g. Lösung funktioniert, egal, auf welcher Bank sich das Programm gerade befindet - insbesondere, wenn das Programm auf mehreren Banks zu Hause sein kann. Anderenfalls müßte die Ausgaberoutine oberhalb von C000h stehen.
F: Wie kann ich leicht zwischen Trace an und Trace aus umschalten?
A: Durch eine IF-Abfrage im Source wie z.B. folgendes:

```
test      equ    1=1                          ; Testschalter
TCALL     MACRO Was
          if     test                         ; Testbetrieb ?
           TCALL-Definitionen wie oben  von LOCAL bis zum zweiten POP
          endif
          call   Was                          ; nur den CALL hinter's endif
          ENDM
```

Mit einem ähnlichen Macro kannst Du natürlich leicht Meldungen ausgeben, ohne sonst irgendwas auszulösen:

```
TNOP            MACRO   Was                   ; Macro hat einen Parameter
                jetzt genau das selbe wie das Macro TCALL, nur daß der CALL
                entfällt, was ein einfaches ';' leistet:
;               call    Was                   ; und den eigentl. CALL
                ENDM
```

Willst Du Werte ausgeben:

```
TVAL            MACRO   Was,Val               ; Macro hat zwei Parameter
                jetzt wieder das TCALL-MACRO hier her, bis zum Label weiter:
                dann so weiter
weiter:
                if      '&Val' = 'SP' or '&Val' = 'sp'
                 ld     hl,0                  ; LD HL,SP geht nicht
                 add    hl,sp                 ; also diesen Umweg
                else
                 ld     hl,&Val
                endif
                call    HLOut                 ; UPRO um HL auszudrucken
                pop     af                    ; Register wieder her
                pop     hl
                ENDM
```

Der Parameter Val enthält daß, was auszugeben ist. Es darf SP, eine Konstante oder ein Label sein. Mit ein paar weiteren IFs kannst Du das Macro auch auf einzelne Register A, B, ..., die Registerpaare BD, DE und HL sowie für IX und IY erweitern. Wie HL in Hex umgewandelt und dann ausgedruckt wird überlasse ich dem Leser (also Dir) als Übung.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: RAM 6.2

**RAM 6.2 - on Stage** (Herbert zur Nedden, 2071)

RAM 6.2 ist nun endlich fertig! Es hat mich ein gutes Stück Arbeit gekostet, da ich u.a. einiges in seinem Inneren umstricken mußte. Hier die Highlights:

## 1. Systemänderung - oder RAM 6.2 ist jetzt ein Betriebssystem

RAM 6.2 beinhaltet - im Gegensatz zu allen früheren Versionen von RAM - sowohl den CCP, natürlich einen ZCPR 3.3-CCP, als auch ein gebanktes BDOS auf Basis von P2DOS: RAM 6.2 ersetzt jetzt beim Laden alles, was sich im Speicher tummelt - folglich auch alle Systembestandteile. RAM 6.2 ist damit ein Betriebssystem und nicht mehr, wie seine Vorgänger lediglich (??) eine Betriebssystem-Erweiterung.

RAM 6.2 verlangt daher auch keinen ZCPR 3.3 beim Booten. Findet es nämlich keinen ZCPR 3.3, und damit keine zugehörige Multiple Command Line (MCL) vor, so stellt es den Befehl X:BOOTCONT in die Kommandozeile, die nach seinem Start durch ZCPR 3.3 (den RAM 6.2 ja mitbringt) ausgeführt wird, wobei das 'X' durch das Bootlaufwerk ersetzt wird. Damit kannst Du dafür sorgen, daß nach dem Booten von RAM 6.2 ohne ZCPR 3.3 bestimmte weitere Befehle ausgeführt werden, indem Du ein COM-Alias (mit SALIAS) unter dem Namen BOOTCONT.COM auf dem Bootlaufwerk erstellst.

Bootest Du jedoch RAM 6.2, wie ich es empfehle, so wie RAM 6.1 unter ZCPR 3.3, dann 'überlebt' natürlich die MCL, d.h. es bleibt wie bisher.

In allen Fällen wird jedoch auch weiterhin sowohl der Shell-Stack als auch der Message Buffer gelöscht. Das halte ich für nicht so schlimm, da sich Shells, wie MENU ggf. selbst reaktivieren können, indem sie sich hinter RAM 6.2 in der MCL verewigen.

Im Rahmen der Installation von RAM 6.2 mußt Du die Lage des BIOS und damit des BDOS und des CCP angeben - mit anderen Worten die Systemgröße festlegen. Bei maximaler Systemgröße (= 63.75 kB gemessan am BDOS) werden keine Named Directories (NDRs) unterstützt. Anderenfalls stehen 28 NDR-Einträge zur Verfügung und der Platz zwischen Free und Toam kann in weiten Grenzen (512 Bytes bis 11 kB) festgelegt werden. Der Platz zwischen Free und Toam wird u.a. für die von CP/M benötigten Allocation Vektoren der Laufwerke verwendet und sollte daher nicht zu klein sein.

## 2. Die weiteren wesentlichen Neuerungen im Überblick

RAM 6.2 benötigt zum Starten eine Systemgröße von mindestens 54kB. Nachdem es gestartet wurde ist die Systemgröße jedoch die, die in RAM 6.2 installiert wurde. Damit können nun auch die, die ein Boot-EPROM von Michael Keßler haben ohne Probleme zu einem 62kB-System kommen!

Ergänzt wird RAM 6.2 durch mein das KLIX Command Package: Das ist ein KLIX-Overaly, welches das Flow Command Package und das Resident Command Package in gebankter Form bereitstellt. Der Bereich, den diese Packages zusammen im Common belegen statt der bisherigen 15 nur noch vier Sektoren.

Meine mit K2Dos gemachten überlegungen, das RCP rauszuwerfen und dafür den CCP zu vergrößern habe ich verworfen. Zum einen bringt das eh nicht so viel, da ich für das FCP Platz im Common benötige und zum anderen nicht unbegrenzten Platz für den CCP in RAM 6.2 vorsehen wollte und konnte.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: KLICK.020

RAM 6.2 bietet einige neue UTILITY-Funktionen wie u.a. die, die das KLICK aktiviert und einige Tastencodes in den Tastaturpuffer stellt. Auch werden andere Tastaturtreiber (wie z.B. KbdMou) jetzt besser unterstützt.

Das Einschleifen von sog. User-Laufwerken, d.h. eigenen Treibern für Laufwerke wurde geändert. Das kommt daher, daß es als neuen RAM-Einsprung den Bank-Jump gibt und damit dieser Treiber ohne Stack im Common auskommt. Den einzigen mir bekannten Treiber für solche Laufwerke, der eine Edicta-Grafikkarte als RAM-Floppy nutzt, habe ich angepaßt (-> KLICK.020).

Weiterhin unterstützt RAM 6.2 auch logische Laufwerke, d.h. solche Dinge wie MountLib, jedoch in einer Form, die keinen Platz zwischen Free und Toam benötigt und sich wesentlich sauberer einschleifen läßt. Der Treiber wird in die Routine zum Lesen und Schreiben von Sektoren eingebunden. Er kann so je nach seinem Gusto sich selbst um den Zugriff kümmern (z.B. indem er die gewünschten Informationen aus seinem Puffer liefert) und/oder RAM 6.2 den Zugriff machen lassen (z.B. den Sektor von Disk lesen lassen) oder einen Fehler melden.

Mehrfachem Wunsch komme ich insofern entgegen, daß nun bis zu zwanzig Config-Codes resident sein können. Obendrein prüft jetzt auch der Cache das Format-ID, damit Du nicht aus Versehen eine 1C-Diskette mit Format 1D bearbeitest.

RAM 6.2 benötigt mindestens 144kB Speicher! Das liegt daran, daß Bank 1 nun komplett von RAM 6.2 eingenommen wird und die interne RAM-Floppy A: erst auf Bank 2 beginnt. Da auf A: die Systemspuren des Bootlaufwerkes stehen (beachte, daß SYSCOPY6 nicht mit nur einem Laufwerk arbeitet) und weiterhin SUB seine $$$.SUB-Datei auf A0: ablegt, muß A: mind. 16kB groß sein. Obendrein werden mindestens 16kB Heap benötigt, da für alle configurierten Laufwerke Informationen und ggf. der Cache-Puffer auf dem Heap landen.

3. Wo ist der Haken

RAM 6.2 kostet DM 25.- incl. Disk&Porto&Verpackung.

**KLICK.020** (Herbert zur Nedden, 2071)

Diese PD enthält einige KLIX-Overlays, die RAM 6.2 voraussetzen; KbdMou 3.0 ist die Version des Treibers, der die neuen Möglichkeiten von RAM 6.2 ausnutzt.

| | |
|---|---|
| ASCII62 | Anzeige der ASCII-Codes und der Bildschirm-Sequenzen für RAM 6.2 |
| DiJey 3.02 | Unterstützt jetzt auch Disketten mit Skew, Cursorpositionierung nun O.K., erfordert RAM 6.2 |
| GDC62 | RAM-Floppy-Treiber für Edicta-Grafikkarte, angepaßt auf die neue Mimik des Einschleifens von User-Laufwerken RAM 6.2 |
| INFO62 | KLIX-Info, erweitert auf die neuen Infos zu RAM 6.2 |
| KbdMou 3.0 | Nutzt RAM 6.2 aus (Abschalten MTX-Tastatur, Glotze an bei jedem Tastendruck), bessere Mausbedienung für NewWord/WordStar-Hauptmenü |
| KFLIB 1.1 | Assembler-KLIX-File-Library, unterstützt jetzt auch Skew |
| MTXMNU 1.3 | Neue Version mit weiteren Features |

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: KbdMou

**Maus und NewWord oder WordStar** (Claudio Romanazzi, 3070)

Das Clubtreffen hat es an den Tag gebracht - wir sind immernoch am Anfang mit unserem Latein. Zwar gibt es die ausgefeilteste Software, doch einige können nichts damit anfangen, weil sie das dazugehörige Wissen noch nicht ihr eigen nennen. Doch das ist nur eine Frage der Zeit.

Inzwischen geht auch die Zeit der MSDos-Welt weiter und gar manches ist es Wert, abgekupfert zu werden. Mit LOADDIR habe ich ja schon einmal soetwas vorgestellt. Jetzt reizt mich ein anderes Thema - die Maus. MTXMENU hat schon eine Mausoption verpaßt bekommen, doch die Hand sollte bis zum nächsten Programm ununterbrochen auf der Maus ruhen und nicht ab und zu zur Tastatur gehen. Daraus folgt zwingend ein weiteres Programm. Z.B. NewWord: Hier haben wir ein Programm, welches von den Machern nicht mehr gepflegt wird. Das heißt, wir müssen uns heute mit miesen Aufrufmimik herumschlagen.

**Anm.d.HzN:** Der Aufruf 'NW TEST.DOC' startet übrigens NewWord mit dem Dokument TEST.DOC; 'WS TEST.DOC PX' läßt WordStar sogar TEST.DOC gleich ausdrucken wonach WS wieder verlassen wird. So mies ist die Aufrufmimik von NW und WS nun wirklich nicht...

Diesem kann aber mit meiner Idee abgeholfen werden. Verzweigt man im Maustreiber, wenn ein bestimmter Code auftritt, zu einem Klix-Programm, so kann dieses dem Editor Befehle übermitteln. Will ich z.B. ein XYZ-File aufrufen, muß ich 'D' gefolgt vom Dateinamen eingeben (<Return> nicht zu vergessen). Statt dessen könnte ich ein Fenster öffnen, und eine Funktion anwählen. Ist das geschehen und die Funktion heißt Datei laden, dann könnte man mit den Cursor auf der Mattscheibe herumrennen und eine Datei anklicken, -> schwupps - geladen. Ergo keine kryptischen Dateinamen mehr eintippen etc.pp. Soweit ist das realisiert in KbdMou 2.10 (KLICK.019). Das ist der Tastaturtreiber den Herbert für sein neues Interface geschrieben hat. Ich habe meinen Source da eingeschleift.

Doch das ist noch nicht alles, könnte ich doch alle CTRL-K-Sachen auf eine Taste legen und damit mausfreundlich gestalten, um schon beim Ausgang aus einer Datei die Hand an der Maus haben zu können. Ein dankbares Feld wäre auch Turbo Pascal. Hier den Aufruf mit anderen Programmen gleichzumachen, wäre eine wahre Freude. Das kann man leicht (?) machen, indem man allen gängigen Programmen die gleiche Aufrufmimik verpaßt.

Das jedoch überlasse ich anderen oder mir im nächsten Jahr (d.h. 1992).

**KbdMou 2.01** (Herbert zur Nedden, 2071)

KbdMou 2.00 hatte leider eine kleine Macke (die sicherlich auch in Version 1.00 drin steckt): Shift-AlphaLock funktioniert nicht. Es ist schon erstaunlich, daß das noch keinem aufgefallen ist. Immerhin ist diese Tastenkombination, mit DiJey in Zusammenhang gesehen, eine, die ich in letzter Zeit immer wieder mal brauchte. Version 2.01 von KbdMou ist diesen Fehler los.

DiJey?? fragst Du Dich! Du solltest Dir mal den Befehl X von DiJey genauer ansehen! Er baut nämlich mit der angeXten Datei eine Kommandozeile in Abhängigkeit der Extension auf, die beim Verlassen des KLICK automatisch wie eine Funktionstaste ausgeführt wird, d.h. i.a. wird der Befehl gleich in die Tat umgesetzt. Shift-Home macht das ganze noch etwas schneller, da dabei KLICK gleich mit verlassen wird, damit Du Dir das Ret und die beiden Esc sparen kannst. DiJey kann folglich so etwas ähnliches wie Claudios LoadDir...

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: KbdMou

**KbdMou 2.10** (Herbert zur Nedden, 2071)

Claudio hatte (s.o.) die Idee, den Tastatur-Maus-Treiber dahingehend zu erweitern, daß mit ihm das Hauptmenü von WordStar oder NewWord bedient werden können - und zwar mit dem Ziel, nicht mehr den Dateinamen eingeben zu müssen.

Also schickte ich ihm eine Version von KbdMou, der beim Drücken einer bestimmten Taste(nkombination) den Interrupt beendete und in eine andere Routine verzweigte: mein Demo gab lediglich etwas auf den Bildschirm aus. Claudio realisierte in dieser Routine daraufhin die gewünschte Funktion, die er oben schon andeutete und die weiter unten näher beschrieben wird. Ich habe seine Lösung dann noch ein klein wenig überarbeitet und hier ist sie.

Mit der Tastenkombination SHIFT-CTRL-HOME (Du kannst Dir auch eine andere installieren) wird die Verzeigung aktiviert. In der 5. Spalte der 2. Zeile (Position ist natürlich auch installierbar) erscheint der Cursor - groß und hibbelig. Mit der Maus wird jetzt der gewünschte Kennbuchstabe angewählt. Bei eingen ist die Chose dann auch schon beendet: X, F L, J und Esc. Bei den anderen Buchstaben erwartet die Verzweigung, daß der Cursor nun in der Directory-Anzeige auf einen Dateinamen gestellt wird.

Beispiel: Mein Cursor erscheint an Position 5,2: direkt auf dem 'D' (Document). Drücke ich jetzt HOME (das ist der Anklicker, weil hier ja die Maus benutzt wird), passiert anscheinend nichts, intern wurde jedoch das D gespeichert. Klicke ich jetzt einen Dateinamen an, wird die Datei als Document geladen. Habe ich statt des D zu Beginn das N angeklickt, gibt's den Non-Document-Modus.

Da der Treiber die installierte Tastenkombination (hier SHIFT-CTRL-HOME) abfängt, ist sie ansonsten wirkungslos. Bin ich aus Versehen in der Verzweigung gelandet (es wird nur abgefragt, ob WordStar oder NewWord aktiv zu sein scheinen), komme ich mit SHIFT-HOME wieder raus.

**KbdMou 3.0** (Herbert zur Nedden, 2071)

Noch 'ne Version! Das liegt daran, daß die beiden oben geannten Versionen schon eine Weile auf dem Markt sind und nur Mangels eines Infos noch nicht gewürdigt werden konnten. Version 3.0 dieses Treibers bietet einige Neuerungen:

Drücke ich <CTRL-LEERTASTE> oder <CTRL-§>, so liefert der Tastaturtreiber den ASCII-Code 00h. Das taten die älteren Versionen von KbdMou nicht. WordStar benötigt ihn, da bei Proportionaldruck der Control-Code ^§ bedeutet, daß die entsprechende Spalte unproportional sein soll.

Die Mausbedienung für das NewWord/WordStar-Hauptmenü wurde verbessert. Der Cursor springt jetzt nur noch auf die relevanten Positionen (Kommandobuchstaben oder erstes Zeichen der Dateinamen), so daß die Bedienung nun sogar mit den Pfeiltasten möglich ist. Weiterhin ist das 'D' (installierbar) vorausgewählt, so daß der Cursor gleich zu Beginn auf den Dateinamen stehen kann. Als Abschluß wird nun <ESC> statt <RET> in den Tastaturpuffer gestellt, damit das Ausdrucken von Dateien bequemer geht.

RAM 6.2 bietet eine neue Utiltiy-Funktion mit der zum einen die Abfrage der MTX-Tastatur abgeschaltet werden kann. Das macht der Treiber um so wieder etwas von unserer kostbaren CPU-Zeit zu sparen. Weiterhin bietet diese Utility-Funktion die Möglichkeit, den Bildschirm wieder zu aktivieren (nachdem der Bildschirmschoner zuschlug). So wird nun der Bildschirm beim Drücken jeder Taste reaktiviert und nicht, wie bisher erst, wenn sich die gedrückte Taste auch auf dem Bildschirm auswirkt.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: MTX-Menu 1.3

**MTX-Menu Version 1.3** (Herbert zur Nedden, 2071)

Hier die Neuerungen:

- Statt des umständlichen Patchens per Moni o.ä. habe ich mich aufgerafft und ein .INS-File geschrieben.

- Auf Holgers Wunsch findet MTXMENU sein .Z80-File jetzt auch auf dem Rootlaufwerk. Bisher habe ich ja noch nicht gehört, daß evtl. jemand mehrere Service-Dateien auf verschiedenen Laufwerken hat. Für diesen Fall ist nichts verloren, die Sache ist installierbar.

- Als nützlich hat es sich erwiesen, Mauskoordinaten schonmal vorzuschreiben, damit beim Entwurf des Screens nur noch die Kommandozeile nachzutragen ist. Bisher war MTXMENU dagegen allergisch, weil explizit Kommandos erwartet wurden. Das ist jetzt geändert.

  Beispiel: 10,12,10,12CRLF

  Beim Beurteilen, wo der Cursor sitzt, wenn ich die Home-Taste betätige, befindet sich diese Kombination schon im Kommandoteil, enthält aber keine Kommandos, nur die Koordianten. Übersichtlichkeitshalber existiert sie schon, nur hat sie noch keine Bedeutung, ergo kommt jetzt auch keine Reaktion!

- Holger bemängelte, daß die Variablen, die ich in meinem Beispielmenu, daß ich leichtsinnigerweise auf dem Clubtreffen weitergab, nicht das Booten überleben. Recht hat er, denn sobald RamX die Bühne betritt, wird erst einmal der obere Ram-Bereich gefegt, sprich alles gelöscht! Sollte eines schönen Tages RamX den Shellstack auslassen, so wäre es möglich, Variable mitzunehmem und gültig bleiben zulassen. So aber....Die einzige Möglichkeit, die ich hier sehe, ist die, die ZCPR-Variablen in der Bootzeile zu bedienen und im ersten Menü auf die internen zu übertragen.

Holger ist übrigens einer der wenigen, die Reaktion geliefert haben. Soweit ich weiß, hat der Club doch über 70 Mitglieder. Benutzen die MTXMENU etwa nicht???

Holger hat dann nochmal zugeschlagen und mich zu erneuten Höchstleistungen angetrieben:

- MTXMENU kann jetzt mit Parameter aufgerufen werden. Dieser Parameter gibt an, mit welchem Menü angefangen wird. Der Parameter überschreibt beim Aufruf das installierte Erstmenü.

- Der Tastaturpuffer wird jetzt bei der Mausabfrage zuverlässig gelehrt.
  **Anm.d.HzN:** Heißt das evtl. geleert?

- Das Tastenfenster ist jetzt vollkommen benutzerabhängig. D.h., das ganze Fenster kann, soll, muß vom User beim Installieren eingerichtet werden. Das heißt natürlich nicht, daß ich nichts vorgebe. So wie es ist, kann es ohne Probleme bleiben ( = <RET> beim Installieren), jedoch kann man auch alles umschmeißen und eigene Geschöpfe programmieren.

- In der vorigen Version konnte MTXMENU bei Holger das Rootlaufwerk nicht finden, bei mir aber schon. Ich habe jetzt die Programmierung geändert, sodaß sie derjenigen von LOADER61.COM gleicht. Ich hoffe, die Probleme sind damit behoben.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: KLIX-Moni / L e s e r b r i e f: Claudio Romanazzi, 3070

**Anmerkungen zu KLICK-Moni** (KLICK.017) (Herbert zur Nedden, 2071)

Hier ein paar Anmerkungen zu KM, die auf Fragen von Claudio Romanazzi basieren:

- SP und PC stehen defaultmäßig auf 100h! Das ist so Absicht. Wenn ich POP HL ausführe landet auch das, worauf der SP zeigt (sprich der Inhalt von 100h) in HL. Willst Du den echten aktuellen SP haben, an dem das Programm unterbrochen wurde, so gib ein RHJ (= Register Holen Ja).

- Zur Frage, warum die bequeme Adreßeingabe nur bei List möglich ist, beim Dump aber erst ein L oder R vorne weg gebraucht wird; das L könnte doch entfallen! Bei List kannst Du direkt eine Adresse eingeben, da die Zeichen 0-F auf dem ersten Punkt gelegt wurden - und das ging nur, da keiner der anderen Auswahloptionen eben diese Zeichen als Selektionszeichen hatte. Sinn macht das hier, da es einen Standardpunkt gibt, der eine Adreßeingabe erfordert. Beim Dump gibt es immerhin zwei gleichwertige Punkte (L und R), die eine Adresse benötigen. Weiterhin ist es so, daß beim Dump häufig Eingaben der Form HL oder HL+A usw. gemacht werden ... Was soll daher wie (miß)verstanden werden ??

- Die Position des Trace-Fensters ist in der Tat etwas ungünstig. Andererseits verschwindet es doch gleich nach der Ausführung des Traces! Wer viel mit KM arbeitet, hat sich eh TS, TU usw. auf eine F-Taste (Tabelle 5) gepackt! Und dann ist's eh wurscht, niwwa? Daher hat es Olaf und mich bislang gestört.

Hast Du auch Wünsche oder Anregungen zu KM?

**Leserbrief** (Claudio Romanazzi, 3070)

Gerade habe ich eine Diskette für Herbert fertiggemacht und gepostet, da fällt mir ein, ich habe lange nichts mehr zu Papier gebracht:

Neulich las ich in einer c't, daß irgendsoein Dos die Eigenschaft hat, gelöschte Dateien noch weiterzuverwalten. Und das geht so: löscht man eine Datei, so werden die freigegebenen Blöcke nicht sofort wiederverwendet, sondern erst mal die noch garnicht beschriebenen. Sind die alle, wird bei den gelöschten Dateien nachgesehen, welche die älteste ist und deren Blöcke werden dann verwendet. Das erscheint mir doch äußerst anwenderfreundlich und sollte auch bei uns seinen Niederschlag finden.
**Anm.d.HzN:** Lieber Claudio, nimm Dir doch einfach mal die P2DOS-Sources aus K2Dos und bau es ein ... RAM 6.3 wäre dann evtl. der Platz, wo ich es dann hintun könnte.

Daß Herbert nach Monaten nur vier Artikel hat, finde ich sehr bedauerlich. Anscheinend ist unsere Kiste so ausgereizt, daß niemand außer ein paar unentwegten Geistern noch Beiträge zu liefern in der Lage ist. Schade!

Von mir gibt es (,obwohl bis März garkeine Zeit,) einiges zu berichten. MTXMENU ist inzwischen bei Version 1.3. Es sind nochmal einige Sachen dazugekommen, die Holger Göbel schmerzlich vermißte:
**Anm.d.HzN:** Habe ich auf die vorherige Seite ausgelagert.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

S o f t w a r e: KLIX-Moni / L e s e r b r i e f: Claudio Romanazzi, 3070

Das zweite Thema ist der Maustreiber. Ich hatte eine Verzweigung für den Maustreiber geschrieben, die bei Wordstar 4.0 und NewWord erlaubt, per Cursor und Druck auf die HOME-Taste Dateien zu bearbeiten. Wie auf KLICK.019 veröffentlicht konnte das natürlich nicht bleiben, weil es viel zu umständlich war. Naja, es waren die ersten Gehversuche und ich war froh, daß es überhaupt lief. Jetzt jedoch habe ich mich an die Feinheiten gemacht und den Cursor anständig programmiert:
**Anm.d.HzN:** Siehe oben, KbdMou 3.0

Ja, was machen die anderen denn so. Da ist Jan Bredereke mit seinem MountLib zu nennen. Ich kann nur sagen, das ist hervorragend, hat schon immer gefehlt, ist unverzichtbar und einsame Klasse. Endlich mal ein Programm, daß nicht nur hohen Gebrauchswert hat, sondern auch das Beiwerk kann sich sehen lassen. Die Dokumentation ist die beste, die ich je gesehen habe, inclusive meiner eingenen. Das liegt natürlich auch an der ausgefeilten Technik und der ausgeklügelten Planung von MountLib. Was da alles an Schnittstellen und Variablen, Lib's und Sources, Hilfsprogrammen und und und ... zu finden ist, zeigt, daß sich hier einer ans Werk gemacht hat, der sich Gedanken um seine (selbstgestellte?) Aufgabe gemacht hat. Solche Leute braucht der Club!
**Anm.d.HzN:** Ich muß schon sagen! Da haben Olaf und ich uns solche Mühe mit dem Handbuch zu RAM 6.0 gegeben und dann obige Kommantare! Das schmerzt!

Und dann wäre da noch Wolfgang Dexheimer. Auf KLICK.016 ist ja von ihm DAS Hammerprogramm. Wer hätte nicht schon einmal geträumt, mit seinem Compi Schecks oder Überweisungen auszufüllen und dabei auf Datenbanken und Computerunterstützung zurückgreifen zu können. Wer wie ich einen Geschäftshaushalt hat, weiß zu schätzen, was Wolfgang da gemacht hat. Auch hier sind soviele Dateien und Möglichkeiten vorgegeben, daß zuerst der Überblick verloren geht. Hat man dann ausprobiert, wie vorgeschlagen, was das System hergibt, so ist man, bin ich jedenfalls, begeistert. Auf das noch folgende Buchhaltungssystem bin ich jedenfalls sehr gespannt. Auch hier muß ich sagen, eine Superidee mit unseren 'wenigen' Mitteln hervorragend zu Computer gebracht. Weiter so und dein Wunsch, oh Wolfgang, daß die schwarze Kiste NICHT stürbe, wird gewährt.

Vielen Dank ihr beiden (an die anderen Macher ist man ja schon SO gewöhnt).

Und nun noch etwas anderes. Neulich schrieb mir Erik d'Hondt aus Belgien. Dabei fiel mir ein, daß er ja die Ideen hatte, unser Zeichensatzeprom zu verändern. Damals ermöglichte er die Rahmengrafik gemischt mit Buchstaben auf unserem Bildschirm darzustellen. Mir (und bestimmt auch euch, die ihr das lest) wäre es recht, wenn man den Zeichensatz nochmal verändert, um die dritte Dimension in die Rahmengrafik zu bringen. Unsere Rahmen sind in der Mitte eines ASCII-Zeichens positioniert. Das bedeutet, ein Schatten (für 3D eben) würde keinen Anschluß an den Rahmen haben. Ergo sollten entweder die Rahmenzeichen geändert werden, oder andere Zeichen als Randrahmen gepatcht werden. Zusätzlich bräuchte man noch die entsprechenden Schatten, sodaß man in der Lage wäre, eine virtuelle Lichtquelle von allen Seiten auf einen Rahmen scheinen zu lassen. Ich glaube die paar Zeichen sind noch frei, um sowas zu realisieren und außerdem, was die anderen können, können wir auch. Wie wär's?

Es grüßt euch alle

Claudio

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

**Kompressionsverfahren** (Herbert zur Nedden, 2071)

Allgemeines

Zweck von Kompressionsverfahren ist es, Dateien zu verkleinern - jedoch so, daß man aus der kleineren Datei die Original-Datei wiederherstellen kann. Mit Verkleinern ist gemeint, daß weniger Bits (oder auch Bytes) für die Daten gebraucht werden. Dabei werden nur die echten Daten mitgerechnet, und nicht die durch das Betriebssystem auf Disk zusätzlich vergebenen Bytes hinter den Daten, um den Sektor oder gar den Block aufzufüllen.

Folglich interessiere ich mich hier nicht für Libraries oder Archive, deren Idee es ist, mehrere Dateien in eine große zu stecken, damit das Betriebssystem nur einmal - nämlich am Ende der großen Datei, also der Library oder des Archives - einige Bytes anhängt, um den Block voll zu bekommen. Selbstverständlich sind Libraries und Archive trotzdem eine feine Sache:

Für viele Anwender ist der Zweck von Libraries und Archiven nicht so sehr die Platzersparnis, sondern in erster Linie die Möglichkeit der Strukturierung durch Zusammenfassung von Dateien. Wenn ich z.B. jemandem ein Programmpaket gebe stecke ich alle zugehörigen Dateien in eine Library, damit auch alles dabei ist und nichts verloren geht; auch bei der Übertragung vieler Dateien via Modem sind Libaries praktisch, da man bei der eigentlichen Übertragung nicht alle Dateinamen einzeln eingeben muß. In gewisser Weise sind Libraries und Archive unter CP/M ein Ersatz für Subdirectories... (sind aber besser kopierbar als SubDirs!)

Beachte jedoch, daß die in Archiven oder Libraries stehenden Dateien grundsätzlich bzw. ggf. komprimiert sind, um dadurch zusätzlichen Platz zu sparen.

Die in der CP/M-Welt gängigen Kompressions-Programme sind:

SQUeeze und UNSQueeze
CRUNCH und UNCRunch
CRLZH und UCRLZH

In der MsDos-Welt sind PKARC und ähnliche Programme recht verbreitet. Sie verbinden i.a. Kompressionsalgorithmen - und zwar u.a. die, die die o.g. Programme auch verwenden - mit Archiven.

Ich will hier verschiedene Verfahren erläutern, die es zum Zwecke der Datenkompression gibt, z.T. indem ich die Funktionsweise der o.g. CP/M-Programme erläutere. Implementationen der im folgenden beschriebenen Algorithmen für andere Betriebssysteme sind mit Sicherheit ähnlich, außer, daß evtl. verwendete Datenfelder anders dimensioniert, die Hashing-Formeln des LZW-Algorithmus oder der Aufbau der komprimierten Dateien etwas anders sind.

Grundsätzliches

Heutzutage versteht man Kompressionsprogramme so, daß sie aus zwei Teilen bestehen: einem Modell und einer Kodierung.

Aufgabe des Modells ist es, die Charakteristik der Originaldaten zu erfassen. Es versucht meist, in der einen oder anderen Form zu erkennen, ob bestimmte Dinge in den Originaldaten selten oder häufig vorkommen. Das kann sich auf die Häufigkeiten von Bytes beschränken aber auch das Vorkommen von Bytefolgen sein.
Aufgabe der Kodierung ist es, das, was das Modell sagt, platzsparend zu kodieren, sprich mit möglich wenigen Bits zu verschlüsseln.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Ein Beispiel dazu: Das Haupt-Modell vom Squeezen ist die Häufigkeit von Bytes, wobei die Bytes völlig unabhängig voneinander betrachtet werden; kodiert wird dieses Modell mit Huffman.

Im folgenden werde ich versuchen, einige Modelle und Kodierungen sowie deren konkrete Implementation in (De)Kompressions-Programmen zu beschreiben.

Sprache, Begriffe

Bei der Beschreibung der Kodierungsverfahren werde ich so vorgehen, daß ich mir vorstelle die Original-Daten zu lesen und kodiert wieder auszugeben, oder umgekehrt, d.h. zu dekodieren - die Richtung ist jeweils klar. Daher werde ich immer wieder davon reden, Bytes oder Bits einzulesen oder auszugeben. Das macht die Formulierungen etwas klarer.

Üblicherweise basieren die Modelle auf Bytes oder Zusammenfassungen von Bytes der Original-Daten. Die Kodierungen hingegen sind i.a. nicht nur auf Bytes anwendbar - die Huffman-Kodierung kann eigentlich alles kodieren, solange das, was kodiert werden soll gezählt werden kann. Daher werde ich bei der Beschreibung der Kodierungen davon schreiben Zeichen zu kodieren; bei der Beschreibung konkreter Implementationen gehe ich dann darauf eingehen, was die Zeichen wirklich sind.

Datenstrukturen und Algorithmen werde ich als Pseudo-Code darstellen, d.h. so ähnlich, wie es in einer Programmiersprache formuliert werden könnte.
Bei den Datenstrukturen bedeuten Byte und Zeichen eben dieses und Pointer eine Information, die das verwendete System als Speicheradresse, Zeiger auf Daten o.ä. verwenden kann; bei den CP/M-Programmen ist Pointer ein 16-Bit-Wort.

THEN- und ELSE-Zweige von IF-Abfragen sind durch Einrückungen kenntlich gemacht, wobei das THEN nie explizit angegeben wird. WHILE- und REPEAT/UNTIL-Schleifen ebenfalls eingerückt. Kommentare im Pseudo-Code sind *kursiv*.

Bei den Algorithmen werde ich mich meistens auf den Kern des Algorithums beschränken, mich also um Vorbereitungs- und Nachbereitungs-Routinen drücken. Damit halte ich die 'Listings' übersichtlicher und damit wohl auch verständlicher.

Zahlen- oder Code-Angaben mit einem 'h' dahinter bezeichnen Hex-Angaben. Entsprechend bedeutet ein 'b' am Ende, daß es sich um Bits handelt. Ansonsten handelt es sich bei Zahlen natürlich um normale Dezimalzahlen.

Da in der Literatur häufig die englischen (oder sind es gar amerikanische) Bezeichnungen für die Algorithmen kursieren, gebe ich diese in Klammern beim ersten Vorkommen mit an. Rein persönlich gefallen mir die englischen Ausdrücke sogar besser, aber trotzdem werde ich in diesem unserem Lande die deutschen verwenden.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Lauflängen-Kodierung (Run-Length-Encoding)

Dies ist wohl das einfachste Verfahren, Daten zu komprimieren - allerdings ist es nicht gerade besonders effizient; schade eigentlich! Aber dafür ist es sehr schnell. Die Idee ist die, ein Zeichen, welches mehr als einmal hintereinander vorkommt, nicht so oft es vorkommt , sondern nur einmal zusammen mit einem Zähler auszugeben. Die Kompression kann man sich in etwa so vorstellen:

```
Loop:  Zähler = 0
       REPEAT
          Lies Neues_Zeichen ein
          Zähler = Zähler+1
       UNTIL (Neues_Zeichen <> Altes_Zeichen)
       Gib Altes_Zeichen aus
       IF Zähler > 1
          Gib Zähler aus
       Altes_Zeichen = Neues_Zeichen
       GOTO Loop
```

Ein paar Problemchen sind sicherlich nicht zu übersehen: Wie kodiert man das mit dem Zähler - schließlich muß er ja von den normalen Zeichen zu unterscheiden sein. Weiterhin kostet der Zähler auch Platz, so daß es vermutlich wenig Sinn macht, ein Zeichen, welches nur zweimal hintereinander kommt durch den Zähler und das Zeichen selbst auszugeben (ist im o.g. Algorithmus nicht beachtet).

(Genaugenommen ist die Lauflängen-Kodierung eine Kodierung mit eingebautem Modell, welches mehrfach nacheinander auftretende Zeichen beachtet.)

Bit7-Lauflängen-Kodierung

Wenn die Zeichen, mit denen man es zu tun hat Bytes aus dem Bereich von 00h-7Fh sind, d.h. Bit 7 nie gesetzt ist, dann kann man sich das für die Kodierung des Zählers zu Nutze machen, indem man mehrfach vorkommende Bytes wie folgt kodiert:
Zähler+128 (sprich mit gesetztem 7. Bit) gefolgt vom Byte

Beachte, daß hierbei der Zähler max. 127 sein kann. Daß in diesem Beispiel im Gegensatz zum o.g. Algorithmus der Zähler vor dem Byte steht ist rein zufällig. Übrigens verwendet mein Programm MsgToMac, welches aus Textdateien Assembler-Sources für Anzeigeprogramme erzeugt, dieses Verfahren.

Standard-Lauflängen-Kodierung

Viele Kompressionsprogramme wenden die Standard-Lauflängen-Kodierung auf die Originaldaten an, bevor sie Ihre eigentliche Kompression vornehmen, um sich diese erste Platzersparnis schon mal zu sichern.

Mehrfach vorkommende Bytes werden bei der Standard-Lauflängen-Kodierung wie folgt in drei Bytes kodiert:
das Byte, 90h, Zähler

Da der Zähler max. 255 sein kann (steckt in einem Byte), müssen größere Vorkommen 'gestückelt' kodiert werden. Weiterhin muß für 90h eine besondere Kodierung bestehen, da dieses Byte bedeutet: Zähler folgt. 90h wird als 90h, 00h übertragen.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Der Dekompressions-Algorithmus:

```
Loop:  IF Warten_auf_Zähler                 gerade zuvor 90h gelesen ?
          Warten_auf_Zähler = Falsch        jetzt brauche ich nimmer zu warten
          Lies Zähler ein
          IF Zähler = 0
             Gib 90h aus                    90h wird als 90h, 00h kodiert
          ELSE
             Zähler = Zähler -1             -1, da schon 1x ausgegeben
       ELSE
          IF Zähler > 0                     Mehrfachbyte noch nicht fertig ?
             Gib Merk_Byte aus              nächste Kopie des Bytes raus
             Zähler = Zähler -1
          ELSE
             Lies Byte ein                  nächstes Byte
             IF Byte = 90h
                Warten_auf_Zähler = Wahr    Warten auf Zähler
             ELSE
                Gib Byte aus
                Merk_Byte = Byte
       GOTO Loop
```

Huffman-Kodierung

Die Idee der Huffman-Kodierung besteht darin, häufig vorkommende Zeichen in wenige und seltene Zeichen dafür in mehr Bits zu kodieren. Genaugenommen ist die Idee jeglicher Kompression, Häufiges mit wenigen Bits und Seltenes mit vielen Bits zu kodieren - denn so kann man ja vermutlich Bits sparen.

Nehmen wir mal an, daß die zu kodierenden Daten die vier Buchstaben A, B, C und D enthalten, die je zwei Bits belegen (für vier Zeichen genügen nämlich zwei Bit: 00b, 01b, 10b und 11b!). Kommt das A 10x, das B 5x, das C 3x und das D 2x vor, brauche ich für die Daten in Zwei-Bit-Verschlüsselung 20 Mal 2 Bit = 40 Bit. Kodiere ich jedoch das A als 1b, das B als 01b, das C als 001b und das D als 000b, dann brauche ich nur noch 35 Bit. Warum das mit den unterschiedlichen Code-Längen funktioniert wird gleich klar - hoffe ich.

Um zu den Bit-Kodierungen der Zeichen zu kömmen muß man die Häufigkeiten der zu kodierenden Zeichen kennen bzw. ermitteln. Dafür gibt es drei Wege:

Statisch (Static): Man geht einfach von angenommenen Häufigkeiten aus.

Dynamisch (Dynamic): Man liest einmal die Daten und zählt.

Adaptierend (Adaptive = Anpassend): Man beginnt mit bestimmten Startwerten (z.B. 'Alle Zeichen kommen gleich oft vor') und korrigiert die Häufigkeiten während der Kodierung.

In der Literatur werden die o.g. Begriffe statisch, dynamisch und adaptierend leider nicht immer so eindeutig wie hier verwendet. Adaptierend wird oft dynamisch und dynamisch dafür statisch genannt. Das liegt daran, daß einige Autoren von den Häufigkeiten der Zeichen (bzw. dem daraus erstellen Baum, der gleich erläutert wird) ausgehen; und die Häufigkeiten sind beim o.g. dynamisch während der Kodierung fest, also statisch, da sie zuvor einmal ermittelt wurden. Beim o.g. adaptierend hingegen, ändern sie sich laufend, sind also dynamisch.

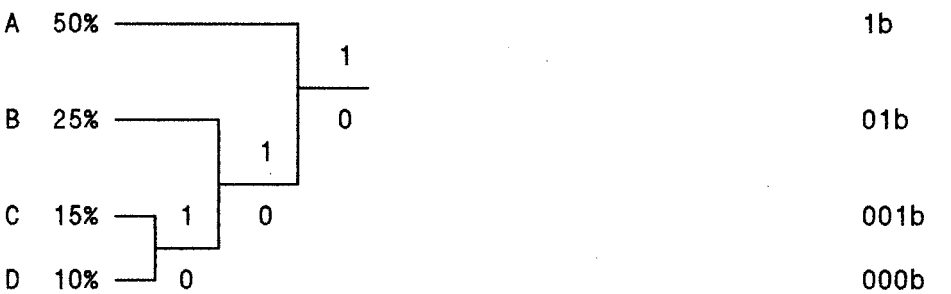
Es ist halt alles nicht so einfach, wenn man's doppelt nimmt ...

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

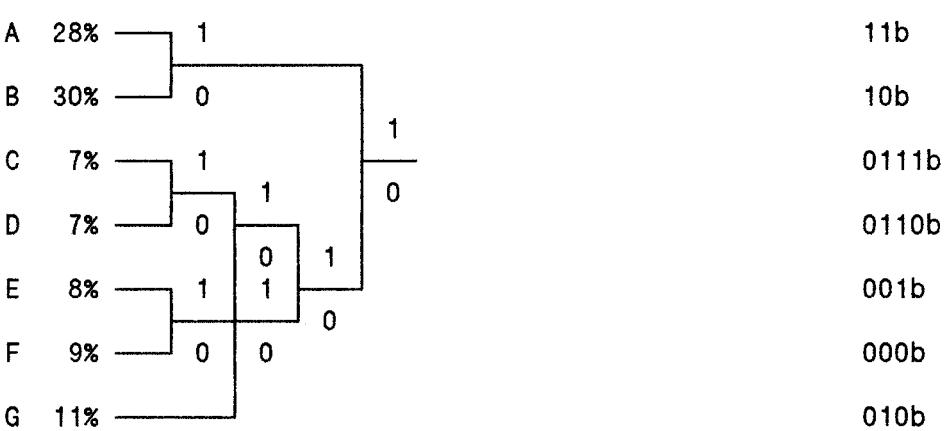
Liegen die Häufigkeiten der vorkommenden Zeichen vor, dann vergibt man die Bitkombinationen in passenden Längen, wobei seltene Zeichen mit besonders vielen Bits beschenkt werden. Das macht man, indem man sich die beiden seltensten heraussucht. Das eine bekommt eine 0b der andere eine 1b an seinen (zu Beginn noch leeren Code) gehängt. Nun faßt man beide zusammen - gemeinsam ist man bekanntlich stärker -: ersetzt die beiden durch ihre Summe in der Aufstellung der Häufigkeiten. Jetzt geht's von vorne los: Es werden wieder die beiden seltensten gesucht, einer mit 0b, der andere mit 1b beglückt und die beiden zusammengefaßt. U.s.w. bis nur noch einer übrig ist. Bekommt eine Zusammenfassung von Zeichen eine 0b oder 1b an Ihren Code gehängt, so bedeutet daß, das die Codes aller zugehörigen Zeichen um dieses Bit verlängert werden.

Ein Bild dazu (Beispiel mit den Buchstaben A, B, C und D von oben):




C und D sind mit ihren Häufigkeiten (15% und 10%) offenbar die seltensten. Also bekommen sie die 1b bzw. 0b, werden zusammengefaßt und sind nun zusammen mit 25% im Rennen. Jetzt sind B und C&D mit je 25% die seltensten: 1b und 0b vergeben und zusammenfassen. U.s.w. Rechts stehen die erhaltenen Codes der Zeichen, die man ablesen kann, indem man von der Spitze der o.g. Grafik, also der Wurzel des Baumes, der da zu sehen ist, zu den Buchstaben, also den Blättern des Baumes wandert.

Wenn mehr als nur vier Zeichen mitspielen, wird die Chose natürlich etwas aufwendiger, wie z.B.:




Eben diesen Baum (wenn man die Zeichen (hier die Buchstaben) etwas umsortiert, kann man seine Äste sicherlich mehr oder weniger entflechten - aber es bleibt ein Baum) kann man nun zum Kodieren und zum Dekodieren verwenden:

Beim Kodieren geht man von dem Zeichen aus zur Wurzel hin und erhält so den Binärcode, der für das Zeichen auszugeben ist; die Länge des Codes ist Länge des hierbei zurückgelegten Weges.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Beim Dekodieren stellt man sich an die Wurzel (allen Übels) und liest ein Bit. Entsprechend verzweigt man im Baum nach der einen (1b) oder anderen (0b) Seite. Jetzt liest man das nächste Bit und wandert wieder entsprechend im Baum weiter. Und das solange, bis man bei einem Blatt des Baumes angelangt ist, wo das gesuchte Zeichen steht.

Bleibt die Frage: 'Wie einigen sich der Kodierer und der Dekodierer über den zu verwendenden Baum?' Das geschieht abhängig davon, wie die Häufigkeiten der Zeichen für die Kodierung ermittelt wurden:

Staisch: Da die Häufigkeiten vorgegeben sind, 'wissen' beide bescheid.

Dynamisch: In diesem Fall muß der o.g. Baum zusammen mit dem kodierten Daten an den Dekodierer weitergegeben werden.

Adaptierend: Wie beim Kodieren wird der Baum auch beim Dekodieren der Daten aufgebaut - beide Programme müssen sich nur über den Startwert des Baumes einigen.

Was nun sind die Vor- und Nachteile der drei o.g. Varianten?

Statisch: Legt man die Häufigkeiten vorher fest, hat das den Nachteil, daß die Kodierung dann nur dann optimal ist, wenn die Häufigkeiten stimmen.
Der Vorteil ist, daß man die Häufigkeiten nicht vorher ermitteln und ggf. mit den kodierten Daten ausgeben muß.

Dynamisch: Zählt man erst mal alle Zeichen um deren Häufigkeiten zu ermitteln hat man zwar den vermeintlich (s.u.) optimalen Baum für die Kodierung, muß dafür jedoch die Daten zweimal lesen: einmal zum Zählen und einmal zum eigentlichen Kodieren.
Weiterhin müssen die ermittelten Häufigkeiten oder der daraus erstellte Baum zusammen mit den kodierten Daten an den Dekodierer mitgegeben werden, da dieser daohne nicht arbeiten kann.

Adaptierend: Beginnt man die Kodierung mit einem Startwert für die Häufigkeiten (i.a. mit 'alle Zeichen mit der gleichen') und paßt sie - und damit auch den Baum - mit jedem verarbeiteten Zeichen an, so ist die Kodierung zwar zu Beginn nicht optimal, wird aber im Laufe der Zeit immer besser. Der Vorteil dieser Methode gegenüber 2. ist, daß auch der Dekodierer die Häufigkeiten ermittelt und so den Baum selbst aufbauen kann. Folglich brauchen bei dieser Art der Huffman-Kodierung weder die Zeichen vorher gezählt noch der Baum mit übertragen zu werden.

In einigen Fällen kann die adaptierende Huffman-Kodierung sogar besser als die dynamische sein, da sie sich im Laufe der Zeit immer besser an die zu kodierenden Daten annähert. Ändert sich nämlich die Charakteristik der Originaldaten und damit auch die der zu kodierenden Daten, z.B. weil ein Programm komprimiert werden soll, welches im vorderen Teil Maschinencode und dahinter Datenfelder mit vielen Texten enthält, so paßt sich der adaptierende Huffman anfangs an den Maschinencode und später automatisch an die Daten an.

Übrigens müssen bei der Huffman-Kodierung die zu kodierenden Daten nicht unbedingt Bytes sein. Genaugenommen können die mit Huffman zu kodierenden Daten sein was sie wollen - Hauptsache man kann die Teile zählen und ihnen so Bitfolgen als Codes zuordnen. Weiterhin wäre anzumerken, daß die Huffman-Kodierung eine Kodierung und kein Modell ist.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Squeeze und Unsqueeze (CP/M)

Squeeze verwendet zur Kompression die Standard-Lauflängen-Kodierung und danach die dynamische Huffman-Kodierung auf Basis der Häufigkeiten der Bytes der Originaldaten. Folglich muß Squeeze die Datei zur Ermittlung der Häufigkeiten einmal lesen und dann zum Komprimieren nochmal. Die komprimierte Datei enthält naturgemäß den o.g. Baum und die komprimierten Daten.

Haben Sie das Modell von Squeeze, die Lauflängen-Kodierung mal bei Seite gelassen, erkannt? Es ist die Häufigkeit von Bytes, wobei diese völlig unabhängig voneinander betrachtet werden. Wenn man bedenkt, daß Huffman schon das mit den Häufigkeiten beinhaltet, scheint das irgendwie doppelt gemoppelt und verwirrend - aber so ist das Leben halt. Man könnte anscheinend auch sagen, daß Squeeze ohne Modell kodiert - nur stimmt das nicht, da Huffman selbst nichts darüber aussagt, was damit kodiert wird.

Die mäßige Kompressionsrate von Squeeze liegt nicht an der Huffman-Kodierung, sondern daran, daß das Modell die Charakteristik der Originaldaten nicht gut erfaßt. Die Annahme, daß alle Bytes stochastisch unabhängig voneinander vorkommen stimmt ja wohl kaum. Man denke nur einmal an den Buchstaben 'q' in deutschen Texten; nach einem 'q' kommt wohl in 99% aller Fälle ein 'u', also sind diese beiden Buchstaben nicht voneinander unabhängig.

Da die Algorithmen, die Squeeze zum Komprimieren verwendet oben beschrieben wurden, und damit auch das, was Unsqueeze zwangsläufig machen muß fehlt eigentlich nur noch die Information, wie Squeeze den aufgrund der Häufigkeiten der Zeichen erzeugten Baum in der komprimierten Datei unterbringt:

Der Baum wird als Array abgespeichert, von dem jeder Eintrag aus zwei Pointern besteht. Den ersten der beiden Pointer bezeichne ich als Left (Links), das zweite als Right (Rechts) - sie entsprechen nämlich dem linken bzw. rechten Unterbaum. Left wird genommen, wenn das gelesene Bit 0b ist, Right, wenn das gelesene Bit 1b ist.

Der Baum von Squeeze/UnSqueeze:

```
Baum:  ARRAY[0..Größe-1] of RECORD
                            Left:   Pointer   Für Bit = 0
                            Right:  Pointer   Für Bit = 1
                            END
```

Die Pointer sind 16-Bit-Worte mit folgender Bedeutung jedes einzelnen Pointers:

1. Ist Bit 15 nicht gesetzt, so steht hier die Nummer des Baum-Eintrags, wo es weiter geht.
2. Ist das höherwertige Byte = 0FEh so heißt das 'Fertig'.
3. Sonst enthält das niederwertige Byte das Komplement des Original-Bytes.

Die Größe des Baumes hängt davon ab, wieviele Bytes wie oft in den Originaldaten vorkamen - sie wird zusammen mit dem Baum in die komprimierte Datei geschrieben.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Der Algorithmus für's Dekodieren eines Bytes
(ohne Ende-Abfrage und ohne Lauflängen-Dekodierung!):

```
        Zeiger = 0                  Wurzel des Baumes
Loop:   Lies Nächstes_Bit ein
        IF Nächstes_Bit = 0b
           Wert = Baum[Zeiger].Left
        ELSE
           Wert = Baum[Zeiger].Right
        IF Wert > 0                 Prüfe Bit 15 auf 0
           Zeiger = Wert
           GOTO Loop
        ELSE
           Byte = Komplement von Low-Byte von Wert
```

Um die Dekompression zu programmieren, kann man zwei Wege beschreiten: Der eine ist der, den von Squeeze gelieferten Baum einzulesen, und mit dem o.g. Algorithmus zu bearbeiten. Es müßte dann nur noch die Abfrage auf das Ende-Zeichen eingebaut werden.

Die Andere (sicherlich schnellere) Variante der Dekompression besteht darin, aus dem eingelesenen Baum gleich die ensprechenden Befehle aufzubauen - was natürlich nur unter Assember sinnvoll machbar ist:

Man nehme folgenden Z80-Code:

```
Entry: CALL  GetBit         ; Lies ein Bit
       CP    0b             ; Prüfe, ob es 0b oder 1b ist (OR A geht auch)
       JR    NZ,Bit1        ; Sprung bei 1b
Bit0:  ......               ; Das, was bei 0b gemacht werden soll
Bit1:  ......               ; Das, was bei 1b gemacht werden soll
```

Diesen Code muß man für jeden Eintrag des Baumes erzeugen und an Stelle der Pünktchen je nach dem, was im jeweiligen Eintrag steht folgendes Eintragen:

- Ist Bit 15 des Pointers =0, der Pointer also ein Zeiger auf einen anderen Eintrag des Baumes, kommt hier ein Sprung an die entsprechende Stelle des erzeugten Codes hin, d.h. ein JP zum passenden Entry.

- Ist das höherwertige Byte = 0FEh ('Fertig') ein Sprung ans Ende.

- Anderenfalls wird hier ein LD A,nn und ein RET eingetragen. An Stelle des nn kommt hier das Original-Byte hin, sprich das Komplement des niederwertigen Bytes.

Der dem ersten Pointer eines Eintrages des Baumes (Left) ensprechende Code kommt hinter den Label Bit0:, der zum zweiten Pointer (Right) hinter Bit1:. Beachte, daß die an Stelle der Pünktchen eingetragene Code (JP nnnn oder LD A,nn mit RET) immer einen Sprungbefehl oder RET enthält und genau drei Bytes lang ist.

Das schöne an dieser Lösung ist, daß sie schnell ist, da das Interpretieren des Baum-Arrays nur einmal erfolgen muß - der daraus erzeugte Code ist, wie man sich unschwer denken kann, schneller als der unter Verwendung des ersteren Algorithmus, der bei jedem gelesenen Bit entweder aus dem erhaltenen Wort die phys. Adresse des Eintrages im Baum ermittelt oder das Byte komplementieret werden muß. Immerhin wird diese Dekodierung für jedes Byte der Originaldaten durchlaufen.

Beide o.g. Algorithmen müssen bei der Implemenieung noch um die Lauflängen-Dekodierung ergänzt werden.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Lempel-Ziv

Das von Squeeze verwendete Modell hat den Nachteil, daß es ausschließlich auf der Basis von Häufigkeiten von Bytes vorgeht. Insbesondere Programm-Sources enthalten häufig vorkommende Zeichenfolgen. Es wäre doch recht praktisch, wenn die Kompression statt auf einzelne Zeichen gleich auf Zeichenfolgen reagieren würde. Nehme ich z.B. einen Z80-Assembler-Source, so strotzt er von Zeichenfolgen wie dem 'LD' mit einem oder zwei Tabs davor und einem dahinter. Klar, daß bei Squeeze die Tabs sowie das 'L' und das 'D' mit recht kurzen Codes versehen werden, aber ...

Die Idee von Lempel und Ziv ist (genauer war) es, vorkommende Zeichenfolgen zu erkennen und diese mit einem Code zu versehen. Bei Squeeze (d.h. Huffman auf Byte-Häufigkeiten) hing die Bit-Länge des Codes der Bytes von deren Häufigkeit ab. Bei auf Lempel-Ziv basierenden Kompressionen hingegen ist die Bit-Länge der Codes fest oder zumindest vorhersehbar, aber dafür verbergen sich hinter einem solchen Code ggf. eine ganze Menge von Zeichen.

Ein Nur-Lempel-Ziv-Kompressionsprogramm ist mir nie untergekommen - schließlich ist Lempel-Ziv ein Modell. Es gibt jedoch einige Kompressionsprogramme, die darauf aufbauende Algorithmen verwenden, die im folgenden erläutert werden: LZW, LZSS, LZH.

Lempel-Ziv-Welch (kurz: LZW)

Der LZW-Algorithmus ist mitterweile in (mindestens) zwei Versionen in Kompressionsprogrammen implementiert:

Fest: Feste Codelänge von 12 Bit und keine Wiederverwendung einmal vergebener Codes.

Variabel: Variable Codelänge von neun bis 12 Bit und Wiedervwerwendung einmal vergebener Codes, wenn sie nicht gebraucht werden.

(Die Codelänge von 12 Bit kommt daher, daß das 'Gedächnis' vom LZW 4096 Einträge hat, für die ein 12-Bit-Pointer zur Adressierung genügt. Sollte eine Implementation mit einem größeren Gedächnis vorgenommen werden, muß an Stelle der 12 Bit ein entsprechend größerer Wert verwendet werden.)

Da beide im Kern den selben Algorithmus verwenden - genaugenommen ist letztere eine Verfeinerung/Verbesserung der ersteren -, beschreibe ich den LZW-Algorithmus erst einmal allgemein. Anschließend gehe ich im Rahmen der Beschreibung der beiden Versionen auf die Unterschiede ein.

Das 'Gedächnis' ist eine Tabelle, in der die Zeichenfolgen festgehalten werden, die schon erkannt wurden. Die Originaldaten werden solange gelesen, wie die ankommenden Zeichen als Zeichenfolge in der Tabelle enthalten sind; d.h. das Lesen wird unterbrochen, wenn eine neue Zeichenfolge gelesen wurde. Diese neue Zeichenfolge ist zwangsläufig um genau ein Zeichen länger, als eine in der Tabelle gespeicherte. Diese neue Zeichenfolge wird 'gelernt', indem das neue Zeichen in der Tabelle hinten an die vorhandene Zeichenfolge gehängt wird. Dann geht es mit dem Lesen weiter, wobei das letzte Zeichen, also das, welches die alte Zeichenfolge ergänzte, als erstes Zeichen der nächsten zu lernenden Zeichenfolge genommen wird.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Zu Beginn der Kompression und der Dekompression wird die Tabelle mit allen möglichen Zeichen (also allen 256 Bytes) als Zeichenfolgen der Länge eins initialisiert, damit überhaupt etwas zum Anfangen da ist. Bei diesen Zeichen wird in der Tabelle vermerkt, daß vor ihnen nichts mehr in einer Zeichenfolge kommt, d.h. sie der Anfang von Zeichenfolgen sind.

Beispiel: Die Zeichenfolge 'ABCDEFGH' wird Zeichen für Zeichen gelernt: Erst kommt 'AB' ('A' ist ja schon wegen der Initialisierung der Tabelle bekannt), dann 'ABC', dann 'ABCD', usw. Da die Zeichenfolge 'ABCD' in der Tabelle als 'D' kommt hinter 'ABC' steht, ist die Zeichenfolge 'ABC' auch weiterhin bekannt.

Was passiert, wenn die Tabelle voll ist erläutere ich weiter unten, da sich hier das Verhalten der beiden Versionen des LZW-Algorithmus unterscheidet.

Wie funktioniert die Dekompression?

Bei der Kompression werden Zeichenfolgen Zeichen für Zeichen gelernt und Information über diesen Leidens.... äh Lernweg ausgegeben, damit die Dekompression diesen nachvollziehen und so die Tabelle und damit auch die Originaldaten wiederherstellen kann.

Das Nachvollziehen des Kompression-Lernprozesses ist möglich, da, wenn eine neue Zeichenfolge bei der Kompression erkannt wird, die alte (um ein Zeichen kürzere) übertragen und dann mit dem neuen Zeichen als Anfang der nächsten Zeichenfolge weitergemacht wird. Die Dekompression macht nämlich ihrerseits folgendes: Nach dem Dekodieren eines Codes hängt sie das erste Zeichen der eben dekodierten Zeichenfolge hinten an die vorhergehende.

Die vielgenannte Tabelle, in der die Zeichenfolgen gemerkt werden ist 4096 (4kB) Einträge lang (bei den CP/M-Versionen 1.x und 2.x von Crunch zumindest). Jeder Eintrag besteht aus einem Pointer und einem Zeichen. Den Pointer bezeichne ich als Pred (kurz Predecessor = Vorgänger), das Zeichen als Suffix (= Anhängsel):

```
Tabelle: ARRAY[0..4095] of RECORD
                             Pred:   Pointer Zeiger auf Vor-Zeichenfolge
                             Suffix: Zeichen Folge-Zeichen der Zeichenfolge
                             END
```

Die Tabellen-Einträge schreibe ich im folgenden in der Form (Pred,Suffix).

Erinnern wir uns daran, daß die Zeichenfolge 'ABCD' dadurch gespeichert wird, daß in der Tabelle steht, daß vor 'D' die Zeichenfolge 'ABC' kommt. In der Tabelle sieht das so aus, daß der Eintrag, für die Zeichenfolge 'ABCD' in Suffix das 'D' und in Pred einen Zeiger auf den Eintrag der Zeichenfolge 'ABC' enthält.

Es gibt weiterhin noch einige besondere Pred-Werte:

NoPred: Dieser Wert kennzeichnet Einträge, vor denen keine Zeichen mehr in einer Zeichenfolge kommen, d.h. sie sind der Anfang eben dieser. (No Pred = Kein Vorgänger). Bei der Initialisierung der Tabelle werden alle möglichen Zeichen als Zeichenfolgen der Länge eins in die Tabelle eingetragen, also als (NoPred,Zeichen).

ImPred: Dieser Wert kennzeichnet Einträge, die in keiner Zeichenfolge vorkommen. Hiermit werden im Rahmen der Initialisierung bestimmte Tabellenplätze gegen Mißbrauch gesperrt - z.B. Tabellenplatz Nummer 0, da Zeiger mit dem Wert 0 meist bedeuten: Nicht belegt. (Impossible Pred = Unmöglicher Pred).

Free: Dieser Wert kennzeichnet freie Einträge in der Tabelle. Hiermit werden zu Beginn der Initialisierung alle Tabelleneinträge vorbelegt.

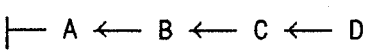
MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Da ein Bild mehr als 1000 Worte sagen soll, drücke ich mich verständlicher aus und spare sicherlich viel Platz, wenn ich den Inhalt und vor allem die Veränderungen der Tabelle grafisch und nicht in Prosa beschreibe. Ist die Zeichenfolge 'ABCD' in der Tabelle gespeichert, so enthält sie folgende vier Einträge, die diese Zeichenfolge darstellen:

1. (NoPred, 'A')
2. (Nummer des obigen Eintrages,'B')
3. (Nummer des obigen Eintrages,'C')
4. (Nummer des obigen Eintrages,'D')

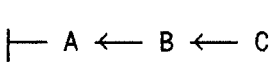
Grafisch:



Hier der LZW-Kompressions-Algorithmus:

```
        Alter_Pred = NoPred
Loop1:  Lies Zeichen ein
Loop2:  Suche (Alter_Pred,Zeichen)                  In Tabelle suchen
        IF Gefunden
           Alter_Pred = Adresse des Treffers von Suche
           GOTO Loop1
        Gib Alter_Pred aus
        Enter (Alter_Pred,Byte)                     In Tabelle eintragen
        Alter_Pred = NoPred
        GOTO Loop2
```

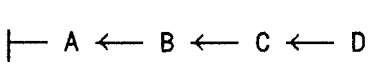
Für die Erläuterungen des o.g. Algorithmus nehmen wir mal an, daß in der Tabelle folgendes steht:



und die Zeichenfolge 'ABCD' gelesen wird:

| Zeichen | Aktion |
|---|---|
| A | Eintrag (Alter_Pred,'A') wird gefunden.<br>(Beachte, daß Alter_Pred zu Beginn = NoPred ist.)<br>Merke dessen Position in Alter_Pred |
| B | Eintrag (Alter_Pred,'B') wird gefunden.<br>Merke dessen Position in Alter_Pred |
| C | Eintrag (Alter_Pred,'C') wird gefunden.<br>Merke dessen Position in Alter_Pred |
| D | Eintrag (Alter_Pred,'D') wird nicht gefunden.<br>Gib Alter_Pred aus und trage Eintrag (Alter_Pred,'D') in Tabelle ein.<br>Setze Alter_Pred = NoPred ... und mache weiter. |

Damit ist die Tabelle nun:



Wie das Eintragen in die Tabelle funktioniert (im obigen Algorithmus als Enter bezeichnet) erläutere ich weiter unten, wenn ich auf die unterschiedlichen Versionen des LZW-Algorithmus eingehe. Anzumerken wäre noch, daß die Tabellenplätze per Hashing, d.h. durch Rechnen mit (Pred,Suffix) vergeben werden, damit die Suche nach einem bestimmten Eintrag schneller geht.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Warum wird eigentlich bei obigem Beispiel den Code für 'ABC' ausgegeben und das 'D' in die nächste Zeichenfolge verpackt statt gleich den Code für die neue Folge 'ABCD' auszugeben? Der Grund ist einfach: Was soll der Dekompressor denn sonst machen? 'ABCD' kennt er noch nicht!

Der LZW-Dekompressions-Algorithmus:

```
        Alter_Pred = NoPred
Loop:   Lies Neuer_Pred
        Decode Neuer_Pred                          Dekodieren
        Enter (Alter_Pred,Erstes_Zeichen)          In Tabelle eintragen
        Alter_Pred = Neuer_Pred
        GOTO Loop
```

Das Dekodieren (Decode) erfolgt rekursiv. Aus den komprimierten Daten wird der Zeiger auf das Ende der Zeichenfolge gelesen; daher muß sich das Programm durch die Tabelle rückwärts bis zu deren Anfang hangeln und dann die Zeichen vorwärts abliefern. Das erste Zeichen der so dekodierten Zeichenfolge wird in der Variablen Erstes_Zeichen gespeichert. Aufgerufen wird Dekodiere mit einem Zeiger auf einen Tabelleneintrag.

Decode *Aufruf mit Pointer als Parameter*

```
        (Pred,Suffix) = Tabelle[Pointer]
        IF Pred <> NoPred
           Decode Pred                             hier ist die Rekursion!
        ELSE
           Erstes_Zeichen = Suffix
        Gib Suffix aus
```

Nehmen wir mal an, daß in der Tabelle folgendes steht und daß Alter_Pred den Zeiger auf das 'C' enthält (dargestellt durch das '↑'), d.h. daß zuvor die Zeichenfolge 'ABC' dekodiert wurde:

```
├── A ←── B ←── C ←── D
                  ↑

├── 1 ←── 2 ←── 3 ←── 4
```

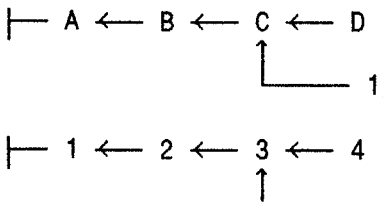
Wird als Neuer_Pred der Zeiger auf die '3' eingelesen, passiert folgendes:

Aufruf von Decode mit Zeiger auf die '3'. Per Rekursion wandert Decode zur '2' und dann zur '1'. Da der Pred des Eintrages mit der '1' NoPred ist, wird in Erstes_Byte die '1' gespeichert und nun die Rekursion wieder verlassend erst die '1', dann die '2' und die '3' ausgegeben. Damit ist Decode fertig.
Anschließend wird noch (Alter_Pred,Erstes_Zeichen) in die Tabelle eingetragen und in Alter_Pred der just gelesenen und dekodierte Pred gemerkt.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Nun sieht die Tabelle wie folgt aus




Zur Erinnerung: Die vorhergehende Zeichenfolge war 'ABC', dann kam '123'. Die Kompression hat nach Lesen von 'ABC1' gelernt, daß die '1' hinter 'ABC' vorkommen kann und dann die nächste Zeichenfolge mit der '1' beginnend gesucht.

LZW - Fixe Codelänge 12 Bit

Diese Version des LZW verwendet immer 12-Bit-Tabellen-Zeiger (bei 4k-Tabelle).

Wie oben erwähnt, werden die Tabelleneinträge abhängig vom Inhalt (Pred,Suffix) vergeben. Dazu wird mit Pred und Suffix etwas gerechnet (Hashing nennt man soetwas - nein nicht das Rechnen an sich, sondern die Errechnung eines Tabelleneintrags aus den Daten): Die errechnete Nummer des Tabelleneintrags ist die mittleren 12 Bit des Quadrats der Summe aus Pred und Suffix.

Was tun, wenn der so errechnete Tabelleneintrag belegt ist? Dann muß ein Ausweicheintrag in der Tabelle herhalten. Es wird einer gesucht und beim aktuellen Eintrag vermerkt, wo dieser ist. Dazu dient ein zusätzliches Array, welches man sich auch als Bestandteil der o.g. Tabelle denken kann. Damit haben wir als Datenfelder für's Gedächnis:

```
Tabelle: ARRAY[0..4095] of RECORD
                            Pred:   Pointer  Zeiger auf Vor-Zeichenfolge
                            Suffix: Zeichen  Folge-Zeichen der Zeichenfolge
                            END

Ersatz:  ARRAY[0..4095] of Pointer           Zeiger auf Ausweich-Eintrag
```

Der Ausweicheintrag wird wie folgt gesucht: Erst wird der Tabelleneintrag, auf den Ersatz zeigt untersucht. Gibt es ihn (d.h. ist Ersatz gefüllt), so geht die Suche mit eben diesem weiter. Hat das auch nicht funktioniert, addiert man 101 (das ist eine Primzahl!) auf die aktuelle Tabellenposition und sucht von dort an vorwärts den nächsten freien Tabelleneintrag; dabei wird nach dem letzten Tabellenplatz mit dem nullten weitergemacht. Ist er gefunden wird dessen Position in Ersatz eingetragen und der neue anschließend belegt. Im Rahmen der Initialisierung wird Ersatz mit dem Wert Leer vorbelegt.

Exkurs in die Mathematik: Warum ist das mit der Primzahl so wichtig? Damit wird sichergestellt, daß man nicht bei mehrfacher Addition der Zahl wieder auf meinen Startplatz komme (die Tabelle wird ja im Kreis abgelaufen) ohne vorher alle anderen Tabelleneinträge zu erwischen. Da hier die Primzahl nur einmal addiert wird, ist Ihre Verwendung nicht unbedingt erforderlich. Beim variablen LZW wird ebenfalls eine Primzahl bei der Suche nach Ausweichplätzen verwendet; dort wird sie jedoch erforderlich, da sie in dem Fall so oft addiert wird bis etwas herauskommt.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Der Algorithmus für das Eintragen von (Pred,Suffix) in die Tabelle:

```
   Enter  Aufruf mit (Pred,Suffix) als Parameter

          Berechne Adresse zu (Pred,Suffix)          Hashing: Mid-Square
   Loop:  IF Tabelle[Adresse] belegt
             IF Ersatz[Adresse] <> Leer              Ersatz-Kette abklappern
                Adresse = Ersatz[Adresse]
                GOTO Loop
          Alte_Adresse = Adresse
          Adresse = Adresse + 101
          WHILE Tabelle[Adresse] belegt
             Adresse = Adresse + 1                   Nach 4095 kommt 0!
          Ersatz[Alte_Adresse] = Adresse
          Tabelle[Adresse] = (Pred,Suffix)
          Ersatz[Adresse] = Leer
```

Der Grund, daß der Tabelleneintrag für (Pred,Suffix) errechnet wird ist, daß dadurch die Suche nach einem bestimmten (Pred,Suffix) in der Tabelle erheblich schneller möglich ist, als die Tabelle sequentiell abzugrasen. Und aus eben diesem Grund werden auch die Ausweicheinträge im Ersatz-Array vermerkt; schließlich wird bei der Kompression laufend in der Tabelle gesucht. Die Suche funktioniert damit dergestalt, daß erst aus (Pred,Suffix) der passende Tabelleneintrag errechnet wird; anschließend muß ggf. nur noch entlang der Ersatz-Kette nachsehen werden um festzustellen, ob der gesuchte Eintrag in der Tabelle steht (bekannt) oder nicht (neu).

Der Dekompressions-Ablauf ist genau der, der schon oben gezeigt wurde, allerdings mit der Änderung, daß sobald die Tabelle voll ist, das Eintragen neuer Codes entfallen kann:

```
          Alter_Pred = NoPred
   Loop1: Lies Neuer_Pred ein
          Decode Neuer_Pred                       Dekodieren
          Enter (Alter_Pred,Erstes_Zeichen)       In Tabelle eintragen
          Alter_Pred = Neuer_Pred
          IF Tabelle nicht voll
             GOTO Loop1
   Loop2: Lies Neuer_Pred ein
          Decode Neuer_Pred                       Dekodieren
          Alter_Pred = Neuer_Pred
          GOTO Loop2
```

LZW - Variable Codelänge und Codewiederverwendung

Der variable LZW komprimiert i.a. besser als der fixe, da er zwei Dinge kann und auch tut, die die letzterer nicht tut:

1. Die Codelänge ist neun bis 12 Bit. Das funktioniert, indem die Tabelleneinträge nicht irgendwie (d.h. mit einem Hashing-Algorithmus) vergeben werden, sondern die Tabelle von unten an gefüllt wird. Da im Rahmen der Initialisierung die ersten 256 Plätze für die möglichen Bytes mit Pred=NoPred belegt werden, braucht man mindestens neun Bit, um einen freien Tabelleneintrag adressieren zu können. Sind alle mit neun Bit adressierbaren Plätze belegt, geht es mit zehn Bit Codelänge weiter usw.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

2. Wozu soll man Tabelleneinträge eigentlich festhalten, wenn sie nicht (mehr) gebraucht werden? Das kann vorkommen, wenn eine neue Zeichenfolge gelernt und in die Tabelle eintragen wird, die nicht mehr (d.h. nur einmal beim Lernen) vorkommt. Die kann man doch getrost wieder vergessen und den frei gewordenen Tabellenplatz für eine neue Zeichenfolge verwenden.

Diese LZW-Version verwendet zusätzlich zu der o.g. Tabelle ein weiteres Array, d.h. hat folgende Felder:

```
Tabelle: ARRAY[0..4095] of RECORD
                            Pred:   Pointer  Zeiger auf Vor-Zeichenfolge
                            Suffix: Byte     Folge-Zeichen der Zeichenfolge
                            END

Zeiger:  ARRAY[0..4095] of Pointer           Zeiger auf die obige Tabelle
```

Die variable Codelänge funktioniert nur indem die Tabelle von unten an aufgefüllt wird. Andererseits ist das Errechnen der Tabellenplätze äußerst sinnvoll, damit die Prüfung, ob ein bestimmter (Pred,Suffix) schon in der Tabelle steht, schnell geht - sonst würde das Komprimieren ewig dauern.

Und genau hier kommt das Zeiger-Array, welches Zeiger auf die Tabelle enthält ins Spiel: Soll ein neuer Eintrag in die Tabelle, so wird im Zeiger-Array ein freier Platz ermittelt - natürlich per Hashing - und dort eingetragen, wo hin der neue Eintrag in die Tabelle kommt. Der neue Tabelleneintrag ist einfach der nächste freie. Im Rahmen der Initialisierung wird Zeiger mit dem Wert Leer vorbelegt.

Die Suche nach einem bestimmten (Pred,Suffix) funktioniert entsprechend: Es wird der zugehörige Zeiger-Eintrag errechnet; dort steht Adresse des gesuchten (Pred,Suffix) in der Tabelle.

Auch hier kann es natürlich beim Hashing zu Kollisionen kommen; in diesem Fall muß dann ebenfalls ein Ausweich-Tabelleneintrag gesucht werden.

Der Algorithmus für das Eintragen von (Pref,Suffix) in die Tabelle:

```
Enter  Aufruf mit (Pred,Suffix) als Parameter

       Berechne Adresse zu (Pref,Suff)          Hashing-Formel s.u.
       WHILE Zeiger[Adresse] <> Leer
          Adresse = Adresse + 5003              Neue Versuchsadresse
       Zeiger[Adresse] = Nächster_Code          (5003 ist eine Primzahl!)
       Tabelle[Nächster_Code] = (Pred,Suffix)
       Nächster_Code = Nächster_Code +1
```

Die Formel für die Berechnung der Adresse ist:

```
256* Low(Pref) AND 0Fh   +   Hi(Pref*16) XOR Suffix
```

Damit wäre erläutert, wie es möglich ist, mit kurzen Codelängen anzufangen ohne darauf verzichten zu müssen, die Adresse von Tabelleneinträgen errechnen und damit schneller finden zu können.

Beim Kom- und Dekomprimieren wird über die Anzahl der in der Tabelle belegten Einträge Buch geführt und immer, wenn ein weiteres Bit für die Adressierung erforderlich ist, die Codelänge automatisch erhöht.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Kommen wir zum zweiten Punkt: Dem Verwerfen unnötiger Codes.

Wie kann es vorkommen, daß eine Zeichenfolge gelernt wird, die man wieder vergessen kann - sprich nicht braucht? Immerhin kam sie doch einmal vor, wird also gebraucht, oder? Dazu müssen wir uns nur mal an den LZW-Algorithmus erinnern: Die neu erkannte Zeichenfolge ist um genau ein Zeichen länger als eine bekannte. Sie wird in die Tabelle eingetragen, indem das neue Zeichen hinten an die alte Zeichenfolge gehängt wird. Ausgegeben wird bei der Kompression jedoch der Code für die alte Zeichenfolge und die nächste zu suchende Zeichenfolge mit dem neuen Zeichen begonnen. (Siehe Beispiel oben).

Tja, und wenn eine so gelernte Zeichenfolge nie mehr vorkommt (sprich nur einmal beim Lernen), dann kann man sie doch gerne aus der Tabelle rauswerfen und den freigewordenen Platz für andere Zeichenfolge(n) verwenden.

Dazu wird im Feld Pred vermerkt, ob ein Eintrag referenziert wurde, oder nicht. Betrachten wir erst einmal den Dekompressions-Algorithmus:

```
        Alter_Pred = NoPred
Loop1:  Lies Neuer_Pred
        Decode Neuer_Pred                       Dekodieren
        Enter (Alter_Pred,Erstes_Zeichen)       In Tabelle eintragen
        Alter_Pred = Neuer_Pred
        IF Tabelle nicht voll
           GOTO Loop1
Loop2:  Lies Neuer_Pred
        Decode Neuer_Pred                       Dekodieren
        ReUse (Alter_Pred,Erstes_Zeichen)       In Tabelle wiederverweden
        Alter_Pred = Neuer_Pred
        GOTO Loop2
```

Im Vergleich zur vorherigen LZW-Version ist zu erkennen, daß, sobald die Tabelle voll ist, versucht wird, einen alten Code durch einen neuen zu ersetzen, d.h. unnötiges aus der Tabelle herauszuwerfen.

Die Routine ReUse, die überflüssige Codes aus der Tabelle rauswirft und durch neue ersetzt ist mit der Enter-Routine ähnlich:

```
ReUse   Aufruf mit (Pred,Suffix) als Parameter

        Berechne Adresse zu (Pred,Suffix)       Hash mich, ich bin der
                                                Frühling
Loop:   IF Zeiger[Adresse] <> Leer              Freie Einträge werden
           RETURN                               nicht wiederverwendet
        IF Tabelle[Zeiger[Adresse]] referenziert
           Adresse = Adresse + 5003             Neue Versuchsadresse
           GOTO Loop
        Tabelle[Zeiger[Adresse]] = (Pred,Suffix)  Neu belegen
```

Als erstes mag erstaunen, daß, wenn man im Zeiger-Array auf einen freien Platz stößt, diesen nicht verwendet. Wenn man jedoch bedenkt, daß die Tabelle voll ist und das Zeiger-Array mehr Einträge als die Tabelle hat, ist es klar, daß bei einer vollen Tabelle in Zeiger noch leere Einträge sein müssen - nur kann ich die nicht mehr gebrauchen - schließlich gehört zu ihnen kein Tabelleneintrag, dessen ich mich bedienen kann. Das Zeiger-Array ist so groß gewählt, damit beim Berechnen der Adresse zu (Pred,Suffix) Hashing-Kollisionen seltener vorkommen.

Irgendwie bin ich noch die Information schuldig, wann Einträge als referenziert gekennzeichnet werden.

MTX-Info - Copyright (C) 1983-1992 Herbert zur Nedden - dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Eingangs wurde gesagt, daß Einträge, die nicht gebraucht werden verworfen werden. Nicht gebraucht heißt, daß die Zeichenfolge, nur beim Lernen selbst aber sonst nicht wieder vorkam. Dazu wird beim Komprimieren und entsprechend auch beim Dekomprimieren vermerkt, ob ein Tabelleneintrag eine Rolle gespielt hat. Zusätzlich werden bei der Initialisierung die Einträge mit Pred=NoPred und mit Pred=ImPred als referenziert gekennzeichnet; d.h. gegen den Rausschmiß geschützt. Das relativiert das 'nicht (mehr) gebraucht wird' von Zeichenfolgen. Wurde sie nämlich bis zu dem Moment, wo die Tabelle voll wurde nicht mehr gebraucht ist sie schon Freiwild - es sei denn, sie wird doch noch mal gebraucht (sprich taucht auf), bevor sie aus der Tabelle rausfliegt.

Das o.g. Decode ändert sich dadurch minimal:

```
Decode Aufruf mit Pointer als Parameter

       Setze Tabelle[Pointer] auf referenziert
       (Pred,Suffix) = Tabelle[Pointer]
       IF Pred <> NoPred
          Decode Pred                          hier ist die Rekursion
       ELSE
          Altes_Zeichen = Suffix
       Gib Suffix aus
```

Crunch und Uncrunch (CP/M)

Dieses Programm-Paar komprimiert die Daten anhand des LZW-Algorithmus auf Byteebene, d.h. die Zeichenfolgen werden aus Bytes gebildet - und zwar in Version 1.x mit dem fixen und in Version 2.x mit dem variablen LZW.

Oben erwähnte ich spezielle Werte für Pred. Crunch kodiert NoPred, ImPred und Free wie folgt: Die Tabelle ist 4096 Einträge groß, so daß 12 Bit für die Adressierung genügen. Daher steht der Zeiger auf einen Tabelleneintrag, den Pred enthält, in den unteren 12 Bit dieses 16-Bit-Wertes, es sei denn Pred ist NoPred (FFFFh), ImPred (7FFFh) oder Free (höherwertiges Byte 80h). Der Initialisierungs-Wert Leer für die Arrays Ersatz und Zeiger ist 0000h bzw. 8000h.

Weiterhin dient Bit 13 von Pred bei Crunch Version 2.x als Kennung, daß der Eintrag referenziert ist: Ein gesetztes Bit 13 kennzeichnet refernzierte und damit gegen den Rausschmiß gefeite Einträge.
Anmerken möchte ich, daß die Pred-Werte FFFFh (= NoPred) und 7FFFh (= gesperrter Eintrag) Bit 13 gesetzt haben und damit als referenziert gelten. Damit sind die Initialisierungs-Einträge in der Tabelle gegen den Rausschmiß geschützt. Bei 80??h (Free) hingegen ist Bit 13 nicht gesetzt.

Crunch Version 2.x hat zusätzlich einige Codes mit besonderen Funktionen belegt: (Um jeglichen Problemen vorzubeugen, werden diese vier Codes bei der Initialisierung der Tabelle mit Pred = ImPred (= gesperrt) eingetragen.)

| | |
|---|---|
| 00 | Ende, also fertig mit der Arbeit |
| 01 | Reinitialisierung: Initialisiere Tabelle und Zeiger neu, d.h. vergiß alls soweit gelernte - aber dekodiere brav weiter. |
| 02, 03 | Reserve - noch keine Wirkung. |

Wann und warum Crunch im Rahmen einer Kompression all sein Wissen verwirft und von vorne mit dem Lernen beginnt weiß ich nicht! Vielleicht, wenn es merkt, daß die Kompressionsrate zurückgeht indem es Buch über die durchschnittliche Länge der erkannten Folgen führt.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Lempel-Ziv-Storer-Szymanski (LZSS)

LZSS ist eine Abwandlung des LZW: Es verwendet an Stelle der Tabelle als Gedächnis einen Ringpuffer in dem die letzten gelesenen Zeichen der Original-Datei stehen. Bevor eine Zeichenfolge bei der Komprimierung ausgegeben wird, wird nachgesehen, ob sie im Puffer steht. Wenn dem so ist, wird ihre Länge und Adresse relativ zur aktuellen Position im Ringpuffer übertragen, anderenfalls geht die Zeichenfolge selbst auf die Reise - unkodiert.

Obwohl es sehr einfach erscheint (die Tabelle mit den Zeichenfolgen und das Hashing des LZW entfallen) ist die Kompressionsrate verblüffend.

Vergleicht man mal die Wissensbeschaffung von LZW und LZSS fällt folgender Unterschied auf: LZW baut eine Tabelle aus den Zeichenfolgen auf, die am Dateianfang stehen - anschließend ist diese Tabelle mit dem Wissen recht statisch. LZSS hingegen sucht die Zeichenfolgen im Puffer, in dem die letzten paar kB gelesener Daten stehen - ändern sich die Zeichenfolgen, die in der Datei vorkommen paßt sich LZSS automatisch daran an.
Irgendwie erinnert das doch an den Unterschied zwischen dem dynamischen und dem adaptiven Huffman, nicht wahr?

Um den LZSS-Algorithmus zu erläutern muß ich einige Konstanten definieren:

PufferGröße: Die Größe des Ringpuffers, d.h. Anzahl von Zeichen die rein passen.
MaxLänge: Maximale Länge einer Zeichenfolge - mehr Zeichen werden beim Vergleichen einfach nicht herangezogen.
MinLänge: Mindestlänge einer Zeichenfolge.
Der Wert MinLänge minus 1 wird i.a. als Schwelle (Threshold) bezeichnet.

Der Puffer sieht eigentlich so aus:

```
Puffer: ARRAY[0..PufferGröße-1] of Zeichen
```

In Kompressionsprogrammen wird er jedoch in Wirklichkeit meist so aussehen:

```
Puffer: ARRAY[0..PufferGröße-1+MaxLänge-1] of Zeichen
```

Die zusätzlichen MaxLänge-1 Zeichen hinten am Puffer haben dabei stets den selben Inhalt wie die ersten MaxLänge-1 Zeichen. Sie stehen hier ein zweites Mal, damit beim Nachschauen, ob eine Zeichenfolge im Puffer steht, das Pufferende nicht beachtet werden muß - das bringt einiges an Geschwindigkeit.

Weiterhin werden bei der Kompression entweder die Original-Daten oder eine Längenangabe gefolgt von einer Adreßangabe ausgegeben. Damit der Dekompressor die erhaltenen Informationen auch verstehen kann, muß er zwischen einem normalen Zeichen und einer Längenangabe unterscheiden können.

Dazu wird einfach ddie Zeichenmenge erweitert: man legt zusätzliche Zeichen für die Längenangaben fest. Theoretisch könnte man wie bei der Standard-Lauflängen-Kodierung ein Zeichen für 'Längenangabe folgt' verwenden - das ist aber eigentlich auch nichts anderes als eine Erweiterung der Zeichenmenge: dieses ausgeguckte Zeichen gefolgt von der Längenangabe ist einfach das neue Zeichen für die jeweilige Längenangabe.

Immerhin ist LZSS das Modell - die platzsparende Verschlüsselung dessen, was das Modell sagt ist Sache der Kodierung, zu der ich noch kommen werde.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Kommen wir zum Kompressions-Algorithmus von LZSS.

```
Lies Zeichenfolge ein                          näheres s.u.
IF Länge ≥ MinLänge
   Gib Länge aus                               Länge ausgeben
   Offset = Zeiger - Adresse
   Gib Offset aus                              rel. Adresse ausgeben
ELSE
   Gib Länge Zeichen ab Puffer[Zeiger] aus     Zeichen selbst ausgeben
Zeiger = Zeiger + Länge                        Zeiger aktualisieren
```

'Lies Zeichenfolge ein' liest solange Zeichen von den Original-Daten in den Puffer, beginnend bei Puffer[Zeiger] ein, wie die gelesenen Zeichen schon an anderer Stelle als Zeichenfolge im Puffer stehen oder MaxLänge Zeichen gelesen wurden. Es liefert die Länge und die Adresse der Zeichenfolge im Puffer.
Folglich wird solange gelesen, wie die gelesenen Daten als Zeichenfolge schon mal übertragen wurden.

Die Dekompression:

```
IF normales Zeichen steht an                   Prüfe ob Längenangabe
   Lies Zeichen ein
   Gib Zeichen aus
   Puffer[Zeiger] = Zeichen                    Akutalisiere Puffer
   Zeiger = Zeiger +1                           ... und Zeiger
ELSE
   Lies Länge ein                              Länge
   Lies Offset ein                             Relative_Adresse
   FOR i = 1 to Länge
       Zeichen = Puffer[Zeiger-Offset]         hole Zeichen aus Puffer
       Gib Zeichen aus
       Puffer[Zeiger] = Zeichen                Aktualisiere Puffer
       Zeiger = Zeiger +1                       ... und Zeiger
```

Bevor ich zur Implementation des LZSS in Kompressionsprogrammen komme, möchte ich ein paar Anmerkungen zu den Längenangaben machen:

Üblicherweise sind die Zeichen, auf die der LZSS losgelassen wird ganz normale Bytes, also die Werte von 00h bis FFh. Die Längenangaben werden einfach durch die nächsten möglichen Werte, also 100h, 101h, usw. dargestellt. Um aus einer Längenangabe den zugehörigen Zeichen-Code zu erhalten addiert man dann 100h minus MinLänge dazu.

Richtig! Wenn ich die komprimierten Daten ohne weitere Behandlung ausgeben würde, bräuchte ich je neun Bit - mit acht Bit kann ich ja gerade die normalen Bytes kodieren. Man erinnere sich aber mal an Huffman - dem war doch die Zeichenmenge ziehmlich egal - Hauptsache man konnte die Zeichen zählen ... doch dazu mehr beim LZH-Algorithmus.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Lempel-Ziv-Huffman (kurz LZH)

Die LZH-Kompression kodiert das Modell LZSS mit dem adaptiven Huffman - vermutlich mit einer kleinen Ergänzung in der Art der Kodierung der Adreßangaben.

Wo liegt der Witz gerade Huffman zu verwenden? Gehen wir mal davon aus, daß Bytefolgen komprimiert werden sollen und Zeichenfolgen bei der LZSS-Komprimierung 3 bis 60 Bytes lang sind. Dann brauche ich 256+58 Codes (256 mögliche Bytes von 00h-FFh und 58 mögliche Längenangaben). Erinnern wir uns daran, wie der Baum für die Huffman-Kodierung aufgebaut wurde: Es waren überhaupt nur die Zeichen von Interesse, die wirklich vorkamen - also war es wurscht, wieviele es sind. Durch die Bildung des Baumes wird sichergestellt, daß die Codelänge in Bit für die Kodierung und Dekodierung geeignet und nicht unnötig lang ist.

Der Huffman bietet mir also eine Möglichkeit, meine 256+58 verschiedenen Codes gut zu kodieren - die krumme Anzahl stört nicht.

Offen ist noch die Frage, wie die Adreßangaben kodiert werden. Der Huffman wurde ja nur für die Übertragung der Original-Bytes und der Längenangaben vorgesehen. Wie die Adreßangaben kodiert werden können erläutere ich an folgendem Programm (ich weiß nämlich nicht, ob diese Art der Kodierung Bestandteil des LZH oder eine persönliche Note von Crlzh ist):

Crlzh und Uncrl Version 1.x (CP/M)

Diese Programme verwenden den LHZ-Algorithmus (Zeichen sind natürlich hier Bytes), kodieren jedoch die Adreßangaben auf ihre eigene Art und Weise.

Zur Kodierung der Adreßangaben: Da der Ringpuffer in der CP/M-Version 4kB lang ist, genügt eine 12 Bit-Adresse. Bei größeren Puffern ändern sich die Längen, das Verfahren kann jedoch das selbe bleiben.
Die oberen 6 Bit der Adresse werden über eine Tabelle in drei bis acht Bit kodiert. Dabei erhalten kleine Werte wenige Bit; da die Adreßangaben relative Adressen zur aktuellen Pufferposition sind, werden Zeichenfolgen, die 'jung' sind bevorzugt. Die unteren 6 Bit gehen unverändert raus. Insgesamt ist die Adreßangabe damit neun bis 14 Bit lang.

Beim Dekodieren einer Adreßangabe werden erst einmal acht Bit gelesen. Diese umfassen auf jeden Fall den Code für die oberen 6 Bit der Adreßangabe. Über eine Tabelle werden diese und die Länge des Codes mit dem die oberen 6 Bit kodiert wurden ermittelt.
Ist diese Länge weniger als acht, stehen in den unteren Bit des eben gelesenen Bytes schon acht minus Länge Bit der unteren 6 Bit der Adreßangabe.
Da insgesamt Länge plus 6 Bit ausgegeben wurden und bislang 8 Bit gelesen wurden, müssen noch Länge minus 2 Bit nachgelesen werden um alle Informationen zu erhalten.

Das mit der Tabelle für die Dekodierung ist recht einfach. Hier ein Beispiel zum Tabelleninhalt. Nehmen wir mal an, die Adreßangabe ist OOOOOQUUUUUUb (Die Buchstaben O = obere, U = untere sollen andeuten, um welchen Teil der Adresse es sich handelt). Dann wird OOOOOOb via Tabelle kodiert und UUUUUUb unverändert übertragen. Wird z.B. OOOOOOb in die fünf Bit XXXXXb kodiert, wird insgesamt für die Adreßangabe folgendes ausgegeben:

XXXXXUUUUUUb

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Bei der Dekompression wird erst mal das Byte XXXXXUUUb gelesen. Alle Einträge in der Dekodiertabelle zu XXXXX000b bis XXXXX111b liefern die selbe Längeninformation: fünf und für die oberen Adreßbits 000000b! Damit ist klar, daß noch fünf minus 2 Bit nachgelesen und mit den unteren acht minus fünf Bit des eben gelesenen Byte zusammengepackt werden müssen. So erhalten wir auch das UUUUUUb.

Das das mit den Tabellen auch so möglich ist, ist nicht zufällig! Die Tabellen sind mit Bedacht so aufgebaut worden - vermutlich mehr oder weniger über einen Baum wie beim Huffman-Algorithmus, wobei die kurzen Längenangaben bevorzugt wurden.

Damit ist der LZH-Algorithmus von Crlzh/Urclzh auch erläutert:

Modell = LZSS, Kodierung = adaptiver Huffmann erweitert um o.g. Adreßkodierung.

Anders ausgedrückt:

Kompression: LZSS-Kompression auf die Originaldaten.
Dann auf so komprimierten Bytes und Längenangaben den adaptiven Huffman, auf Adreßangaben die o.g. Kodierung anwenden.

Dekompression: Eingelesene Daten, wenn ein Byte oder eine Längenangabe gefragt ist via adaptivem Huffman dekodiert; Einlesen von Adreßanangaben erfolgt gemäß o.g. Dekodierung.
Dann LZSS-Dekompression

Arithmetische Kodierung

Wie der Name vermuten läßt, hat diese Kodierung etwas mit Mathematik zu tun (eigentlich haben das so ziehmlich alle Kompressions-Algorithmen).

Folgende zwei Punkte möchte ich jedoch der Beschreibung der arithmetischen Kodierung vorausschicken:

- Häufigkeiten von Zeichen (oder was auch immer) sind eigentlich Zahlen zwischen 0 und 1. Dabei bedeutet die 0 nie und die 1 immer. Oft werden Häufigkeiten jedoch als %-Zahlen angegeben. 25% liest sich irgendwie verständlicher als 0.25. Erinnert man sich daran, daß % nichts anderes als 'pro 100' bedeutet, wird deutlich, daß man die Häufigkeit nur mit 100 multiplitzeren muß, um die entsprechende %-Zahl zu erhalten.
- Ein Intervall besteht aus den Zahlen, die zwischen zwei Randpunkten (Intervallgrenzen genannt) liegen, wobei von den beiden Randpunkten keiner, einer oder beide zum Intervall dazugehören können.
  [a,b) bezeichnet das Intervall der Zahlen, die zwischen a und b liegen, wobei a dazu gehört (daher die eckige Klammer), b jedoch nicht (daher die runde Klammer). [0,1) bezeichnet folglich alle Zahlen die größer-gleich Null und kleiner als Eins sind.

Die Grundidee der arithmetischen Kodierung ist es, die Zeichen als Intervalle der Form [a,b) darzustellen, die so lang wie ihre Häufigkeiten sind und, beginnend bei Null, hintereinander gelegt werden. Da die Summe der Häufigkeiten der Zeichen 1 ist (was nichts anderes bedeutet, daß ein Zeichen immer irgendein Zeichen ist) ergibt diese Latte von Intervallen das Intervall [0,1). Klar?

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Hier ein Beispiel (es ist das selbe wie oben beim Huffman):

| Zeichen | Häufigkeit = Länge des Intervalls | Intervall |
|---|---|---|
| A | 0.50 | [0.00,0.50) |
| B | 0.25 | [0.50,0.75) |
| C | 0.15 | [0.75,0.90) |
| D | 0.10 | [0.90,1.00) |

Der Code eines Zeichens ist das zugehörige Intervall; wenn ich das Intervall kenne, kenne ich auch das Zeichen. Erfreulicherweise genügt es jedoch, irgendeine Zahl anzugeben, die in dem Intervall des Zeichens liegt, um das Zeichen eindeutig zu identifizieren. Der Wert 0.67 liegt z.B. im o.g. Intervall [0.50,0.75), entspricht folglich dem 'C'.

Da es darum geht, möglichst kurze Codes zu verwenden, nimmt man natürlich die kürzeste Zahl (gemessen in Ziffern), die in dem Intervall des Zeichens liegt. Da alle vorkommenden Zahlen im Intervall [0,1) liegen, sind sie der Form Null-Komma-Irgendwas; daher kann man sich diese Null-Komma in den ausgegebenen Codes schenken.

Als Codes für die o.g. Zeichen kann ich z.B. folgende verwenden, wobei ich die Null-Komma mit angebe, da es die Lesbarkeit erhöht:

A: 0.0 B: 0.5 C: 0.8 D: 0.9

Natürlich erfolgt die eigentliche Kodierung nicht auf der Basis von Dezimalzahlen, sondern unter Verwendung von Binärzahlen. Hier als Beispiel für o.g. Zeichen die Intervalle in binärer Angabe und die kürzesten Codes als Bitfolge:

| Zeichen | Häufigkeit | Intervall | kürzeste Zahl | zugehöriger Code |
|---|---|---|---|---|
| A | 0.50 | [0.000000b,0.100000b) | 0.0b | 0b oder nix |
| B | 0.25 | [0.100000b,0.110000b) | 0.1b | 1b |
| C | 0.15 | [0.110000b,0.111001b) | 0.11b | 11b |
| D | 0.10 | [0.111001b,1.000000b) | 0.1111b | 1111b |

(Die dezimale 0.10 ist binär 0.000110011001100110011001100...b, also ist 0.90 = 0.1110011001100...b; bei den o.g. Intervallen habe ich mich auf die ersten sechs Nachkomma-Binärstellen beschränkt.)

Irgendwie überzeugen die Codes nicht so besonders, oder? Bedeutet eine 1111b nun 'BBBB', 'CC', 'BBC', 'BCB', ... oder 'D' ? Lediglich eine erhaltene 0b ist hier anscheinend eindeutig! Nur kann man ja 0.0b auch als 0.b schreiben, so daß das 'A' auch ohne Code auskommen würde...

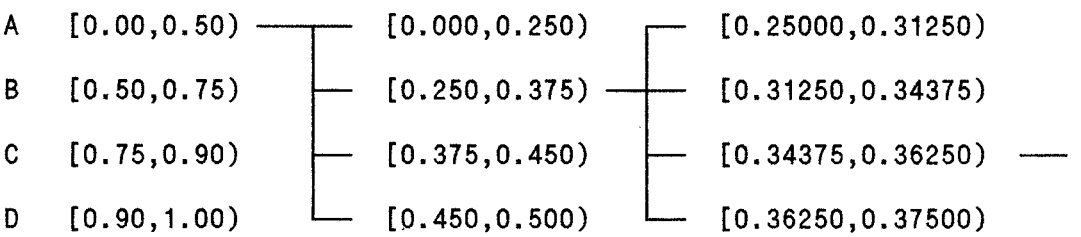
Obiges ist ja auch nur der erste Schritt der Kodierung. Bei der arithmetischen Kodierung werden nämlich Zeichenfolgen und nicht einzelne Zeichen in einen Code umgesetzt. Und das geschieht wie folgt:

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Das erste Zeichen einer Folge liefert ein Intervall. Jetzt stellen wir uns vor, daß dieses Intervall des Zeichens genauso wie zu Beginn das Intervall [0,1) entsprechend der Häufigkeiten der Zeichen unterteilt ist. Dann erhalten wir durch das zweite Zeichen wieder ein Intervall, welches im ersten liegt. U.s.w.

Als Beispiel mag die Zeichenfolge 'ABC' dienen. Da ich nicht binär denke, verwende ich Dezimalzahlen für die Intervallgrenzen - das ist sicherlich für Menschen so leichter les- und nachrechenbar.




A Das erste Zeichen 'A' liefert das Intervall [0.00,0.50), welches die Länge 0.5 hat. Dieses wird nun, wie zu Beginn das Intervall [0,1), entsprechend der Zeichenhäufigkeiten unterteilt. Damit erhalten wir die Intervalle in der mittleren o.g. Spalte, die zusammengenommen das vorherige Intervall, also [0.00,0.50) ergeben. Da das Intervall 0.5 lang ist, sind die darin enthaltenen neuen Intervalle 0.5 mal die Zeichenhäufigkeiten Lang.

B Das nächste Zeichen ist das 'B', so daß die Chose mit dem zugehörigen Intervall [0.250,0.375) weitergeht, welches seinerseits entsprechend der Häufigkeiten der Zeichen unterteilt wird und so zur nächsten Spalte mit Intervallen führt.

C Schließlich kommt das 'C', welches zum Intervall [0.34375,0.36250) führt. Und aus diesem Intervall nehmen wir uns nun irgendeine Zahl mit möglichst wenigen Stellen: z.B. 0.35. (In diesem Beispiel sind es natürlich Dezimalstellen, in Natura sind's Binärstellen.)

Die Dekodierung ist im Prinzip eine Wiederholung des obigen Spiels: 0.35 liegt im Intervall von 'A': [0.00,0.50). Damit ist das erste Zeichen schon einmal klar ist. Dieses Intervall wieder entsprechend der Häufigkeiten unterteilt läßt erkennen, daß jetzt ein 'B' kommt, da 0.35 in [0.250,0.375) liegt. Ebenso kommen wir zum 'C'.

Oben war ja noch das Problem offen, ob der Code 1111b 'BBBB', 'CC', 'BBC', ... oder 'D' bedeutet. Jetzt ist es sicherlich klar: Es muß das 'D' sein, da der Wert 1111b im Intervall [0.111001b,1.000000b) liegt.
'BBBB' wird über die Intervall-Folge [0.10b,0.11b) - [0.1010b,0.1011b) - [0.101010b,0.101011b) - [0.10101010b,0.10101011b) in die Zahl 0.10101010b und damit in den Code 10101010b umgesetzt.

Bevor ich zum Kodierungsalgorithmus kommen kann, muß ich noch einen Punkt klären: Woran ist das Ende einer kodierten Zeichenfolge erkennbar? Das geschieht dadurch, daß ein ungenutztes (ggf. neues) Zeichen als Ende-Kennung genommen wird. Normalerweise haben wir es mit 256 verschiedenen Zeichen zu tun: 00h bis 0FFh. Also nehmen wir einfach ein weiteres Zeichen als Ende-Kennung - es kommt natürlich in einer Zeichenfolge nur einmal vor - und berechnen die 257 erforderlichen Häufigkeiten.

In obigen Beispielen habe ich mehrfach Intervalle geteilt und geteilt usw. Um die Formeln zu erläutern, wie man das berechnen kann, brauche ich ein paar Bezeichnungen. Die Häufigkeit des Zeichens x nenne ich $H_x$, das Intervall für das Zeichen $I_x$.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Die Zerteilung geht so:

```
Teile  Aufrufparameter ist ein Intervall [a,b) welches zu teilen ist

       Länge = b-a                                  Länge von [a,b)
       Unten = a                                    Intervallanfang
       FOR x = Erstes_Zeichen TO Letztes_Zeichen    Alle Zeichen nacheinander
           Ix = [Unten , Unten + Hx*Länge)          Intervall zum Zeichen x
           Unten = Unten + Hx*Länge                 Nächste Intervall danach
```

Hier der Kodierungs-Algorithmus:

```
       Teile [0,1)                                  Startintervalle
       WHILE Zeichen da                             Bis zum bitteren Ende
          Lies Zeichen                              Zeichen her
          Teile IZeichen                            Intervall des Zeichens
                                                    ... teilen
       Nimm Zahl aus IEnde-Kennung                  Ende-Kennung ran
       Gib Zahl aus                                 Code ausgeben
```

Und die Dekodierung:

```
       Teile [0,1)                                  Startintervalle
       Lies Zahl                                    Lies den Code, d.h. die
                                                    ... Zahl von Kompression
       REPEAT
          Suche Intervall IZeichen in dem Zahl liegt  Intervall suchen
          Gib Zeichen aus                           Zeichen ausgeben
          Teile IZeichen                            Intervall des Zeichens
                                                    ... teilen
       UNTIL Zeichen = Ende-Kennung                 Bis zur Ende-Kennung
```

Die Ermittlung der Häufigkeiten der einzelnen Zeichen kann, wie bei Huffman, auf dreierlei Arten erfolgen:

Statisch (Static): Man legt sie einfach vorher fest

Dynamisch (Dynamic): Man liest einmal alle Daten und Zählt

Adaptierend (Adaptive = Anpassend): Man beginnt mit bestimmten Startwerten (vermutlich i.a. 'Alle Zeichen kommen gleich oft vor') und korrigiert die Häufigkeiten während der Kodierung.

Die Vor- und Nachteile sind natürlich sie selben wie bei der Huffman-Kodierung, so daß ich darauf nicht mehr eingehen muß.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Bleiben mir noch ein paar Anmerkungen zur Implementierung:

1. Bei der Teilung der Intervalle muß sichergestellt werden, daß kein Intervall auf die Länge Null schrumpft. Das ist möglich, falls die Differenz der Intervallgrenzen so klein wird, daß die Rechengenauigkeit nicht ausreicht.
   Üblicherweise werden die Daten nicht in eine eine Zeichenfolge mit einem einzigen (ewig langen) Code kodiert, sondern in mehrere Zeichenfolgen unterteilt - spätestens, wenn die Rechengenauigkeit nicht mehr mitspielt. Dazu muß die WHILE-Abfrage bei obigem Kodierungs-Algorithmus entsprechend modifiziert und sowohl der Ko- wie auch der Dekodierungsalgorithmus solange aufgerufen werden, wie Daten ankommen.

2. Bei der Kodierung wird nach und nach der Code aufgebaut - und wird immer länger. Sobald die ersten Stellen der Ober- und Untergrenze des aktuellen Intervalls, d.h. des Intervalls welches geteilt werden soll identisch sind, können diese Stellen schon ausgegeben und dann für die Kodierung verworfen werden. Sie können sich ja nicht mehr ändern.
   Damit entfällt die Notwendigkeit, den kompletten Code im Speicher zu halten. Außerdem verringert das Verwerfen der ausgegebenen Stellen die Länge der Zahlen, mit denen gearbeitet werden muß.

3. Bei der Dekodierung muß der Code auch nicht auf einmal eingelesen werden. Sobald genügend viele Stellen eingelesen wurden, um ein Intervall eindeutig zu identifizieren kann schon mit der Teilung begonnen werden. Ähnlich wie bei der Kodierung können führende Stellen verworfen werden, sobald sie bei der Ober- und Untergrenze des aktuellen Intervalls identisch sind.

4. Für die Implementierung bietet sich, ähnlich der LZH-Kompression die Verwendung der adaptiven arithmetischen Kodierung an, um nicht gezwungen zu sein, die Originaldaten zweimal zu lesen und die Häufigkeitentabelle zu übertragen.

Liv-Zempel-Arithmetisch (LZAri)

Diese Kompression ähnelt vermutlich dem LZH, nur daß an Stelle des Huffman die arithmetische Kompression angewendet wird.

Dies ist nur eine Vermutung, da ich noch nicht mit einem LZAri-Programm zu tun hatte, geschweige es untersuchen konnte.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Dateiformate unter CP/M

Abschließend möchte ich noch aufzeigen, wie CP/M-Dateien aufgebaut sind, die komprmierte Daten enthalten.

Beim Komprimieren von Dateien unter CP/M wird das mittlere Zeichen der Extension des Original-Dateinamens durch einen neuen Buchstaben ersetzt:

| | | |
|---|---|---|
| Squeeze: | Q | *Lauflängen und dynamischer Huffman* |
| Crunch: | Z | *Lauflängen und Lempel-Ziv-Welch* |
| Crlzh: | Y | *Lempel-Ziv-Storer-Szymanski- und adaptiver Huffman, jedoch eigene Kodierung der Adressangaben* |

Ist die Extension nur ein Zeichen lang, wird der o.g. Buchstabe einfach angehängt; hat der Dateiname keine Extension, so wird die Extension mit obigem Buchstaben gefüllt.

I.a. verweigern die Dekompressionsprogramme ihre Arbeit, wenn sie keinen der o.g. Buchstaben in der Mitte der Extension vorfinden - entsprechend weigern sich Kompressionsprogramme komprimierte Dateien nochmals zu komprimieren.

Alle komprmierten Dateien enthalten drei Teile:

Vorspann: Dieser beginnt mit einer Kompressionskennung und enthält weiterhin den Original-Dateinamen und einige Verwaltungsinformationen.

Daten: Die komprimierten Daten.

Nachspann: Dieser enthält ggf. eine Prüfsumme.

Abgesehen von der Kompressionskennung unterscheidet sich das von Squeeze verwendete Format des Vor- und Nachspanns von dem, den die anderen (Crunch, Crlzh) verwenden.

Die Kompressionskennung von Squeeze ist 76h gefolgt von FFh. Diese Sequenz wurde ausgewählt, da 76h der Opcode für HALT ist und so garantiert keine ausführbare Datei anfangen wird. FFh seinerseits kommt in keiner Textdatei vor. Damit ist diese Sequenz eigentlich recht unmißverständlich. Für Crunch und CrLZH wurde das FFh einfach um jeweils 1 auf FEh bzw. FDh erniedrigt.

Squeeze:

| | |
|---|---|
| 76h FFh | *Kompressionskennung Squeeze* |
| Prüfsumme (Wort) | *Prüfsumme: Summe der Original-Bytes modulo 10000h, also einfach als 16-Bit-Summe* |
| Filename.Ext | *Original-Dateiname, ohne Leerzeichen* |
| 00h | *Kennung: Ende des Dateinamens* |
| Baumlänge (Wort) | *Anzahl der Einträge im Baum* |
| Baum | *Huffman-Baum, je Eintrag zwei Worte* |
| Daten | *Komprimierte Daten* |

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

T h e o r i e: Kompressionsverfahren

Crunch/Crlzh:

| | |
|---|---|
| 76h FEh<br>76h FDh | *Kompressionskennung Crunch* bzw.<br>*Kompressionskennung Crlzh* |
| Filename.Ext | *Original-Dateiname, ohne Leerzeichen, die Extension jedoch dreistellig, wenn Stamp und/oder Info folgen* |
| 01h Stamp | *OPTIONAL: Timestamp 15 Bytes lang und 01h davor als Kennung. 00h-Werte im Stamp werden als 0FFh übergeben* |
| [Info] | *OPTIONAL: Informationstext, in eckigen Klammern.* |
| 00h | *Kennung: Ende der obigen Daten* |
| Revision-Level | *Versionsangabe* |
| Significant Revision-Level | *Signifikanter Teil der Versionsangabe, aus dem der Algorithmus hervorgeht. I.a. ist dies die Versionsangabe ohne Low-Nibble.*<br>*Crunch: <20h LZW Fix 12 Bit Codelänge*<br>*=20h LZW variable Codelänge und ReUse*<br>*>20h Zu hoch für o.g. Algorithmen*<br>*CrLZH: =10h LZH*<br>*>10h Zu hoch für o.g. Algorithmen* |
| Checksum-Flag | *00h = Mit 16-Bit-Prüfsumme wie Squeeze*<br>*01h = mit 16-Bit CRC*<br>*sonst ohne Prüfsumme* |
| Reserve-Byte | *I.a. steht hier eine 05h* |
| Daten | *Komprimierte Daten* |
| Prüfsumme (Wort) | *Prüfsumme bzw. CRC oder nichts* |

Dank

An dieser Stelle möchte ich Herbert Oppmann danken. Er hat mir einige Unterlagen zum Thema Kompression zu Verfügung gestellt und mir einige gute Tips für diesen Artikel gegeben.

Herbert zur Nedden

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de

K o m i k: Überlebenskunst

**Informatiker sind die besten Überlebenskünstler** (Herbert zur Nedden, 2071)

Erhalten habe ich diese Ausarbeitung von der Schneider/Armstrad User Group; aus deren Clubzeitung.

Man stelle sich einmal einen Informatiker im tiefsten Winter in einem dunklen Wald von hungrigen knurrenden Wölfen verfolgt vor. Hier ist der Informatiker geradezu in seinem Element. Er steht nämlich vor einem Problem, und solche zu lösen hat er ja während seines Studiums sehr ausführlich und mühsam erlernt.
Das Problem ist zwar bereits gegeben, aber irgendwann einmal hat er vor langer, langer Zeit gelernt, daß ein Problem erst spezifiziert sein will. Er beginnt also:
Gegeben: Landschaft mit 1 Informatiker und n Wölfen, n aus NAT;
Gesucht: Landschaft mit 1 Informatiker und keinen Wölfen.
Lösungsweg: Wölfe mit einem Prügel verjagen.

Sicher kann sich unser Informatiker denken, daß das Problem nicht einfach zu lösen ist. Also beginnt er es in Teilprobleme zu zerlegen. Etwa in die n Teilprobleme für alle i aus {1..n} den Wolf i verjagen.
Nun ist unser Informatiker überglücklich. Er benutzt eine simple FOR-Schleife, in der er nacheinander die n Teilprobleme löst und somit seine Teillösungen sogar schon zu einer Gesamtlösung zusammengesetzt hat.
Daß der Algorithmus korrekt ist und terminiert, hat unser Informatiker schnell beweisen. Was nun weiter geschieht, ist typisch, wenngleich es zwei Möglichkeiten gibt.

**Fall 1** - Wir haben einen Durchschnittsinformatiker vor uns. In Ermangelung eines Rechners benutzt er sich selbst als Maschine und läßt das Programm auf SICH ablaufen. Er beginnt damit, Wolf Nr. 1 zu verjagen, kommt zu Wolf Nr. 2, doch spätestens jetzt hat ihn ein Wolf, der laut Algorithmus noch gar nicht an der Reihe ist, ins Bein gebissen, worauf er in Panik gerät, das ganze schöne formale Denken vergißt und einfach instiktiv die Flucht ergreift. Später dann, wenn er wieder in Sicherheit ist und wieder klar denken kann, bricht eine ganze Welt in ihm zusammen. Dies kommt davon, wenn man sich als Durchschnittsinformatiker mit praktischen Problemen beschäftigt.
**Fall 2** - Ganz anders, wenn wir einen hochbegabten, mathematisch besonders geschulten Informatiker aus Karlsruhe in die Wildnis schicken, der schon nach dem 3. Semester das Vordiplom und nach dem 7. das Hauptdiplom gemacht hat. Er sieht zwar n Wölfe, zweifelt jedoch daran, daß die Zahl der Wölfe ohne sein Zutun konstant bleiben wird. Es könnten ja während des Verjagens noch nicht verjagte Wölfinnen Junge werfen. Um den Aufwand des Wölfeverjagens unter diesem Aspekt abzuschätzen, muß zuerst eine Differentialgleichung gelöst werden, ganz abgesehen davon, daß das Problem neu spezifiziert werden muß. Mit Erschrecken stellt unser Informatiker jedoch fest, daß ab einem bestimmten n der Algorithmus nicht mehr terminiert (es werden in gleicher Zeit mehr Junge geworfen, als er Wölfe verjagen kann). Er wird also eine neue Spezifikation vorhemen:
Gegeben: Ort a mit n Wölfen und 1 Informatiker, ein Ort b;
Gesucht: Ort a mit n+k Wölfen (k ist die Anzahl der zwischenzeitlich geborenen Wölfe), ein Ort b mit mindestens 1 Informatiker.
Lösungsweg: Flucht von Ort a nach Ort b.

Nach Ausführung seines Algorithmus trifft er dann unseren Durchschnittsinformatiker, der wahrscheinlich auf eine Baumspitze geflüchtet ist, wohin er sich eilends auch begibt und wartet, bis die Wölfe abziehen.
Sind die Wölfe erst weg, so werden sich beide Informatiker schnell darüber einig, daß man den Baum am besten per rekursivem Abstieg herunterkommt. Da sie lange auf dem Baum saßen, waren sie stark durchgefroren. Doch zum Glück kam ihnen eine alte Algorithmenentwurfsmethode entgegeb, und eine alte Axt, die herumlag, entpuppte sich als ein ausgezeichnetes Programmierwerkzeug.

MTX-Info – Copyright (C) 1983–1992 Herbert zur Nedden – dieses PDF darf nur auf www.mtx-info.de online stehen / this PDF may only be online on www.mtx-info.de