# Teil V
# Dateiformate

An sich sollte es selbstverständlich sein, daß eine Programmdokumentation auch eine Beschreibung der verwendeten Dateiformate enthält. Wie so vieles ist es das nicht, was mich aber nicht hindern soll, hier mit gutem Beispiel voranzugehen.

## 1 CHAOS*ultd*-Bilder

CHAOS*ultd* verwendet für Bilder ein eigenes Format[34]
Eine Bilddatei besteht aus einem Header, der Farbtabelle, den Parametern, zusätzlichen Parametern und den Bilddaten, wobei letztere immer gepackt werden, auch dann, wenn sie dadurch länger werden[35].
Der Header sieht (als C-Struktur) folgendermaßen aus:

```
typedef struct
{
/* id's */
char chaos_id[8]; /* CHSultd5 */
char frac_id[8];
/* größe, planes und colors */
unsigned int size_x;
unsigned int size_y;
unsigned int planes;
unsigned int colors;
/* zus. parameter */
int lastline;
int last_flag;
int xor_offset;
/* länge der folgenden daten */
long par_len;
long xpar_len;
long pic_len;
/* coltab in long */
/* parameter */
/* zus. parameter */
/* bilddaten */
} FRAC_HEAD;
```

`chaos_id` ist eine Kennung, die die Zeichen `CHSultd5` beinhalten muß. `frac_id` ist eine weitere Kennung, die die Berechnungsroutine, die für das Bild verantwortlich ist, kennzeichnet.

[34] Wozu Standartformate verwenden, wenn man auch eigene definieren kann?
Aber im Ernst: die verschiedenen Bildgrößen und die Verwaltung der Parameter ließen mir ein spezielles Format sinnvoll erscheinen und wenn man sowieso ein eigenes Format verwendet, ist es auch schon egal, wie speziell dieses Format dann ist (denkbar wäre höchstens noch ein GEM-Image-Format gewesen, mit einem erweiterten Header; vielleicht führe ich das mal als alternatives Format ein).

[35] das ist kein Scherz, das kann wirklich passieren, die Daten werden aber nicht nennenswert länger

`size_x`, `size_y`, `planes` und `colors` legen die Bildgröße, die Anzahl der Bitebenen und die Zahl der Farben fest. Die Bildgröße ist – unabhängig davon, ob und wieweit das Bild schon berechnet wurde – die Größe des fertig berechneten Bildes.
`lastline` gibt an, ob das Bild fertig ist oder nicht. Ist `lastline` 0, so gibt es gar keine Bilddaten, -1 bedeutet, daß das Bild fertig berechnet ist. Andernfalls gibt `lastline` die Anzahl der berechneten und gespeicherten Zeilen an, aber nur falls das unterste Bit in `last_flag` gesetzt ist[36]. Ist dieses Bit gelöscht, so ist für `lastline`$\neq 0$ stets das gesammte Bild gespeichert.
`xor_offset` ist für das Packen der Daten von Bedeutung und wird gleich erklärt; `par_len`, `xpar_len` und `pic_len` gibt die Länge der drei Datenblöcke an. Die Parameter und zus. Parameter sind natürlich Bildtypabhängig, bleiben also die Bilddaten.
Im Farbmodus werden zunächst die Bitplanes, die im Screen-Format ja wortweise hintereinanderstehen getrennt, d.h. es kommt erst die ganze Bitplane 0, dann 1 usw. (man hat also gewissermaßen ein GEM-Standartformat, wobei die Größe des Blockes durch `size_x` und `size_y` festgelegt ist).
Dann werden die Bilddaten von hinten mit dem `xor_offset` mit sich selbst via xor verknüpft. Der Offset entspricht im Allgemeinen einer oder mehreren Bildschirmzeilen. Dadurch entsteht ein Bild, dessen Zeilen (außer der ersten) immer nur die Differenz zur darüberliegenden Zeile beinhalten und das deshalb i.a. besser zu packen ist. Ist `xor_offset` 0 so entfällt die Xor-Verknüpfung. Um die Xor-Verknüpfung beim Laden rückgängig zu machen muß man die Daten erneut via xor miteinander verknüpfen, diesmal allerdings von vorne.
Anschließend werden die Bilddaten im Stad-Format gepackt und gespeichert.
Das heißt die gepackten Daten beginnen mit den drei Bytes Kennbyte, Packbyte und Spezialbyte, anschließend folgen die gepackten Graphikdaten, wobei folgende Kodierungen Anwendung finden:
Das Kennbyte und das nachfolgende Byte n bedeuten, daß das Packbyte n+1 mal zusammengefaßt wurde.
Das Spezialbyte und die beiden nachfolgenden Bytes a und n bedeuten, daß das Byte a n+1 mal zusammengefaßt wurde.
Alle anderen Bytes müssen so wie sie in der Datei stehen in das Bild übernommen werden.

# 2 Filme

Film-Dateien haben sich gegenüber FRACTAL Version 4.1 (und 4.3) nicht geändert.
Sie bestehen aus dem Header 'film', der (als 2 Byte Integer abgelegten) Anzahl verschiedener Bilder sowie der Anzahl der Bilder im Film. Es folgen die Anzeigeoptionen (eine SHOW_OPTS-Struktur), daran schließen sich die Informationen über die im Film enthaltenen Bilder an. Und zwar für jedes Bild Dateipfad, Name und Dateiextension, jeweils als nullterminierter String (unter diesen Dateinamen werden die Bilder beim Einladen eines Filmes gesucht). Deshalb ist es auch nötig, daß die Bilder beim Speichern des Filmes gespeichert sind, FRACTAL könnte sonst höchstens raten, wohin man die Bilder abspeichern wird. Dateipfade können (seit FRACTAL V4.3) relativ sein, sie beginnen dann nicht mit einem Laufwerk. In diesem Fall ist der Pfad relativ zum Verzeichnis der Filmdatei zu verstehen.

[36] die anderen Bits sind reserviert

Zuletzt folgt eine Liste mit der Reihenfolge der Bilder im Film, wobei sich die Nummern (1 Byte Integer) auf die Reihenfolge der Dateinamen beziehen.
Filmdateien dürfen nicht länger als 32000 Bytes sein.

# 3 Einstellungen

Die Einstellungsdateien will ich hier nicht im Detail erläutern, da sie sicher nicht sonderlich interessant sind.
Eventuell doch von Interesse ist allerdings der prinzipielle Aufbau dieser Dateien, die ja nicht nur die Voreinstellungen von CHAOS*ultd* selbst, sondern auch die der diversen Berechungsroutinen – letztere in nicht festgelegter Reihenfolge und womöglich wechselnder Anzahl.
Realisiert wird dies durch eine Block-Struktur. Jeder Block beginnt mit einem 12 Byte langen Header: die ersten 8 Byte erhalten eine Kennung (CHAOS*ultd* verwendet für seine Einstellungen `CHSultd5`, für die Berechnungsroutinen wird die gleiche Kennung verwendet wie in Bilddateien. Es folgt als Langwort (4 Byte) die Länge der folgenden Daten. Anschließend kommen die Einstellungsdaten sebler, deren Inhalt von den Berechnungsroutinen abhängt; die CHAOS*ultd*-Einstellungen bestehen aus einer `CHS_SET`-Struktur, die in der Headerdatei `XCOMMON.H` definiert ist.

# Teil VI
# die Schnittstelle für externe Routinen

Die folgenden Erläuterungen sind nur für Programmierer bestimmt, die eigene Berechnungsroutinen für Bilder irgendeiner Art schreiben wollen.
Aus technischen Gründen ist es nicht ratsam dies in einer anderen Sprache als C[37] und vermutlich auch da wiederum nur in Pure C (bzw. Turbo C) zu versuchen. Wie weit andere C-Compiler mit den Parameterübergabe-Konventionen von Turbo/Pure C zurecht kommen weiß ich nicht (vermutlich eher nicht, Turbo/Pure C übergibt Parameter auch in Registern).
Man sei sich auch darüber im Klaren, daß die eigentliche Berechnungsroutine (jedenfalls solange man sie nicht sehr aufwendig optimiert) meist nur einen kleinen Teil der nötigen Routinen darstellt, insbesondere sollte man in der Lage sein, mit GEM-Dialogboxen umzugehen. Ich setze im folgenden stets vorraus, daß Interessenten in C programmieren können, insbesondere den Umgang mit Strukturen beherrschen, ebenso die Programmierung von GEM-Programmen.
Wie aber werden externe Routinen überhaupt eingebunden? Und welches Format muß die erzeugte `XCH`-Datei haben?
Um die zweite Frage zuerst zu beantworten: bei der `XCH`-Datei handelt es sich um eine normale Programm-Datei, die aber keinen Startup-Code besitzt. Erzeugen kann man eine solche Datei mit Pure C, indem man in der Projektdatei einfach den Startup-Code wegläßt (man kann das so erzeugte Programm natürlich nicht vom Desktop aus ausführen).
Das Programm enthält auch keine Funktion `main`, eingebunden wird es vielmehr so, daß die erste Funktion der Source-Datei nach dem Laden und Relozieren aufgerufen wird (bei mehreren Sourcedateien ist es die erste Funktion der ersten Sourcedatei, es kommt nur darauf an, daß diese Funktion am Anfang des TEXT-Segmentes steht, was man gegebenenfalls mit einem Debugger überprüfen sollte). Diese Funktion hat nur zwei Aufgaben: erstens wird ihr von CHAOS*ultd* ein Zeiger auf eine Struktur mit Routinen und globalen Variablen übergeben, die CHAOS*ultd* den externen Routinen zur Verfügung stellt, und zweitens gibt die Routine an CHAOS*ultd* einen Zeiger auf eine Struktur zurück, in der die Adressen der externen Routinen stehen.
Das war's dann gewissermaßen schon, zum Initialisieren der externen Routinen wird speziell eine der Routinen aufgerufen.

## 1 Beschreibung der benötigten Routinen

Zunächst eine Vorbemerkung: ich bin kein sonderlich guter Programmierer und insbesondere programmiere ich zwar nicht konzeplos aber doch mit viel zu oft wechselnden Konzepton. Dies macht sich auch bei den bereitzustellenden Routinen bemerkbar, etwa dadurch, daß mal der Bildschirm vor Ausgabe einer Dialogbox schon gesichert ist, mal noch zu sichern ist. Auch Rückgabecodes sind nicht unbedingt einheitlich. Sorry, das ist sicher nicht sehr gelungen, aber die Schnittstelle ist halt nach und nach entstanden. Was bei den Routinen jeweils zu beachten ist wird aber im Folgenden ausführlich erläutert.

[37] in Frage käme höchstens noch Assembler, aber es erscheint mir übertrieben, die ganzen Routinen zur Parameter-Ein- und Ausgabe in Assembler zu schreiben

Alle Angaben und Routinen müssen in eine `CHAOS`-Struktur eingetragen werden, die dann bei der Initialisierung an CHAOS*ultd* übergeben wird (genaueres siehe unten).
Die nötigen Struktur-Deklarationen finden sich alle in der Headerdatei `XCOMMON.H`. CHAOS*ultd* gibt es bekanntlich in zwei Versionen, sw- und Farb-Version, diese haben einen gemeinsamen Quelltext, in dem jedoch mittels bedingter Compilierung zwischen den Versionen unterschieden wird. Für die Farb-Version ist das Makro `COLOR` definiert (mittels `#define COLOR`, oder einer Compileroption `-dCOLOR`), für die sw-Version nicht. Dies gilt auch für die `XCOMMON`-Headerdatei.

## 1.1 allgemeine Angaben

### id

```
char id[8];
```
Kennung der Routine, die auch in allen Bilddateien und für den Parameter-Block in der Einstellungsdatei verwendet wird.
Mein Vorschlag ist, die ersten vier Zeichen als Kennung für den Autor, die anderen vier als Kennung für die Routine zu verwenden (dementsprechend beginnen meine Kennungen auch alle mit TMMW).

### menue

```
char *menue;
```
Zeiger auf den Menüeintrag des Programmes (ohne vorangestellte Leerzeichen und Tastaturbefehl). Dieser darf maximal 16 Zeichen lang sein, er wird auch bei der Auflistung der Berechnungsroutinen im Programm-Info verwendet.

### name, version und text

```
char *name;
char *version;
char *text;
```
Diese drei Texte werden ausschließlich für das Programminfo verwendet. `name` sollte den Namen des Programmautors (max. 24 Zeichen), `version` eine Versionsnummer und ein Erstellungsdatum (max. 20 Zeichen) und `text` eine allgemeine Kurzbeschreibung (max. 64 Zeichen) enthalten (die Texte sind nicht optional, will man nichts angeben, so müssen die Zeiger mindestens auf einen Leerstring zeigen).

### icon und xicon

```
ICONBLK *icon; ICONBLK *xicon;
```
Zeiger auf Icon für die Anzeige der Bilder auf dem Desktop. `icon` wird für nicht gespeicherte Bilder, `xicon` für gespeicherte verwendet; Konvention ist, gespeicherte Bilder mit einem Haken zu kennzeichnen.
Die Icon müssen eine Größe von 48x24 Punkten haben, der Buchstabe des Icons wird für die Kennzeichnung der Bildgröße verwendet und sollte oben links liegen.

**set_len und set**

```
long set_len;
void *set;
```
`set_len` bezeichnet die Länge der Voreinstellungen, `set` zeigt auf diese (man muß also für die Voreinstellungen eine Struktur anlegen sowie eine globale Variable, auf deren Adresse man dann in `set` einträgt).

**par_len**

```
long par_len;
```
Länge der Bildparameter

**make_len**

```
long make_len;
```
Länge der Eingabeparameter für das Erzeugen neuer Bilder, im folgenden auch als Bild(er)parameter bezeichnet.

**neu_len**

```
long neu_len;
```
Länge der Parameter für `make_neu`.

**neu_par und neu_flag**

```
void *neu_par;
int neu_flag;
```
Flags für `make_neu`, siehe Beschreibung der `make_neu`-Routinen

**last_flag**

```
int last_flag;
```
Flag: falls `last_flag` 1 ist, so entspricht ein `lastline`-Wert größer Null in den Objekt-Parametern der letzten berechneten Bildschirmzeile. Ist `last_flag` 0, so wird immer der ganze Bildschirm abgelegt.

## 1.2 Routinen

Mit Ausnahme der Routinen `do_init`, `get_param`, `get_picpar` und `do_draw` sind alle Routinen optional, d.h. sie können fehlen, ohne daß dies zu Programm-Abstürzen führt. Diese drei Routinen stellen natürlich nur die Minimalanforderungen an die Berechnungsroutinen dar, mit ihnen kann man lediglich Bilder erzeugen.

**do_init**

```
int do_init(int nr);
```
Die Routine wird beim Programmstart aufgerufen. Sie sollte alle globalen Initialisierungen vornehmen, also etwa Resourcen laden usw.
Der Parameter `nr` ist die Typ-Nummer der Routine unter der diese intern verwaltet wird.

Zurückgeben muß die Routine 0, falls erfolgreich initialisiert wurde, konnte die Routine nicht initialisiert werden, so gibt man 1 zurück, die Berechnungsroutinen werden dann nicht aufgenommen.
Anmerkung: nicht kümmern muß sich die Routine um das Laden der Voreinstellung. Sie werden von CHAOS*ultd* selbst geladen. In der Initialisierungsroutine stehen sie allerdings noch *nicht* zur Verfügung.

**get_param**

`int get_param(void *param, X_PARAM *x_param);`
Diese Routine dient der Eingabe der Bildparameter für neu zu berechnende Bilder.
`param` ist ein Zeiger auf die Parameter für die Erzeugung neuer Bilder. Diese Parameter sind i.a. nicht direkt die Parameter eines Bildes, es soll ja auch die Erzeugung von Bildfolgen möglich sein. `x_param` ist ein Zeiger auf eine Strukur mit zusätzlichen, Bildtyp-unabhängigen Parametern, die die Routine `get_param` ausfüllen muß.

```
typedef struct
{
int anzahl; /* Anzahl der zu erzeugenden Bilder */
int size; /* Bildgröße (0 klein, 1 mittel, 2 groß) */
#ifndef COLOR
int color; /* 0 normal, 1 invers */
#else
int col_tab[16]; /* Farbtabelle */
#endif
int xor_offset;
char name[16];
char path[64];
} X_PARAM;
```

Das dürfte weitgehend klar sein. `xor_offset` ist der gleiche Wert wie auch in Bild-Dateien. Er wird in Byte angegeben. Wichtig ist, daß bei Berechnungsabbruch mindestens soviele Bilddaten vorliegen, wie `xor_offset` angibt, sonst kommt es zum Crash (das kann eigentlich nur vorkommen, wenn das Bild nur soweit berechnet gespeichert wird (also wenn `last_flag` 1 ist). `xor_offset` ist normalerweise ein Vielfaches der Zeilenlänge oder 0 (kein Xor beim Packen).
Die Routine braucht sich *nicht* um das Sichern und Restaurieren des Bildhintergrundes kümmern; Rückgabewerte sind 0 im Falle eines Abbruches und 1 für Ok, also für das Erzeugen der Bilder.

**get_picpar**

`void *get_picpar(void *param_in, int nr,int anzahl);`
Diese Routine muß aus den von `get_param` eingelesenen Parametern (für ein oder mehrere Bilder) die Parameter eines konkreten Bildes machen.
Übergeben wird ein Zeiger auf die Bild(er)parameter (`param_in`), die Nummer des aktuell zu erzeugenden Bildes (`nr`, gezählt wird von 0 an) und die Gesamtzahl der zu erzeugenden Bilder.

Zurückliefern muß die Routine die Parameter des Bildes Nummer `nr`. Dabei handelt es sich jetzt wirklich um die Parameter eines Bildes, wie sie dann auch von CHAOS*ultd* verwaltet werden sollen.
Im einfachsten Fall, wenn die Berechnungsroutinen nur die Erzeugung einzelner Bilder zuläßt, können die Bild(er)parameter natürlich mit den Bildparametern übereinstimmen, und diese Routine hat nichts weiter zu tun, als den Zeiger `param_in` wieder zurückzugeben. Läßt man die Berechnung mehrerer Bilder zu, so müssen die Parameter für das aktuelle Bild interpoliert werden. Für die Bildparameter mit den interpolierten Werten sollte man dabei in den Bild(er)parametern Platz bereithalten, da man sie natürlich nicht auf dem Stack ablegen kann und globale Variablen unnötig Speicherplatz brauchen.

**set_param**

```
void set_param(void);
```
Diese Routine ist zur Eingabe der Voreinstellungen der Parameter gedacht. Der Bildschirmhintergrund wird vor Aufruf gesichert und nachher restauriert.
Die Parameter müssen natürlich in dem in `set` angegebenen Speicherbereich verwaltet werden, wenn sie korrekt gespeichert werden sollen.

**extended**

```
void extended(void);
```
Diese Routine wird bei Anklicken des Menüpunktes der Berechnungsroutinen mit `CONTROL` aufgerufen. Hierfür ist keine besondere Funktion vorgesehen, man kann mit der Routine machen was man will.
Wie bei `set_param` wird der Bildschirm gerettet und restauriert.

**get_neu_param**

```
int get_neu_param(void *param);
```
Mit dieser Routine werden die Parameter für neu berechnen eingegeben. `param` ist ein Zeiger auf die zugehörigen Parameter, in den die Routine die eingelesenen Werte eintragen muß.
Die Routine muß darüberhinaus die Einträge `neu_par` und `neu_flag` setzen, in `neu_par` trägt man `param` ein (nur bei Ok), `neu_flag` setzt man auf 1 falls Ok gewählt wurde, -1 sonst. Daß diese Parameter von den lokalen Routinen zu setzen sind liegt daran, daß nur so die gemeinsamen Neu-Parameter der Fractal-Routinen zu realisieren waren.
Rückgabewert: 0 für Ok, 1 für Abbruch

**make_neu_param**

```
void *make_neu_param(FR_OBJC *objc, void *param, X_PARAM *x_par,
int *change, int *redraw, int *size);
```
Mit dieser Routine werden aus den Daten des neu zu berechnenden Objektes und den Neu-Parametern die neuen Daten ermittlet.
`objc` ist ein Zeiger auf das Objekt (die Objektparameter erhält man dann mittels `objc->par->data`), `param` ist ein Zeiger auf die Neu-Parameter, `x_par` einer auf die allg. Parameter des *neuen* Bildes (nötig um etwa neu Farben oder Größe einzutragen).

`change`, `redraw` und `size` sind Flags, um anzugeben, ob und wieweit die Parameter geändert wurden. Die Flags sind beim Aufruf von make_neu_param gelöscht (0) und müssen gegebenenfalls auf 1 gesetzt werden und zwar `change` wenn sich irgendwas an den Parametern geändert hat, `redraw` wenn durch eine Änderung die Neuberechnung des Bildes nötig ist (praktisch immer, nicht aber wenn nur die Farben geändert wurden) und `size` wenn sich die Bildgröße ändert.
Zurückgegeben wird ein Zeiger auf die neuen Bildparameter (Platz für diese reserviert man vernünftigerweise in den Neu-Parametern).

**make_vier**

```
void make_vier(FR_OBJC *in, void *out, int nr);
```
Mit dieser Routine berechnet man die Parameter für vier Bilder. `in` ist ein Zeiger auf das zu vergrößernde Objekt, `out` einer auf die Parameter des neuen Bildes, deren Speicher man ausnahmsweise nicht selber bereitstellen muß.
`nr` gibt an, welches der vier Viertelbilder gemeint ist, und zwar ist 0 links oben, 1 rechts oben, 2 links unten und 3 rechts unten.

**do_draw**

```
int do_draw(FR_OBJC *object, void *param, void *puffer);
```
`do_draw` ist die zentrale Berechnungsroutine.
Übergeben wird `object`, ein Zeiger auf die Objekt-Struktur, `param`, ein Zeiger auf die Parameter (eigentlich überflüssig, weil identisch `object->par->data`) und `puffer` ein Zeiger auf einen 32000 Byte großen (*nicht* 32kByte!) Puffer, mit dem man machen kann was man will (fast).
Die Routine muß schlicht das Bild auf den Bildschirm zeichnen, alles andere erledigt CHAOS*ultd*.
Zurückgeben muß die Routine den Tastaturstatus (`SHIFT, CONTROL etc.`) beim Abbruch (oder Null).

**dont_draw**

```
void dont_draw(FR_OBJC *object,void *param);
```
Diese Routine wird, falls existent, aufgerufen, falls ein neu erzeugtes Bild nicht gleich berechnet werden soll (etwa bei Berechnungsabbruch beim Erzeugen von Bildfolgen). Sie ist im allgemeinen überflüssig und muß auch nicht existiern, wird aber gebraucht, wenn man etwa eingegebene Parameter als zusätzliche Parameter verwalten will (das ermöglicht eine variable Länge). `object` und `param` wie bei `do_draw`.

**show_par**

```
int show_par(FR_OBJC *object, long *buf);
```
Diese Routine soll die Bildparameter des Objektes `object` anzeigen. `buf` ist ein Puffer, in den der Bildschirm bei Aufruf gerettet werden muß, der Bildschirm muß auch wieder restauriert werden. Dies geschiet am einfachsten mit den von CHAOS*ultd* zur Verfügung gestellten Funktionen `draw_objc` und `undraw_objc` bzw. `copy_screen` (vgl. unten). Man

darf *nicht* den Bildschirm auf `buf` umstellen, da `buf` nicht auf eine hinreichend gerade Adresse zeigen muß.
Als Rückgabewert wird eine 0 erwartet, falls sich die Parameter sicher nicht geändert haben (Name oder Farben kann man unter Umständen hier ändern), sonst eine 1.

**show_info**

```
void show_info(FR_OBJC *object, long *buf);
```
Die Funktion `show_info` dient der Anzeige eines Bildinfos, abgesehen vom (nicht vorhandenen) Rückgabewert gilt das gleiche wie bei `show_par`.

**get_info**

```
void get_info(FR_OBJC *object, PIC_INFO *info);
```
Mit `get_info` holt sich CHAOS*ultd* die Informationen die es zur Anzeige ein Bildinfos über mehrere Bilder benötigt (Rechenzeit, Zahl der Iterationen und der Punkte). Existiert die Funktion `show_info` nicht, so wird `get_info` – falls es existiert – auch für einzelne Bilder verwendet.

**get_objcblk**

```
void get_objcblk(FR_OBJC *object, double *x_min, double *x_max, double
*y_min, double *y_max);
```
Diese Routine gibt an CHAOS*ultd* die Koordinaten des Bildrechteckes eines Objektes zurück, natürlich nur wenn es solche gibt.
`object` ist ein Zeiger auf das Objekt, `x_min`, `x_max`, `y_min` und `y_max` sind Zeiger auf die Koordiaten.
Falls dies Routine existiert, nimmt CHAOS*ultd* auch an, daß der Bildschirm direkt ein Koordinatensystem darstellt, d.h. daß man die Bild-Koordinaten eines Raster-Punktes durch Interpolation zwischen den Koordinaten des Bildrechteckes ermitteln kann[38]. Nur in diesem Fall funktioniert die Koordinatenanzeige von Bilder zeigen oder das Anzeigen eines Blockes.

**get_xobjcblk**

```
void get_xobjcblk(FR_OBJC *object, double *x_min, double *x_max, double
*y_min, double *y_max);
```
Die Routine entspricht exakt der Routine `get_objcblk`, nur daß keine Folgerungen aus ihrer Existenz gezogen werden.

**get_pkt**

```
void get_pkt(FR_OBJC *object, double *x, double *y);
```
Mit der Routine `get_pkt` holt sich CHAOS*ultd* die Koordinaten für den Punkt, der bei Konstante anzeigen in Bilder zeigen angezeigt wird. Die Routine macht nur Sinn, wenn auch die Routine `get_objcblk` existiert.

[38] nicht der Fall ist dies z.B. bei 3d Fractalen

**show_pktinfo**

```
void show_pktinfo(FR_OBJC *object, int x, int y, long *scr, long *buf);
```
Diese Routine wird für die Anzeige eine Punktinfos aufgerufen.
`object` ist das betroffene Bildobjekt, `x` und `y` sind die (Bildschirm)-Koordinaten des Punktes, `scr` ist ein Zeiger auf den Bildschirm, in dem das Bild liegt, und `buf` ist ein Puffer um den Hintergrund zu retten.

**dr_param**

```
void dr_param(FR_OBJC *object);
```
Die Hardcopyroutine erlaubt es Bildparameter unter die Hardcopy zu drucken. Dazu wird diese Funktion aufgerufen. `object` ist das ausgedruckte Objekt.
Die Parameter sollten sich auf vier Zeilen beschränken, damit zwei große Bilder auf eine Druckerseite (11 Zoll) gehen.

**cnv_load**

```
int cnv_load(int frac_typ, L_PARAM *par, PIC_DATA *xparam, PIC_DATA *param,
PIC_DATA **x_par);
```
Diese Routine dient dazu, beim Laden die Parameter von Fraktalen anderer Auflösung (oder auch einer anderen Version der Berechnungsroutinen) anzupassen.
Übergeben wird der Bildtyp (für den Fall, daß mehrere Bildtypen zusammengefaßt sind) in `frac_typ`, ein Zeiger auf zusätzliche Parameter in `par` mit Größe, Farben, xor-Offset etc. (vgl. Header-Datei) ein Zeiger auf die Parameter der Datei `xparam`, ein Zeiger auf die Parameter des Bildes `param`. `PIC_DATA *x` ist ein Zeiger auf einen Block der internen Speicherverwaltung, mit `x->len` erhält man die Länge des Blockes + 8, `x->data` referenziert den Blockinhalt.
Die geladenen Parameter müssen nach `param` kopiert werden, sie müssen ja nicht die gleiche Länge wie die Bildparameter haben (die Länge erhält man wie gesagt durch `xparam->len-8`).
Außerdem erhält man einen Zeiger auf einen Zeiger auf die (geladenen) zusätzlichen Parameter `x_par`. Möchte man letztere freigeben, so kann man dies mit `pic_mfree(*x_par)` tun, muß dann aber auch `*x_par` auf 0 setzen.
Die Funktion kann 1 zurückgeben, wenn sie die Parameter nicht anpassen kann, dann wird das Laden abgebrochen. Ansonsten muß 0 zurückgegeben werden.

**reserved**

Weitere achte Einträge (4 Byte) sind reserviert für mögliche Erweiterungen. Sie müssen auf 0 gesetzt werden; falls es Erweiterungen geben wird, werden diese natürlich optional sein.

# 2 Beschreibung der von CHAOS*ultd* zur Verfügung gestellten Routinen

Die von CHAOS*ultd* zur Verfügung gestellten Routinen und Variablen sind in der `COMMON`-Struktur zusammengefaßt. Ein Zeiger auf diese Struktur wird beim ersten Aufruf der Rou-

tinen unmittelbar nach dem Laden als Parameter übergeben, die Routine kann ihn dann irgendwo sichern.

## 2.1 Routinen

CHAOS*ultd* stellt den externen Routinen die folgenden Funktionen zur Verfügung:

### draw_objc

```
void draw_objc(OBJECT *object, long *buffer);
```
Die Funktion `draw_objc` zeichnet die Dialogbox, auf die der Zeiger `object` zeigt. `buffer` ist ein Zeiger auf einen 32000 Byte großen Speicherbereich, in den vorher der Bildschirmhintergrund kopiert wird. Ist `buffer` 0, so wird der Hintergrund nicht gesichert[39].

### xdraw_objc

```
void xdraw_objc(long *buf);
```
`xdraw_objc` sichert lediglich den Bildschirm in den Puffer auf den `buf` zeigt (32000 Byte). Ist `buf` 0, so geschieht nichts (auch kein Absturz).

### undraw_objc

```
void undraw_objc(long *buf);
```
`undraw_objc` entfernt eine gezeichnete Dialogbox, indem es den Hintergrund aus dem Puffer `buf` zurückkopiert. Dies funktioniert natürlich nur, wenn man den Hintergrund auch in den Puffer gerettet hat.

Anmerkung: die Funktionen `draw_objc`, `xdraw_objc` und `undraw_objc` schalten die Maus ab und wieder an, darum braucht man sich also nicht zu kümmern. Man sollte beim Kopieren von Bildschirmen deswegen auch diese Funktionen und nicht die unten genannte Funktion `copy_screen` verwenden, die solches nicht erledigt.
Des weiteren sei noch angemerkt, daß die Funktionen unter Bildschirm stets den logischen Bildschirm verstehen, also den, der sich via `Logbase()` ermitteln läßt.

### write_dbl

```
void write_dbl(char *s, double zahl,int l,int p);
```
Die Funktion `write_dbl` dient zur Ausgabe einer Fließkommazahl `zahl` in einen String `s`. `l` gibt die gewünschte maximale Länge, `p` die Zahl der Nachkommastellen an.
Die Zahl wird zunächst mit `sprintf(str,‘‘%*.*lf‘‘,l,p,zahl);` in einen Zwischenpuffer ausgegeben, der 128 Zeichen lang ist (sollte der String länger werden so ist mit Systemabsturz zu rechnen).
Anschließend werden abschließende Nullen entfernt (z.B. `1.2000` wird zu `1.2`), lediglich *eine* direkt auf den Dezimalpunkt folgende Null bleibt stehen; der String wird dann mit maximal `l` Zeichen nach `s` kopiert.

[39] der Zeiger `buffer` darf natürlich nicht auf eine ungerade Adresse zeigen, bei den folgenden Routinen sei solches jeweils a priori angenommen, von CHAOS*ultd* zur Verfügung gestellte Puffer erfüllen dies Bedingung natürlich; genauso muß `object` auch wirklich ein Zeiger auf eine Dialogbox sein, andernfalls ist mit dem Schlimsten zu rechnen

Erklärung: bei der Ausgabe einer Fließkommazahl mit `printf` kann es erstens passieren daß der String länger als die angegebene Länge wird, was sich in Dialogboxen verheerend auswirken kann, zweitens werden abschließende Nullen ausgegeben, die die Lesbarkeit der Zahlen nicht gerade erhöhen. Beides versucht zu Routine zu vermeiden, wobei im ersten Fall in Kauf genommen wird, daß die ausgegebene Zahl womöglich verstümmelt wird.
Eine Ausgabe in wissenschaftlicher Darstellung, d.h. mit Zehnerexponent, ist nicht vorgesehen, da alle bisher von mir realisierten Routinen mit Zahlen rechnen, die nicht in Bereiche kommen, wo dies sinnvoll wäre.

### ipol, xipol

```
double ipol(double anf, double end, int act_nr, int max_nr, int mode);
void xipol(double anf1, double end1, double anf2, double end2,
double *anf,double *end,int act_nr,int max_nr);
```

`ipol` dient zur Interpolation zwischen `anf` und `end`. `act_nr` stellt die Nummer des gewünschten Wertes, `max_nr` die maximale Nummer dar (da die Nummern von 0 gezählt werden, ist `max_nr` gleich der Anzahl der Nummern-1). `mode` bezeichnet die Art der Interpolation:

| mode | |
|---|---|
| 0 | degressive Interpolation, die Schrittweite nimmt ab |
| 1 | lineare Interpolation, die Schrittweite ist konstant |
| 2 | progressive Interpolation, die Schrittweite nimmt zu |
| 3 | Zufallszahl zwischen anf und end (keine Interpolation) |

`xipol` interpoliert ebenfalls und zwar mit Wertepaaren. Dabei ändert sich die Differenz zwischen den beiden Werten pro Schritt um einen konstanten *Faktor*. `anf1` und `end1` sowie `anf2` und `end2` sind Anfangs- und Endwerte für die beiden Werte. In `anf` und `end` werden die interpolierten Werte zurückgegeben. `act_nr` und `max_nr` sind wie oben zu verstehen.
Die `xipol`-Routine ermöglicht das gleichmäßige Vergrößern von Blöcken. Interpoliert man dabei die Koordinaten linear, so ergibt sich am Ende ein größerer Vergrößerungs*faktor*, da die absolute Änderung konstant ist. Durch die degressive Interpolation wird dies zwar abgeschwächt, aber nicht wirklich behoben. Die `xipol`-Routine berechnet die Zwischenwerte dagegen so, daß sich ein konstanter Vergrößerungsfaktor ergibt.

### plot_pixel

```
void plot_pixel(int x, int y,void *addr);
void plot_pixel(int x, int y,int col,void *addr);
```

Die Routine `plot_pixel` zeichnet einen Punkt. Die erste Deklaration gilt für die sw-Version, die zweite für die Farbversion.
`x` und `y` beschreiben die Koordinaten des Punktes (in der üblichen Weise, die linke obere Bildschirmecke ist (0,0)). `addr` ist ein Zeiger auf den Bildschirmspeicher, in dem der Punkt gesetzt werden soll.
`col` bezeichnet die Farbe, die der Punkt erhalten soll.
In der sw-Version werden lediglich Punkte gesetzt, Löschen ist nicht möglich, deshalb wird auch keine Farbe benötigt[40].

[40] in der Farbversion kann man durch `col`=0 Punkte auch löschen

Die unterschiedliche Deklaration der Funktion in den beiden Modi ist ein bißchen lästig, man kann sie aber durch Definition eines Makros unterdrücken:
`#define pl_pixel(x,y,col,addr) plot_pixel(x,y,addr)`
in der sw-Version bzw. in der Farb-Version:
`#define pl_pixel(x,y,col,addr) plot_pixel(x,y,col,addr)`
So kann man in beiden Fällen eine Farbe angeben, sollte natürlich immer bedenken, daß diese in der sw-Version ignoriert wird.
Alternativ könnte man die Deklaration für die Farb-Version auch für die sw-Version verwenden, dann wird eben ein zusätzlicher Parameter (in D2) übergeben, den die aufgerufene Funktion schlicht ignoriert.

### set_point

```
void set_point(int x,int y,int col,int flag);
```
`set_point` zeichnet ebenfalls Punkte. Diese Funktion darf *nur* beim Zeichnen von Bildern (also in der Routine `do_draw`) aufgerufen werden. Sonstiger Gebrauch wird mit Absturz nicht unter zwei Bomben geahndet!
`x` und `y` sind wieder die Bildschirmkoordinaten des Punktes, `col` seine Farbe. `flag` ist in der Farb-Version redundant, in der sw-Version wird mittels `flag` zwischen Einzelpixeln und 4er-Gruppen unterschieden. Ist `flag 0` so wird ein einzelnes Pixel gesetzt (nicht gelöscht, `col` hat dann wieder keine Bedeutung). Ist `flag 1` so wird ein aus vier Pixeln ((x,y), (x+1,y), (x,y+1) und (x+1,y+1)) bestehender Bildpunkt in der Graustufe `col` gesetzt. `col` kann die Werte 0 (weiß), 1 (ein Punkt), 2 (zwei diagonal liegende Punkte), 3 (ein Punkt nicht) und 4 (schwarz) annehmen. Auch hier werden grundsätzlich Punkte nur gesetzt!

### get_pixel

```
int get_pixel(int x,int y,void *addr);
```
`get_pixel` liefert die Farbe des Bildpunktes (`x`, `y`) im Bildschirm `addr`

### copy_screen

```
void copy_screen(long *s,long *d);
```
`copy_screen` kopiert (ziemlich schnell) 32000 Byte von `s` (wie Source) nach `d` (wie Destination). Da dies genau die Länge eines Bildschirmes ist (Overscan geht ja nicht) erklärt sich wohl auch der Name.

### clr_screen

```
void clr_screen(long *s);
```
`clr_screen` löscht 32000 Byte ab `s`.

### pic_malloc, pic_mfree und check_mem

```
PIC_DATA *pic_malloc(long len,PIC_DATA **mother);
void pic_mfree(PIC_DATA *addr);
int check_mem(long len);
```

Diese Funktionen erlauben der Zugriff auf die interne Speicherverwaltung, der aber nicht ganz unproblematisch ist.
Das Problem besteht vor allem in der Belegung von Speicher durch die `do_draw`-Routine (etwa als zusätzliche Parameter), da sichergestellt sein muß, daß für das Ablegen des berechneten Bildes garantiert genug Speicher übrigbleibt.
Ich möchte an dieser Stelle nicht weiter auf diese Funktionen eingehen, da man meist auch ohne sie auskommen dürfte.
Falls die Berechnungsroutinen generell zusätzlichen Speicher brauchen, so sollten sie ihn in der `do_init`-Routine vom TOS (also mit `Malloc`) anfordern.

### gettime

```
long gettime(void);
```
Die Funktion `gettime` liefert den akutellen Wert des 200 Hz Zählers (also den Wert der Systemvariablen an der Adresse $4BA).

### get_objc, get_obblk, get_konst und get_block

```
FR_OBJC *get_objc(int typflag);
int xget_obblk(FR_OBJC *objc, double *x_min, double *x_max,
double *y_min, double *y_max);
FR_OBJC *get_konst(int typflag, double *x, double *y);
FR_OBJC *get_block(int typflag, double *x_min, double *x_max,
double *y_min, double *y_max);
```
Diese Funktionen ermöglichen die Übernahme von Parametern bestehender Bilder bei der Eingabe von Parametern.
Die Funktion `get_objc` liefert einen Zeiger auf ein Objekt, das der Anwender auf dem Desktop auswählen kann, zurück (oder 0, falls kein Objekt gewählt wurde). `typflag` gibt an, welche Objekte gewählt werden können: ist `typflag` -1, so können beliebige Fractale, -2 beliebige Bilder (nicht Filme) gewählt werden. Ansonsten kann noch ein positiver Wert (größer 1) angegeben werden, dann können nur Fractale dieses Types ausgewählt werden (normalerweise ist das dann natürlich die Typnummer der eigenen Routinen, wie man sie beim Initialiseren übergeben bekommt). Der Bildschirmaufbau der Eingabefunktion wird von dieser Routine *nicht* verändert.
Die Funktion `get_obblk` liefert – falls möglich – in den Rückgabeparametern `x_min`, `x_max`, `y_min` und `y_max` das Koordinatenrechteck des Objektes `objc` zurück. Als Rückgabewert wird 0 geliefert, wenn kein Rechteck ermittelt werden konnte, 1 bedeutet, daß die Werte in `x_min`, `x_max`, `y_min` und `y_max` gültig sind.
Mit der Funktion `get_konst` kann man den Benutzer eine Koordinate in einem Bild, mit `get_block` einen Ausschnitt wählen lassen. Rückgabewerte sind `x` und `y` bzw. `x_min`, `x_max`, `y_min` und `y_max` sowie (als dierektes Funktionsergebnis) ein Zeiger auf des Objekt in dem die Koordinate bzw. der Ausschnitt liegt.
Ein Rückgabewert von 0 bedeutet, daß kein Objekt ausgewählt wurde (oder im ausgewählten Objekt kein Block/keine Konstante bestimmt werden kann); in diesem Fall ist der Bildschirmaufbau der Eingabefunktion *nicht* verändert. Ein Rückgabewert von -1 bedeutet einen Abbruch der Auswahl beim Markieren der Konstante/des Blockes. In diesem Fall wurde das Bild schon angezeigt, so daß der Bildschirm der Eingabefunktion überschrieben

wurde; dargestellt wird dann wieder der CHAOS*ultd*-Desktop und man muß die Dialogbox neu zeichnen lassen.
`typflag` wird wie bei `get_objc` ausgewertet (-2 ist natürlich nicht sehr sinnig, da in Bildern ohne Parameter natürlich keine Koordinaten zur Verfügung stehen).

#### fsel

```
int fsel(char *path,char *sel,int *button,char *label);
```
Aufruf der Dateiselectorbox entsprechend `fsel_exinput`, allerdings wird bei älteren AES-Versionen automatisch (unter Unterdrückung von `label`) `fsel_input` verwendet.

#### dr_string

```
void dr_string(char *str);
```
Ausgabe eines Strings `str` auf den Drucker entsprechend der gewählten Ausgabefunktion. Zum Drucken der Parameter unter eine Hardcopy ist diese Funktion vernünftigerweise zu verwenden, da man sich dann nicht selber um den eingestellen Ausgabemodus kümmern muß.

#### set_unoutlined

```
void set_unoutlined(FR_OBJC *object);
```
Die Funktion `set_unoutlined` setzt für das Objekt `object` den Status nicht fertig, etwa wenn man (wie bei Hüpfer möglich) nachträglich die Zahl der gewünschten Iterationen erhöht.

#### irand und xrand

```
long irand(long dummy);
double xrand(long dummy);
```
Zufallszahlengenerator nach D. Knuth. `irand` liefert eine (Long-)Integer Zufallszahl zwischen 0 und 1000 000 000, `xrand` eine Fließkomma-Zufallszahl zwischen 0 und 1. `dummy` dient der Initialisierung: ist `dummy` ungleich Null, so wird der Zufallszahlengenerator (für beide Routinen gemeinsam!) initialisiert und zwar mit einem zufälligen Wert (aus dem xbios-Zufallszahlengenerator) falls `dummy` -1 ist, mit `dummy` selbst sonst.
Man beachte, daß die Zufallszahlen nach der Initialisierung mit einem bestimmten Wert natürlich deterministisch und wiederholbar sind, initialisieren sollte man den Zufallszahlengenerator auch nur einmal am Anfang einer Routine (wenn man ihn bei jedem Aufruf auf einen festen Wert initialisiert, dann wird er auch immer den gleichen Wert liefern!).

#### conv_time

```
void conv_time(unsigned long ms, int *h, int *m, double *s);
```
Konvertiert die in Millisekunden angegebene Zeit `ms` in Stunden `h`, Minuten `m` und Sekunden `s`. Man beachte, daß der letzte Wert als Fliekommazahl zurückgegeben wird.

#### conv_sc_mm und conv_mm_sc

```
void conv_sc_mm(double scale, double offset, int delta, double *min, double
*max);
void conv_mm_sc(double min, double max, int delta, double *scale, double
*offset);
```

Mit diesen Routinen kann man zwischen der Definition von Bildschirmkoordinaten durch Nullpunkt und Vergrößerung (offset/scale) und durch minimalen und maximalen Achsenabschnitt umrechnen. `delta` gibt die *halbe* Höhe bzw. Breite des gesamten Bildschirms an ($\pm$ `delta` ist also die maximale und minimale Koordinate bei `scale` 1 und `offset` 0.

### Spezielle Funktionen für die Farb-Version

Die folgenden Funktionen existieren nur in der Farb-Version:

#### make_alert, make_drawobjc und make_undraw

```
int make_alert(int button,int *coltab,char *alert);
void make_drawobjc(OBJECT *objc);
void make_undraw(int *coltab);
```

Mit diesen Funktionen kann *während des Berechnens* eines Bildes eine Alert- bzw. Dialogbox ausgegeben werden. Das Problem besteht dabei im Auflösungswechsel, da für die Berechnung in niedrige Auflösung geschaltet wurde, für die AES-Ausgabe aber mittlere Auflösung gebraucht wird.
`make_alert` gibt eine Alertbox aus. `button` und `alert` entsprechen den Parametern von `form_alert`, `coltab` ist ein Zeiger auf die Farbtabelle des Bildes (üblicherweise die, die auch in der `FR_OBJC`-Struktur steht). Zurückgegeben wird der Rückgabewert von `form_alert` also der ausgewählte Button.
`make_drawobjc` und `make_undraw` sind die Analoga zu `draw_objc` und `undraw_objc`, wobei ein Puffer nicht angegeben werden muß. Für `make_undraw` muß man wie bei `make_alert` die Farbtabelle angeben. Die Dialogbox wird bei `make_drawobjc` nur gezeichnet, nicht aufgerufen (mit `form_do`, dies bleibt der externen Routine vorbehalten.
**Achtung!** Die Routinen benötigen als Zwischenspeicher den Puffer, der auch der Zeichenroutinen `do_draw` zur Verfügung steht. Der Inhalt dieses Puffers geht deshalb beim Aufruf einer dieser Routine verloren!
Zur Ausgabe von Dialogboxen beim Anzeigen von Parametern etc. darf man diese Funktionen nicht verwenden, dort man gibt einfach die Dialogbox (z.B. mit `draw_objc`) aus.

#### set_colors

```
void set_colors(int *col_tab,long *scrn,FR_OBJC *object,int flag);
```

Die Funktion `set_colors` dient zur Einstellung der Farbtabelle. Wird sie von externen Routinen aufgerufen, so muß der Parameter `scrn` 0 sein, `flag` dagegen 1.
`col_tab` zeigt natürlich auf die einzustellende Farbtabelle, bleibt noch `object`. `object` ist 0 oder ein Zeiger auf ein Bild-Objekt; im letzteren Fall wird im oberen Bildteil bei der Farbeinstellung das Bild, sonst einfach Farbbalken dargestellt. Läßt man die Farben eines bestehenden Bildes ändern, so sollte man natürlich einen Zeiger auf dieses Bild übergeben.

## 2.2 Variable

CHAOS*ultd* stellt noch eine Reihe von Variablen mit verschiedenen Daten zur Verfügung. Alle diese Werte sind ausschließlich *read-only* - Änderungen darf man *nicht* vornehmen.

### fr_x0, fr_y0, fr_dx, fr_dy und pic_size

```
int *fr_x0;
int *fr_y0;
int *fr_dx;
int *fr_dy;
int *pic_size;
int *line_len;
```

Integer-Arrays mit jeweils drei Einträgen für die drei Bildgrößen 0 (klein), 1 (mittel) und 2 (groß).
`fr_x0` und `fr_y0` enthalten die Koordinaten der linken oberen Ecke, `fr_dx` und `fr_dy` die Breite bzw. Höhe der Bilder in Pixeln *minus 1*!
`pic_size` enthält die Länge (ungepackter) Bilddaten in Byte, `line_len` ist die Länge einer Zeile, ebenfalls in Byte und zwar pro Bildebene.
Die Arrays enthalten die folgenden Werte:

```
#ifdef COLOR
int fr_x0[3]={80,48,0};
int fr_y0[3]={50,30,0};
int fr_dx[3]={159,223,319};
int fr_dy[3]={99,139,199};
int line_len[3]={20,28,40};
#else
int fr_x0[3]={160,96,0};
int fr_y0[3]={100,60,0};
int fr_dx[3]={319,447,639};
int fr_dy[3]={199,279,399};
int line_len[3]={40,56,80};
#endif

int pic_size[]={8000,15680,32000};
```

### scr

```
long *scr;
```

`scr` ist nur gültig während des Zeichnens von Bildern. Dann zeigt `scr` auf den Bildschirm, und kann (und sollte) für `plot_pixel`-Aufrufe verwendet werden.

### settings

```
CHS_SET *settings;
```

`settings` ist ein Zeiger auf die Einstellungen von CHAOS*ultd*. Die `CHS_SET`-Struktur ist in der Headerdatei definiert.

**reserved**

Auch in dieser Struktur ist Platz für Erweiterungen gelassen, hier sogar 16 Lang-Worte, die alle auf Null gesetzt sind.

# 3 Vorgehensweise

Zunächst einmal kann man sich, nachdem man diese Beschreibung gelesen hat, die Beispielroutinen für Feigenbaumdiagramme anschauen.
Möchte man eigene Routinen verwirklichen, so sollte man sich als erstes die nötigen Parameter überlegen, und eine entsprechende Struktur für die Bildparameter definieren, ebenso eine für die Eingabeparameter.
Dann muß man die Eingaberoutinen schreiben (natürlich nicht unbedingt gleich mit allen Details) und die Routine zur Erzeugung des Bildparameter aus den Eingabeparametern; das Laden der nötigen Resourcedatei erledigt man in der `do_init`-Routine.
Hat man dies geschafft, so fehlt nur noch das Herz aller Berechnungsroutinen, nämlich die eigentliche Zeichen-Routine.
Hat man alle diese Routinen mehr oder weniger fertig, so erstellt man eine `CHAOS`-Struktur, in die man alle Routinen und anderen Daten einträgt; für noch nicht existierende Routinen trägt man 0 ein.
Anmerkung: in der `CHAOS`-Struktur tauchen jede Menge Zeiger auf `void` auf, die man in Deklarationen konkreter Berechnungsroutinen natürlich besser durch Zeiger auf die jeweiligen Strukturen ersetzt. Die Warnungen vor verdächtigen Zeigerumwandlungen, die dann bei Eintrag der Routinen in eine `CHAOS`-Struktur auftreten, kann man beispielsweise durch das Präprozessorkommando `#pragma warn -sus` unterdrücken (falls man die Warnungen nicht in den Compiler-Optionen eh abgeschaltet hat).
Um die `CHAOS`-Struktur nun bei der Initialisierung an CHAOS*ultd* zurückzugeben braucht man noch eine `XCHAOS`-Struktur, die aus Sicherheitsgründen und für die Übergabe mehrerer `CHAOS`-Strukturen nötig ist.
Die `XCHAOS`-Struktur besteht aus einer 8 Byte langen Id in die man "CHSultd5" einträgt, der Anzahl der Routinen (`anz`) sowie einem Zeiger auf ein Array mit Integer-Flags (`flag`) und einem Zeiger auf ein Array mit `CHAOS`-Strukturen (`chs`)[41]. In den Flags muß das unterste Bit gesetzt sein, falls die Routine sw-tauglich ist, das 2. Bit steht für Farbtauglichkeit (läuft die Routine in beiden Auflösungen, so setzt man eben beide Bits); die anderen Bits sind reserviert und auf Null zu setzen).
Hat man dies geschafft, so schreibt man noch die Übergaberoutine für die Parameter (analog der der Feigenbaum-Diagramme); dann kann man sein Programm übersetzen und via CHAOS*ultd* aufrufen.
Beim Linken kann und sollte man übrigens die Stack-Größe auf Null setzen, da ohnedies der Stack von CHAOS*ultd* verwendet wird; ein eigener Stack wäre nur Speicherplatzverschwendung. Der Stack von CHAOS*ultd* ist übrigens 8 kByte groß und dürfte bei allen externen Funktionen problemlos mit bis zu 6 kByte belastbar sein.
Ein echtes Problem ist das Debuggen, der Source-Level-Debugger hilft einem so gut wie garnicht und auch Low-Level-Debugging geht nur ohne Label, also praktisch auch nicht.

[41] es kann natürlich auch jeweils nur ein Wert vorhanden sein, dann ist `anz` halt 1 und `flag` zeigt auf einen Integer-Wert, `chs` auf eine `CHAOS`-Struktur

Wer hier Probleme hat, kann sich ja mal mit mir in Verbindung setzen, eventuell stelle ich auch die Sourcen von CHAOS*ultd* zur Verfügung, so daß die Routinen als Bestandteil des Programmes entwickelt und so auch anständig debuggt werden können. Eine generelle Freigabe des Sourcecodes kommt aber nicht in Frage, dieser Hinweis steht nicht unbeabsichtigt an wenig herausragender Stelle.

### Weitere Hinweise

An Algorithmen kommen im Prinzip alle Algorithmen in Frage, die Bilder um ihrer selbst willen erzeugen – natürlich auch wenn sie nicht chaotisch sind[42].
Dabei sollte der Algorithmus aber auch eine gewisse Bandbreite an verschiedenen Bildern bieten, Algorithmen, die die Erzeugung genau eines Bildes ermöglichen, sind ja vielleicht ganz nett, in CHAOS*ultd* aber fehl am Platze, es ist in solchen Fällen allemal einfacher, ein Programm zu schreiben, daß ein solches Bild berechnet und auf den Bildschirm ausgibt, so daß man es mit einem Screen-Dump-Programm abspeichern kann.
CHAOS*ultd* ist dazu gedacht, viele Bilder zu berechnen; das Programm unterstützt durch Berechnungsabbruch und -fortsetzen insbesondere das Berechnen von Bildern über Nacht oder wenn man sonst nicht da ist[43].
Deswegen sollte man dem Benutzer bei der Wahl der Parameter freie Hand lassen und so möglichst viel Variationen zulassen.
Die Routinen sollten auch, so wie die vorhandenen, die Berechnung von Bildfolgen ermöglichen.
In jedem Fall sind für die Parametereingabe GEM-Dialoge zu verwenden. Man sollte dabei darauf achten, daß alle Parameter in beliebiger Reihenfolge angegeben werden können, braucht man mehr als eine Dialogbox, so sollte man zwischen diesen hin- und herschalten können, bzw. die zweite Dialogbox wäre als Unterdialog (wie beim Farben einstellen für Fractale) zu realisieren[44].
Soweit irgendmöglich sollten die Routinen den Benutzer bei der Eingabe der Parameter unterstützen, insbesondere die Übernahme von Parametern aus bestehenden Objekten sollte möglich sein.
Sehr viele der Routinen sind optional, das heißt aber nicht, daß externe Routinen auf sie alle verzichten können. Eine Option sollte nur dann weggelassen werden, wenn sie sich für den implementierten Bildtyp nicht, praktisch nicht oder nur mit unverhältnismäßig großem Aufwand realisieren läßt.
Ich denke die von mir bisher geschriebenen Routinen erfüllen diese Forderungen recht gut – am wenigsten die Feigenbaum-Routinen, die sind aber auch vor allem als Beispiel zur Einbindung externer Berechnungsroutinen entstanden.
Generell sollte man sich diese Routinen zum Vorbild nehmen, nicht nur weil ich sie für ganz brauchbar halte, sondern auch, weil Einheitlichkeit für ein Programm wie CHAOS*ultd* umso wichtiger ist, je mehr Routinen es gibt (vermutlich schreibt ja eh keiner welche und ich könnte mir das ganze gelabere sparen).
Zur Schnittstelle gibt es noch zu sagen, daß sie höchstens erweitert wird, nicht aber verändert, es sei denn, irgendeine unvorhergesehene Katastrophe tritt auf. Für CHAOS-

[42] nicht geeignet ist CHAOS*ultd* allerdings für bewegte Bilder (etwa zelluläre Automaten)

[43] ein Punkt den externe Routinen auf jeden Fall unterstützen sollten, wenngleich dies für manche Berechnungsmethoden möglicherweise nicht realisieren läßt

[44] die Eingabe der Berechnungsformel für freidef. Fract. natürlich ein Verstoß gegen diese Regel

*ultd* V5.0 entwickelte exteren Routinen werden somit auch (ohne Änderungen) mit allen weiteren Version von CHAOS*ultd* V5.x zusammenarbeiten.
Dies gilt allerdings nicht für eine etwaige GEM-Version. Da ich dafür das Programm mehr oder weniger neu entwickeln müßte (weshalb es eine GEM-Version auch nicht so bald geben wird) wird auch die Schnittstelle neu definiert werden.