R E M I S

Programmier – Handbuch

Stand: 08.08.86

Herausgeber:

Gesellschaft für Mathematik und Datenverarbeitung mbH

Postfach 1240, Schloß Birlinghoven
D – 5205 Sankt Augustin 1
Telefon(02241) 14 – 1, Telex 8 89 469 gmd d
Telefax(02241) 14 28 89, BTX *43900#
Teletex 2627 – 224135 = GMDVV

Autoren:

Udo Schneider

Unter Mitarbeit von:

Hilmar von dem Bussche

Texterstellung:

Dieser Text wurde mit der EUMEL – Textverarbeitung erstellt und aufbereitet und mit dem Agfa Laserdrucksystem P400 gedruckt.

Umschlaggestaltung:

Hannelotte Wecken

Hinweis:

Diese Dokumentation wurde mit größtmöglicher Sorgfalt erstellt. Dennoch wird für die Korrektheit und Vollständigkeit der gemachten Angaben keine Gewähr übernommen. Bei vermuteten Fehlern der Software oder der Dokumentation bitten wir um baldige Meldung, damit eine Korrektur möglichst rasch erfolgen kann. Anregungen und Kritik sind jederzeit willkommen.

1. Überblick 3

2. entfällt 13

3. Einrichten einer Datenbank 16
   3.1 Überblick 16
   3.2 Datenbeschreibung 17
   3.3 Formulardefinition 21

4. Programmieren 23
   4.1 Überblick 23
   4.2 Datenbankfunktionen 25
   4.3 Anwendung von Formularen 29

5. Datenbankverwaltung 38

6. Menue – Behandlung 40

7. Abfragesprache sql 43

8. Weitere Funktionen 45

9. Mehrfachbenutzung 47
   9.1 Überblick 47
   9.2 REMIS starten 48
   9.3 Bearbeiten der Datenbasis 50
   9.4 Programmieren von Transaktionen 52
   9.5 Datenbasis archivieren 54

10. Anhang 55
    10.1 Beispiel Schulbibliothek 55
    10.2 REMIS – Funktionen nach Gruppen 75
    10.3 REMIS – Funktionen (alphabetisch) 86
    10.4 Syntax des SELECT – Kommandos (sql) 117

Vorwort

Der vorliegende Text wendet sich an den in der Benutzung von EUMEL (und ELAN) erfahrenen Programmierer. Auch dieser wird vor naiver Benutzung von REMIS gewarnt und zu sorgfältigster Programmierung ermahnt. Vor der Erstellung einer ernsthaften Anwendung sollte er die erforderlichen REMIS – Funktionen ausprobieren. REMIS ist (auch aus Performance – Gründen) gegen unordentliche Benutzung nicht abgesichert.
Hinweise auf mögliche Fehler werden in das Handbuch eingearbeitet.

Hinweise an: U. Schneider
c/o GMD, Z2.W
Postfach 1240

D – 5205 St. Augustin 1

Tel.: 02241/14 2475

1. Überblick

Das REMIS – Datenmodell

Datenstruktur

In einer REMIS – Datenbank werden Objektklassen (CLASS) eingerichtet.
Einer Klasse werden Segmente (oder Felder) (SEGMENT) zugewiesen.
Zu einem Objekte einer Klasse gehört aus jedem zur Klasse gehörenden Segment ein Wert.
Insoweit entspricht die REMIS – Datenstruktur der Datenstruktur des Relationenmodells.
Segmente können vom Typ TEXTSEGMENT, INTSEGMENT oder REALSEGMENT sein und jeweils Werte des entsprechenden ELAN – Datentyps aufnehmen:

TEXT	variabel lang, maximal 32 000 Zeichen
INT	ganze Zahl zwischen – 32 768 und + 32 767
REAL	Gleitkommazahl mit 13 – stelliger Mantisse und Exponenten zwischen – 126 und + 126

Weiterhin bietet REMIS (ähnlich Codasyl – Datenbanksystemen) die Möglichkeit, Beziehungen zwischen Objekten zu verwalten. Hierfür gibt es noch die Segmenttypen REFSEGMENT und GROUPSEGMENT, die Werte der in REMIS definierten Datentypen REF und GROUP aufnehmen können.

Ein REF – Wert (Referenz) ist ein interner Ojektbezeichner *),
ein GROUP – Wert eine Menge von internen Objektbezeichnern.

REMIS bietet keinen wahlfreien Zugriff auf Objekte über Segmentwerte an (nur eine Suchfunktion, die auf sortierten Objektmengen binär sucht, sonst fortlaufend). Für schnellen Zugriff auf zu einem Objekt gehörende andere Objekte sollten REF – und GROUP – Segmente eingerichtet werden. Anders als bei Codasyl – Systemen ist die Verknüpfung von Objekten in REMIS nur einseitig gerichtet. Mit anderen Worten: Wenn bei einem Objekt eine Referenz auf ein anderes Objekt eingetragen wird, dann wird von REMIS nicht auch beim referierten Objekt eine Referenz in Gegenrichtung (auf das referierende Objekt) eingetragen.

*) Interne Objekt – Bezeichner
Man kann sich die Realisierung einer Klasse als Tabelle vorstellen, in die die Objekte eingetragen werden.
Objekte behalten, solange sie existieren, ihren festen Platz innerhalb der Tabelle. Daher können sie innerhalb von REMIS eindeutig durch die Angabe ihres Tabellenplatzes bezeichnet werden. Mithilfe der gleichen Angabe können sie auch schnellstmöglich erreicht werden. (Wahlfreier Zugriff, etwa über Inverslisten, ist nicht realisiert.)

Die REMIS – Begriffe entsprechen Begriffen aus anderen Datenmodellen in etwa wie folgt:

REMIS	Datei	Relationenmodell	Codasyl
Klasse	(Datei)	(Relation, Tabelle)	(Satzart)
Gruppe	Datei	Relation, Tabelle bzw. View	Satzart (Subschema) + Set
Objekt	Satz	Tupel, Zeile	Satz
Segment	Feld	Attribut, Spalte	Feld
Referenz		Cursor	current of run – unit

Anmerkung:

Die der Klasse entsprechenden Begriffe stehen in Klammern, da durch Datenmanipulationsanweisungen Objekte nur als Gruppenmitglieder erreicht werden können, nicht aber als Klassenmitglieder (s.u.).

Ein Beispiel

Es soll eine Datenbank eingerichtet werden, mit der insbesondere der Bereich Ausleihe in einer Schulbibliothek unterstützt wird.

Diese Datenbank sollte Daten über den Buchbestand der Bibliothek enthalten und Daten über Ausleihen:

Daten über Bücher
- Signatur (Nummer oder Bezeichnung eines einzelnen Buches)
- Autor
- Titel
- Verlag
- Erscheinungsjahr

Daten über Ausleihen
- Signatur
- Ausleihdatum
- Ausleihername
- Ausleiheradresse

Oder mit anderen Worten: In der Datenbank müssen verwaltet werden
- Datengebilde oder Objekte, die ein Buch beschreiben und
- Datengebilde oder Objekte, die eine Ausleihe beschreiben.

Da in der Schulbibliothek sicher viele Werke in mehreren Exemplaren vorhanden sind und in mehreren Ausleih – Objekten derselbe Ausleiher aufgeführt wird, würden vielleicht besser folgende Objektklassen (Menge gleichartiger Objekte) eingerichtet:

WERK – Objekte	EXEMPLAR – Objekte	BENUTZER – Objekte
Werknummer	Signatur	Benutzernummer
Autor	Werknummer	Name
Titel	Ausleihdatum	Vorname
Verlag	Benutzernummer	Adresse
Erscheinungsjahr		(für Schüler: Klasse für Lehrer: "L")

Zwischen diesen Klassen gibt es Beziehungen, die dadurch dargestellt sind, daß in den EXEMPLAR – Objekten auch die Werknummer und die Benutzernummer aufgeführt sind.

Diese Beziehungen haben folgende Bedeutung:

W : Werkexemplar
umgekehrt gelesen: ist Exemplar von

A ausgeliehen an
umgekehrt: hat ausgeliehen

Hierfür sollen folgende Integritätsbedingungen gelten:

W1: Zu jedem WERK – Objekt gibt es mindestens ein EXEMPLAR – Objekt (mit derselben Werknummer)

W2: Zu jedem EXEMPLAR – Objekt gibt es ein WERK – Objekt

A1: Zu jedem EXEMPLAR – Objekt im Zustand "ausgeliehen" muß ein BENUTZER – Objekt existieren
(d.h. im EXEMPLAR – Objekt und im BENUTZER – Objekt müssen dieselbe Benutzernummer stehen)

Für diese Anwendung könnte eine REMIS – Datenbank mit folgender Datenstruktur eingerichtet werden:

(Objekt –)Klasse WERK mit den Segmenten
- autor
- titel
- verlag
- erscheinungsjahr
- exemplare (* Gruppe der zugehörigen Exemplare *)

(Objekt –)Klasse EXEMPLAR mit den Segmenten
- signatur
- ausleihdatum
- werk (* Referenz auf zugehörige Werkdaten *)
- ausleiher (* Referenz auf Ausleiherdaten *)

(Objekt –)Klasse BENUTZER mit den Segmenten
- name
- vorname
- klasse
- entliehen (* Gruppe der ausgeliehenen Exemplare *)

Datenmanipulation

Für Einrichtung, Benutzung und Verwaltung müssen Programme in ELAN geschrieben werden. Die REMIS – Sprache ist eingebettet in ELAN.
(Für Datenbankabfragen steht 'sql' zur Verfügung, das aus einer SELECT – Anweisung ein ELAN – Programm generiert und ausführt.)

Beispiel:

Alle Titel aus der Gruppe "alle werke" ausgeben:

```
TEXTSEGMENT CONST titel := segment("titel","WERK");  (*kann entfallen
REF VAR werk := first(group("alle werke"));
WHILE another(werk)
REP line; put(werk_titel);
    next(werk)
PER
```

oder

```
   SELECT titel FROM "alle werke"
```

Ein Programm kann nur bestimmte (referierte) Objekte (bzw. seine Segmente) bearbeiten. Es muß also zunächst eine Referenz (REF – Variable) auf ein Objekt positionieren. Dies geschieht durch Positionierung der Referenz in einer Gruppe oder durch Lesen des Referenzwertes aus einem Refsegment. Insbesondere ist das Erzeugen von Objekten nur möglich durch die Funktion "insert", die einen neuen Referenzwert erzeugt und in eine Gruppe einfügt.

Eine Gruppe enthält eine geordnete Menge von Referenzen auf Objekte derselben Klasse. Einer Gruppe werden Segmenttypen der zugehörigen Klasse in einer gewünschten Reihenfolge zugeordnet. Die Gruppe entspricht damit in etwa der herkömmlichen formatierten Datei (bei Datenbanksystemen der logischen Datei, Benutzersicht (view) oder dem Subschema).

Die Reihenfolge der Objekte in einer Gruppe wird hergestellt beim Einfügen der Objekte (Referenzen) durch ein Programm bzw. durch Sortierung.

Beispiel

Auf der Klasse WERK mit folgendem Inhalt

autor	titel	verlag	erschj
Müller	Englisch 1	Knaur	1975
Müller	Englisch 2	Knaur	1975
May	Winnetou 1	xyz	1890
May	Winnetou 2	xyz	1892
May	Winnetou 3	xyz	1893

kann man z.B. folgende Gruppen definieren:

Gruppe: alle werke

autor	titel	verlag	erschj
May	Winnetou 1	xyz	1890
May	Winnetou 2	xyz	1892
May	Winnetou 3	xyz	1893
Müller	Englisch 1	Knaur	1975
Müller	Englisch 2	Knaur	1975

Gruppe: fachliteratur

autor	titel	erschj
Müller	Englisch 1	1975
Müller	Englisch 2	1975

Gruppe: unterhaltung

autor	titel
May	Winnetou 1
May	Winnetou 2
May	Winnetou 3

Merke:

1. Objekte stehen nur als Mitglieder von Gruppen zur Verfügung.

2. Ein einzelnes Objekt wird durch eine Referenz (Wert einer REF – Variablen) bezeichnet (referiert).

3. Der Wert einer REF – Variablen wird nur von REMIS, nicht vom Programm gesetzt. Vom Programm wird eine Referenz positioniert (first, next, last, prior).

4. Eine Referenz wird im Prinzip auf ein Objekt in einer Gruppe positioniert. (Wurde eine Referenz aus einem Refsegment gelesen, dann gehört sie sozusagen zu einer unbenannten Gruppe, die nur das referierte Objekt enthält. Eine anschließende Positionierung ist sinnlos.)

5. Schreib – /Lesefunktionen betreffen nur einzelne Segmente eines Objekts.

6. Segmente können nicht unmittelbar bearbeitet werden. Von REMIS werden nur Werte in Segmente eingetragen bzw. Segmentwerte ausgegeben. Dies gilt insbesondere auch für Referenz – und Gruppensegmente.
   (Ausnahme: Operator _, s.u.)

REMIS – Datentypen

REF	zeigt auf ein (oder kein) Objekt (und ggfs. eine Position in einer Gruppe und ggfs. ein Standard – Formular)
GROUP	enthält eine Reihe von Referenzen (auf Objekte einer Klasse und ggfs. ein Standardformular)
TAG	(Notizzettel) enthält ein Formular

Grenzwerte

Klassen	je Datenbank	50
Segmente	je Datenbank	146 minus Anzahl der Klassen
Objekte	je Klasse	15 000
Gruppen	je Datenbank	500
Formulare	je Datenbank	100
Felder	je Formular	100

Benutzung von REMIS

Zunächst wird REMIS auf dem Rechner in einer Manager – Task mit dem Namen REMIS installiert. Unter REMIS werden Arbeitstasks eingerichtet, in denen die Arbeiten mit der Datenbank durchgeführt werden (siehe Kapitel 9.2).

Für die Programmierung stehen zur Verfügung:

- Datenbankfunktionen
- Formularfunktionen (Maskenfunktionen)
- kombinierte DB/Formularfunktionen
- Menue – Generator
- Abfragesprache (Klein – sql)

2. Kapitel entfällt

3. Einrichten einer Datenbank

3.1 Überblick

Es ist recht einfach auf REMIS ein Anwendungssystem zu entwickeln. Im Laufe der Entwicklung ergeben sich jedoch meist Gesichtspunkte, die eine Änderung der Datenstruktur erfordern. Solche Änderungen (abgesehen vom Hinzufügen von Datengebildetypen (Klassen, Segmente, Gruppen)) können umfangreiche Arbeiten (Datenumwandlung, Programmänderung, – übersetzung, – insertierung) zur Folge haben. Der Entwicklung des endgültigen Anwendungssystems sollte daher ein sorgfältiger Datenbankentwurf vorausgehen, der natürlich durch die Entwicklung von Prototypen sehr gut unterstützt werden kann.

Eingerichtet wird die Datenbank in einer Arbeitstask. Es werden eingetragen
- die erforderlichen Datengebildetypen
- ggf. Formularbeschreibungen.

Es ist nützlich, die Datenbasis mit Daten – und Formularbeschreibungen zu sichern. Abschließend sollten für die weitere Arbeit die Segmentnamen zur Verfügung gestellt werden.

3.2 Datenbeschreibung

Bevor mit der Datenbank gearbeitet werden kann, müssen die erforderlichen Klassen, Segmente und Gruppen eingerichtet werden.
Da Zugriff auf Objekte nur über Gruppen möglich ist, ist je Klasse mindestens eine Gruppe erforderlich.
Für eine Gruppe werden die über diese Gruppe erreichbaren Segmente der zugehörigen Klasse festgelegt.

Die Eigenschaften (zugeordnete Segmente, Standardformular, ...) von Gruppen, die in GROUP – Segmenten abgelegt werden, werden für eine Beispielgruppe deklariert, die dem GROUPSEGMENT – Typ zugeordnet wird. Beispielgruppen sollten nicht zur Verarbeitung benutzt werden, also leer bleiben.

Mit den u.a. Prozeduren sollte in einer Datei ein ELAN – Programm erstellt werden. Es legt bei Ausführung (run) in der Datenbasis eine Datenstrukturbeschreibung (Schema) ab. Mithilfe dieser Prozeduren können jederzeit weitere Klassen, Segmente und Gruppen in die Datenbank aufgenommen werden.

Anmerkung: Im Folgenden sind unter Namensparameter (...name) Textkonstante (z.B. "abc") zu verstehen.

Klasse einrichten

```
new class (klassenname)
```

Anmerkung: Segmentnamen müssen in der Datenbank eindeutig sein.
Sie sollten den Namensregeln für ELAN entsprechend gebildet werden, damit sie auch als Variablennamen verwendet werden können.

Segment einrichten

```
new segment (segmentname, klassenname, standardwert)
```

richtet ein: wenn standardwert vom Typ:

INTSEGMENT	INT
REALSEGMENT	REAL
TEXTSEGMENT	TEXT

```
new segment (segmentname, klassenname,
             name der referierten klasse, referenz)
```

richtet ein REFSEGMENT ein;
referenz muß nilref sein oder ein Objekt der referierten Klasse referieren.

Gruppe einrichten

```
GROUP VAR g := new group (klassenname, gruppenname)
```

Gruppensegment einrichten

```
new segment (segmentname, klassenname, gruppenvariable)
```

richtet ein GROUPSEGMENT ein mit den Eigenschaften der durch gruppenvariable (z.B. g) bezeichneten Beispielgruppe und einer leeren Gruppe als Standardwert. Gruppenname und segmentname müssen gleich sein.

Einer Gruppe Segmente zuweisen

```
add group segment (gruppenvariable, SEG segmentname)
all class segments (gruppenvariable)
```

ordnet der Gruppe alle noch nicht zugeordneten Segmente der zugehörigen Klasse zu in der dort vorliegenden Reihenfolge (wichtig für sort)

Konversionsmethode festlegen

Zur Erleichterung der Programmierung werden von einigen Datenbankfunktionen INTSEGMENT-Werte in Texte konvertiert und umgekehrt. Folgende Konversionsmodi stehen zur Verfügung:

modus		innen	außen
0	zahl als text	25	"25"
1	Konversionstabelle	2	"Frau"
2	Datumskonversion	– 1396	"06.03.84"

Es gebe die INTSEGMENTe "anrede" und "ausleihdatum":

Modus 0: Standardwert

Modus 1:
```
new methode (SEG "anrede", 1);
    new conversion table entry (SEG "anrede", "", 0);
    new conversion table entry (SEG "anrede", "Herr", 1);
    new conversion table entry (SEG "anrede", "Frau", 2)
```

Modus 2:
```
new methode (SEG "ausleihdatum", 2)
```

Eine Beschreibung des erzeugten Schemas kann erstellt werden durch:

```
dblist
```

Die Beschreibung wird in einer Datei ("DB – list") erzeugt und durch den Editor angezeigt. Sie kann gedruckt werden mit dem Editor – Kommando:

```
ESC ESC print
```

Segmentnamen zur Verfügung stellen

```
insert declarations
```

Die Segmentnamen stehen anschließend für die Programmierung in dieser Task (und ihren Söhnen) zur Verfügung.

Weitere Prozeduren zur Bearbeitung des Schemas sind im Anhang aufgeführt.

3.3 Formulardefinition

Formulare (TAGs) werden in der Datenbank aufbewahrt und über einen Namen identifiziert. Für Erstellung bzw. Veränderung eines Formulars stehen die Dialogfunktionen

```
design form
design fields
```

zur Verfügung. Für die Erstellung eines neuen Formulars wäre damit z.B. folgendes Programm zu erstellen:

```
TAG VAR t;
TEXT VAR name;
nil(t);
design form(t);
design fields(t);
put("Bitte Formularnamen eingeben");
get(name);
store tag(t,name)
```

Formularvariable definieren

```
TAG VAR t
```

Leeres Formular definieren

```
nil (t)
```

Formularbild entwerfen

```
design form (t)
```

Das Bild von t wird in einer Textdatei zur Bearbeitung mit edit angeboten. Dabei haben folgende Zeichen eine besondere Bedeutung:

blank	DURCHSICHTIG	hier wird bei der Formularausgabe der Bildschirm nicht verändert
!	BLANK	hier wird ein blank ausgegeben
<	INVERS AN	ab hier erfolgt die Ausgabe in Invers – Darstellung
>	INVERS AUS	ab hier erfolgt die Ausgabe normal

Für die Darstellung von BLANK, INVERS AN, INVERS AUS können mit der Prozedur "trans" andere Zeichen vereinbart werden.

Formularfelder (für Daten) definieren

```
design fields (t)
```

Vom Benutzer sind nacheinander einzugeben:

1. **Feldnummer** (oder **ESC** (beenden) oder anderes)
2. Feldanfang: **Cursor positionieren, RETURN**
   Der Cursor kann auch durch Eingabe eines im Formularbild vorhandenen Zeichens auf das nächste dieser Zeichen (in Schreibrichtung) gesetzt werden.
3. Feldende: **Cursor positionieren, RETURN**
4. **ESC** oder **Wiederholung von (1) und (2)** zur Definition weiterer Teilfelder
   Ein Feld kann aus mehreren Teilfeldern (jeweils in einer Zeile) zusammengesetzt werden. Die Schritte (2) bis (4) werden wiederholt, solange hier nicht ESC eingegeben wird.

5 . Feldeigenschaften

- **geschützt** (keine Eingabe über dieses Feld)
- **geheim** und **Ausgabezeichen**
  Anstelle jedes in diesem Feld auszugebenden Zeichens wird das Ausgabezeichen ausgegeben.
- **Segmentname** oder **symbolischer Feldname** (numerisch)
- Auskunftsnummer (Nummer einer diesem Feld zugeordneten Erklärung

Alle Eingaben werden mit RETURN beendet (oder ESC, wenn keine weiteren Angaben gemacht werden sollen (s.a. Kap. 4.3).

Formular speichern

```
store tag (t, name)
```

speichert t unter dem Namen name

Weitere Prozeduren zur Formularbehandlung sind im Anhang aufgeführt.

4. Programmieren

4.1 Überblick

Ein Programm arbeitet im wesentlichen mit den REMIS – Elementen Segment, Gruppe, Referenz. Segmente und Gruppen haben bei Einrichtung der Datenbank (externe) Namen bekommen. Diese müssen für den Zugriff in interne Bezeichner (Adressen) umgesetzt werden. Damit dies nicht bei jeder Operation geschehen muß, verwenden viele REMIS – Operationen als Parameter interne Bezeichner, die sich das Programm vorher von REMIS übergeben lassen muß. Dem ELAN – Programmierer wird die zu verwendende Technik sicher aus folgendem Beispiel klar:

```
SEGMENT CONST autor := segment("autor");
REF VAR werk:=first(group("wgr"));
put(werk_autor);
```

gibt den Autor des ersten Werks in der Gruppe "wgr"
auf dem Bildschirm aus
Definition von _ (siehe Anhang):
TEXT OP _ (REF CONST r, SEGMENT CONST s)

Die Definition von Segmentbezeichnern kann entfallen, wenn in der Task (nach Laden der Datenbank)

```
insert declarations
```

ausgeführt wurde. Anschließend stehen alle (externen) Segmentnamen als Segmentbezeichner zur Vefügung.

Die Funktionen zur Bearbeitung der Datenbank (lesen, schreiben) sind (wenn nicht Formulare verwendet werden, s.u.) auf einzelne Segmente eines bestimmten Objekts (Datensatz) bezogen. Objekte werden nur über Gruppen zur Verfügung gestellt. Einzelne Objekte einer Gruppe werden bezeichnet (referiert) durch dieser Gruppe zugeordneten REF – Variable.

Im Programm werden also zunächst REF – Variable definiert, einer Gruppe zugeordnet und in der Gruppe positioniert. Einer Gruppe können mehrere REF – Variable zugeordnet werden.

Anschließend können Objekte behandelt werden:

- Objekt anlegen
- Objekt löschen
- Objekt in Gruppe einfügen
- Objekt aus Gruppe ausfügen
- Gruppe leeren
- Gruppe sortieren
- Segment schreiben
- Segment lesen

REMIS enthält Funktionen, die es gestatten, Daten über vordefinierte Formulare vom Bildschirm einzulesen bzw. auf dem Bildschirm zur Vearbeitung anzubieten. Ebenso können Formulare für die Datenausgabe verwendet werden. Für die Bearbeitung von Datenbankdaten über Formulare stehen besonders komfortable Funktionen zur Verfügung.

Ein Formular wird von REMIS in einer Variablen vom Typ TAG (Notizzettel) aufgebaut. Formulare können auf dem Bildschirm bearbeitet oder auf ein Pinboard geheftet werden, auch neben – und übereinander. Das zuletzt ausgegebene Formular überschreibt die vorhergehenden außer an den DURCHSICHTIGen Stellen. Das Pinboard kann auf ein Ausgabemedium (z.B. Datei oder Drucker) ausgegeben werden.

4.2 Datenbankfunktionen

Im Folgenden wird für externe Namen, die als Textkonstante zu übergeben sind, die Bezeichnung '...name' verwendet, sonst sind interne Bezeichner gemeint.

Segmentbezeichner definieren (gegebenenfalls):

```
SEGMENT VAR s := segment(segmentname)
SEGMENT VAR s := segment(segmentname, klassenname)
```

Gruppenbezeichner definieren (nicht immer erforderlich)

```
GROUP VAR g := group(gruppenname)
```

REF – Variable definieren und positionieren

```
REF VAR r := first (group (gruppenname))
REF VAR r := last  (group (gruppenname))
```

REF positionieren

`next (r)`: nächstes Objekt der Gruppe
`prior (r)`: vorhergehendes Objekt der Gruppe
`first (r)`: erstes Objekt der Gruppe
`last (r)`: letztes Objekt der Gruppe

`first (r, segmentbez = suchwert)`:
erstes Objekt der Gruppe, das im angegebenen Segment den angegebenen Wert enthält. Falls ein solches Objekt gefunden wurde, zeigt r auf dieses Objekt, andernfalls bei nach dem angegebenen Segment sortierten Gruppen auf das nächste, sonst hinter das letzte Objekt der Gruppe.

`next (r, segmentbez = suchwert)`:
wie first, jedoch wird ab dem nächsten Objekt in der Gruppe weitergesucht

Test auf gültige Referenz

`another (r)`: liefert TRUE, falls r ein gültiges Objekt referiert
`found`: liefert TRUE, falls durch das letzte first oder next mit Auswahlbedingung ein Objekt gefunden wurde

Objekt anlegen

`insert (r)`:
legt ein neues Objekt in der zugehörigen Klasse an und fügt es in die zugehörige Gruppe vor der aktuellen Position ein. r zeigt anschließend auf das neue Objekt.

Anmerkung: Es ist sicherzustellen, insbesondere für Ref – und Groupsegmente, daß vor dem Lesen (s.u.) eines Segmentwertes ein definierter Wert eingetragen wurde (Segment schreiben). Falls dies durch das Anwendungssystem nicht garantiert werden kann, sollte unmittelbar nach insert in alle Segmente des neuen Objekts ein Wert eingetragen werden (z.B. mit reset).

Objekt löschen

`delete (r)`:
löscht das durch r referierte Objekt aus der zugehörigen Klasse. Dadurch ist es auch aus allen Gruppen, in denen es enthalten war, entfernt. r referiert anschließend das nächste Objekt der zugehörigen Gruppe oder das Gruppenende.

Objekt in Gruppe einfügen

`hold (gruppe, r)`:
hängt r an gruppe an. gruppe gilt anschließend als 'nicht sortiert'.
`hold (wohin, r)`:
fügt r vor der durch die REF – Variable wohin bezeichneten Position in die zu wohin gehörende Gruppe ein. Das Sortierkennzeichen der Gruppe wird nicht verändert. wohin zeigt anschließend auf das eingefügte Objekt (r).

Objekt aus Gruppe ausfügen

`unhold (r):`

> r wird aus der zugehörigen Gruppe entfernt. r referiert anschließend kein <
> Objekt (another (r) – – > FALSE). <

Gruppe leeren

`clear (gruppe):` entfernt alle Referenzen aus der Gruppe

Gruppe sortieren

`sort (gruppe, anzahl):`
sortiert die durch gruppe referierten Objekte nach den ersten anzahl Segmenten (bezogen auf die Reihenfolge, in der die Segmente der Gruppe zugewiesen wurden)

Segment schreiben

`write (r, segment, wert):`
schreibt wert in das angegebene Segment des referierten Objekts. Die Typen von segment und wert müssen einander entsprechen (TEXT, INT, REAL, REF, GROUP). Ist segment vom Typ SEGMENT, dann muß wert vom Typ TEXT sein. In diesem Fall wird die vereinbarte Konversion (siehe Kap. 3.2) durchgeführt.
`reset (r, segment):` trägt den Standardwert ein, sonst wie write.

Anmerkung: Der Programmierer muß sicherstellen, daß r ein gültiges Objekt referiert (IF another (r) ...) und daß segment zur referierten Klasse gehört. Andernfalls erfolgt die Eintragung auf eine falsche Stelle.

Segment lesen

`read (r, segment, wert):`
übergibt in der Variablen wert den Inhalt des angegebenen Segments des referierten Objekts. Die Typen von segment und wert müssen einander entsprechen (TEXT, INT, REAL, REF, GROUP). Ist segment vom Typ SEGMENT, dann muß wert vom Typ TEXT sein. In diesem Fall wird die vereinbarte Konversion (siehe Kap. 3.2) durchgeführt.

`r_segment:`
Der _ – Operator liefert den Wert des angegebenen Segments des referierten Objekts (nicht für GROUPSEGMENTe). Der Operator ist schwächer als die in ELAN definierten Operatoren. Der Ausdruck r_segment sollte daher in Ausdrücken mit anderen Operatoren eingeklammert werden: (r_segment).

Anmerkungen:

1. In segment muß ein gültiger Wert eingetragen sein, sonst ist das Ergebnis unbestimmt (Fehler oder falsch).
2. Aus REFSEGMENTen wird der gefundene Wert geliefert, auch wenn er kein gültiges Objekt referiert. Wenn dieser Fall eintreten kann, dann muß wert vor der Weiterverwendung geprüft werden (IF another (wert) ...).
3. Vom Programmierer ist sicherzustellen, daß r ein gültiges Objekt referiert und segment zur referierten Klasse gehört. Andernfalls ist das Ergebnis falsch.

4.3 Anwendung von Formularen

Verwendung von Formularen im Dialog

Zunächst wird das Formular auf den Bildschirm geschrieben. (Vorher Bildschirm loeschen?) Anschließend können über einzelne durch die Feldnummer bezeichnete Formularfelder Daten vom Bildschirm gelesen, auf dem Bildschirm ausgegeben oder auf dem Bildschirm zur Bearbeitung angeboten werden. Die Feldnummer bezieht sich auf Felder (nicht Teilfelder). Ebenso arbeitet das Programm mit Feldinhalten (nicht Teilfeldinhalten).

Mit den folgenden Tasten kann die Schreibmarke auf dem Bildschirm gesteuert werden und es können Eingaben korrigiert werden:

in einem Feld

RECHTS	Cursor eine Stelle nach rechts
LINKS	Cursor eine Stelle nach links
HOP LINKS	Feldanfang
HOP RECHTS	hinter bzw. auf letztes Zeichen
RUBIN	fügt an der aktuellen Cursorposition ein Leerzeichen ein und verschiebt den Rest des Feldinhaltes um ein Zeichen. Wenn das Feld (nicht Teilfeld!) überläuft, werden die letzten Zeichen nicht mehr dargestellt.
RUBOUT	löscht an der aktuellen Cursorposition ein Zeichen, der Rest des Feldinhalts wird nachgerückt.
HOP RUBOUT	löscht Feldinhalt ab Cursorposition

auf anderes Feld

RETURN	nächster bzw. letzter Feldanfang
HOP RETURN	Anfang des vorhergehenden bzw. ersten Feldes
UNTEN	eine Zeile nach unten, falls dort ein Feld ist
OBEN	eine Zeile nach oben, falls dort ein Feld ist

Formularbearbeitung beenden

ESC
bei Einfeldbearbeitung auch durch Verlassen des Feldes (s.o.)

Anmerkung: Die Zuordnung dieser und evtl. weiterer Positionierungsfunktionen kann sich durch Anpassung an die Textverarbeitung ändern.

Verwendung von Formularen ohne Datenbankbezug

Formulare werden dem Programm in TAG – Variable zur Verfügung gestellt:

```
TAG VAR t
t := tag (formularname)          oder
t INITBY formularname
```

Formular auf den Bildschirm schreiben

```
(* page;        ggfs. Bildschirm löschen *)
show (t)
```

Für die folgenden Funktionen muß feldnummer als Zahlvariable definiert sein und vor der Benutzung einen Wert zugewiesen bekommen:

```
INT VAR feldnummer;
feldnummer := 2
```

Ein Feld bearbeiten

text in Feld feldnummer von t ausgeben

```
put (t, text, feldnummer)
```

text von Feld feldnummer von t lesen

```
get (t, text, feldnummer)
get (t, feldnummer)          (* liefert den Text *)
```

text auf Feld feldnummer von t zur Bearbeitung anbieten

```
putget (t, text, feldnummer)
```

mehrere Felder bearbeiten

Bei folgenden Prozeduren kann der Benutzer mehrere Formularfelder bearbeiten. Dazu muß ein Bereich von 100 Textwerten übergeben werden. Die Werte werden den Formularfeldern in der Reihenfolge ihrer Feldnummern zugeordnet.

```
ROW 100 TEXT VAR texte;
(* vor put, putget: Bereich, soweit benutzt, mit Werten füllen, z.Bsp.
FOR i FROM 1 UPTO fields(t)
REP texte(i) := "" PER              *)

put (t, texte)
get (t, texte, feldnummer)
putget (t, texte, feldnummer)
```

Die Schreibmarke wird anfangs in das Feld feldnummer gesetzt. Sie kann vom Benutzer über alle nicht schreibgeschützten Formularfelder bewegt werden. Die Bearbeitung des Formulars wird beendet durch ESC. feldnummer enthält dann die Nummer des verlassenen Feldes.

weitere Prozeduren

Größte jemals benutzte Feldnummer in Formular t

```
fields (t)
```

Anzahl der von Formular t belegten Zeilen

```
y size (t)
```

Länge des Feldes feldnummer von Formular t

```
length (t, feldnummer)
```

Schreibschutz für Feld feldnummer in Formular t

```
protect(t,feldnummer,TRUE)   (* schützen *)
protect(t,feldnummer,FALSE)  (* freigeben *)
```

Formular vertikal verschieben

```
t SCROLL zeilenzahl
```

positive zeilenzahl: nach unten
negative zeilenzahl: nach oben
Hierdurch ist es möglich, Formulare beginnend mit Zeile 1 zu entwerfen und im Programm auf eine in der jeweiligen Dialogsituation geeignte Zeile zu verschieben.

Code des Zeichens, mit dem das Formular verlassen wurde

```
leaving code    (z.B. 27 (ESC))
```

Cursor entsprechend Benutzereingabe auf anderes Feld

```
execute command code(t, feldnummer)
```

Mit den letzgenannten Funktionen ist es möglich anstelle von get (t,texte,feldnummer) eine entsprechende eigene Prozedur zu schreiben:

```
REP get(t,texte(feldnummer),feldnummer);
    IF ok THEN execute command code(t,feldnummer) FI
UNTIL leaving code = esc PER.
ok: IF feldnummer = 4
    THEN plausibles erscheinungsjahr
    ELSE TRUE
    FI.
plausibles erscheinungsjahr:
    texte(feldnummer) = "" COR
    (int(texte(feldnummer)) > 1900 AND
    int(texte(feldnummer)) < 1985)
```

Hierdurch wird der Benutzer im Feld 4 (des Formulars "werk") festgehalten, bis er kein oder ein plausibles Erscheinungsjahr oder ESC eingibt.

Verwendung von Formularen mit Datenbankdaten

Zusätzlich zu den bisher genannten stehen für die Bearbeitung von Datenbankdaten neben anderen die folgenden Funktionen zur Verfügung. Zur Erleichterung der Programmierung wird dabei ausgenutzt, daß
- den Formularfeldern Segmente zugeordnet wurden
- zu einer Gruppe ein Standardformular definiert wurde.

Dadurch können mit einer Operation alle einem Formular zugeordneten Segmente eines referierten Objekts bearbeitet werden (ohne Umweg über ROW 100 TEXT).
Außerdem kann das zu einer Referenz gehörende Formular angesprochen werden.

Der Zusammenhang wird folgendermaßen hergestellt:
1. Die Referenz – Variable gehört zu einer Gruppe:
   Das der Gruppe zugeordnete Formular wird benutzt und die Segmente, die Gruppe und Formular zugeordnet sind.
2. Die Referenzvariable gehört nicht zu einer Gruppe:
   Der Referenzwert wurde z.B. aus einem refsegment gelesen. Dieser Referenzwert kann aber vorher nur in einer Variablen erzeugt worden sein, die einer Gruppe zugeordnet war. Der Referenzwert übernimmt dabei das der Gruppe zugeordnete Formular. In diesem Fall werden alle Segmente benutzt, die zum Formular und zur referierten Klasse gehören.
3. Falls die in Frage kommende Gruppe kein Standardformular hat, hat auch die Referenz keins.

Formular zur Verfügung stellen

```
TAG VAR t := tag (ref)
```
(kann entfallen, wenn t im Programm nicht benutzt wird)

Formular zeigen

```
show (ref)
```

Alle zugeordneten Segmente des referierten Objekts ausgeben, lesen, bearbeiten lassen

über das zugehörige Formular:

```
put (ref)
get (ref)
putget (ref)
```

(Dieses Formular kann nicht durch SCROLL verschoben werden.)

über ein beliebiges Formular t:

```
put (ref, t)
get (ref, t)
putget (ref, t)
```

text auf dem Segment s zugeordneten Feld von t ausgeben

```
put (t, s, text)
```

Verwendung von Formularen für die Ausgabe

Für die Datenausgabe werden Formulare mit Daten gefüllt und auf ein Pinboard geheftet, das nach Fertigstellung auf das aktuelle Ausgabegerät (z.B. Bildschirm) oder in eine Datei kopiert werden kann. Formulare können auch übereinander geheftet werden.
Auf einem Pinboard wird man meist das Bild einer Druckseite erzeugen.

Feld feldnummer des Formulars t mit text füllen

```
fill (t, text, feldnummer)
```

Pinboard leeren

```
clear board
```

Vorbereitetes Pinboard aus Datei datei laden

```
get board (datei)
```

Formular t auf Pinboard heften

linke obere Ecke auf Position (spalte, zeile):

```
pin (t, spalte, zeile)
pin (t, spalte)              (nächste freie Zeile)
pin (t)            (Spalte 1, nächste freie Zeile)
```

linke obere Ecke auf Symbol auf Pinboard:

```
pin (t, symbol)
pin (t, symbol, erfolg)
```

(erfolg wird auf TRUE gesetzt, wenn symbol gefunden wurde, sonst FALSE;
vorher deklarieren: BOOL VAR erfolg)

text ohne Benutzung eines Formulars auf Pinboard heften

```
pin (text, spalte, zeile)
```

Nächste freie Zeile (Inhalt des Zeilenmerkers)

```
next boardline    (liefert ganze Zahl (INT))
```

Nächste Ausgabezeile bestimmen (Zeilenmerker setzen)

```
set boardline (zahl)
```

Pinboard ausgeben

```
put board             auf aktuellem Ausgabemedium (z.B. Bildschirm)
put board (datei)     in Datei datei
```

Ausgabe von Datenbankdaten

Formular t mit den zugeordneten Segmenten des durch ref referierten Objekts füllen

```
t FILLBY ref
```

Das dem Segment s zugeordnete Feld von t mit text füllen

```
fill (t, s, text)
```

5. Datenbankverwaltung

Sicherung gegen Zerstörung

REMIS enthält keine eigenen Sicherungsvorkehrungen außer

```
save base.
```

Die Datenbank kann also gegen Fehler in der Arbeitstask gesichert werden, indem sie durch save base zur Sicherungstask kopiert wird. Dies sollte immer dann geschehen, wenn sie einen aufzubewahrenden Inhalt (insbesondere auch Formulare) bekommen hat.

Insbesondere ist dies auch dann nützlich, wenn das in der Arbeitstask insertierte Anwendungssystem ersetzt werden soll. Häufige Vorgehensweise:

```
save base (/"BIBsave")
end
begin
load base (/"BIBsave")
insert declarations
```
neues System insertieren

Archivieren der Datenbank

```
begin ("bibsave","BIBsave") (* um "BIBsave" nicht zu gefährden *)
fetch all
archive (archivname)
clear (archive)
save all (archive)
end
```

archivierte Datenbank laden: `fetch all (archive)`

Gleichzeitige Mehrfachbenutzung

Gleichzeitiges Ändern der Datenbank aus verschiedenen Tasks ist mit der Single – User – Version (ohne Parallel – Zugriff) nicht möglich. Unterschiedliche Datenbankkopien können nicht konsistent zusammengeführt werden.
Allerdings kann nebeneinander aus verschiedenen Datenbankkopien Ausgabe erzeugt werden.

Sicherung gegen unbefugte Benutzung

Gegen unbefugte Benutzung kann nur gesichert werden, indem die Arbeitstask gegen unbefugte Benutzung gesichert wird und in der Arbeitstask dem Benutzer die Möglichkeit zu programmieren entzogen wird. Dies kann zum Beispiel durch Einsatz des Menue – Programms erreicht werden.

6. Menue – Behandlung

Die Menue – Behandlung besteht aus zwei Teilen, einem Laufzeitpaket, das im fertigen Anwendungssystem insertiert sein muß, und einem Programmgenerator, der nur zur Umwandlung der Spezifikation benötigt wird.

Der Generator erzeugt aus einer Spezifikationsdatei mit der externen Darstellung eines Menue – Baums einen Datenraum mit der internen Darstellung des Menue – Baums, sowie ein ELAN – Programm, das diesen Menue – Baum interpretiert, d.h. entsprechend den Tastendrücken des Bedieners zu verschiedenen Menues und Unterprogrammen verzweigt und eine Fehlerbehandlung durchführt.

Für die Darstellung der Menues bei der Ausführung werden zwei Formulare ("morchelrahmen" und "morchelmenue") verwendet, die mit design verändert werden können. Auf "morchelrahmen" werden die ausgewählten Menuepunkte angezeigt und in dem Feld mit dem symbolischen Namen 4711 Fehlermeldungen. Auf "morchelmenue" werden in den Feldern 2 bis vorletztes Feld die Menuepunkte angezeigt und im letzten Feld mit dem symbolischen Namen 4711 Meldungen aus dem Menuebaum.

Die externe Darstellung enthält zu jedem Menuepunkt folgende Angaben:

- Die Hierarchiestufe
- Die Beschreibung der Funktion, die durch Anwahl eines Menue – Punktes angestoßen wird
- optional eine im Falle der Anwahl auszuführende Elan – Prozedur (bei Knoten auf niedrigster Hierarchiestufe, d.h. wenn kein Untermenue folgt, ist sie notwendig)
- optional eine im Falle der Anwahl auszugebende Meldung bei Knoten, denen ein Untermenue folgt

Gleichzeitige Mehrfachbenutzung

Gleichzeitiges Ändern der Datenbank aus verschiedenen Tasks ist mit der Single – User – Version (ohne Parallel – Zugriff) nicht möglich. Unterschiedliche Datenbankkopien können nicht konsistent zusammengeführt werden.
Allerdings kann nebeneinander aus verschiedenen Datenbankkopien Ausgabe erzeugt werden.

Sicherung gegen unbefugte Benutzung

Gegen unbefugte Benutzung kann nur gesichert werden, indem die Arbeitstask gegen unbefugte Benutzung gesichert wird und in der Arbeitstask dem Benutzer die Möglichkeit zu programmieren entzogen wird. Dies kann zum Beispiel durch Einsatz des Menue – Programms erreicht werden.

6. Menue – Behandlung

Die Menue – Behandlung besteht aus zwei Teilen, einem Laufzeitpaket, das im fertigen Anwendungssystem insertiert sein muß, und einem Programmgenerator, der nur zur Umwandlung der Spezifikation benötigt wird.

Der Generator erzeugt aus einer Spezifikationsdatei mit der externen Darstellung eines Menue – Baums einen Datenraum mit der internen Darstellung des Menue – Baums, sowie ein ELAN – Programm, das diesen Menue – Baum interpretiert, d.h. entsprechend den Tastendrücken des Bedieners zu verschiedenen Menues und Unterprogrammen verzweigt und eine Fehlerbehandlung durchführt.

Für die Darstellung der Menues bei der Ausführung werden zwei Formulare ("morchelrahmen" und "morchelmenue") verwendet, die mit design verändert werden können. Auf "morchelrahmen" werden die ausgewählten Menuepunkte angezeigt und in dem Feld mit dem symbolischen Namen 4711 Fehlermeldungen. Auf "morchelmenue" werden in den Feldern 2 bis vorletztes Feld die Menuepunkte angezeigt und im letzten Feld mit dem symbolischen Namen 4711 Meldungen aus dem Menuebaum.

Die externe Darstellung enthält zu jedem Menuepunkt folgende Angaben:

- Die Hierarchiestufe
- Die Beschreibung der Funktion, die durch Anwahl eines Menue – Punktes angestoßen wird
- optional eine im Falle der Anwahl auszuführende Elan – Prozedur (bei Knoten auf niedrigster Hierarchiestufe, d.h. wenn kein Untermenue folgt, ist sie notwendig)
- optional eine im Falle der Anwahl auszugebende Meldung bei Knoten, denen ein Untermenue folgt

Beispiel:

```
1  "n: Titel aufnehmen"                 titel aufnehmen
1  "ä: Titel ändern"                    titel aendern
1  "w: weitere Exemplare aufnehmen"     weitere exemplare
1  "a: Ausleihe"                        ausleihe
1  "r: Rückgabe"                        rueckgabe
1  "l: Exemplar löschen"                exemplar loeschen
1  "b: Benutzeranzeige"                 benutzeranzeige
1  "t: Titelanzeige"                    titelanzeige
1  "z: zeige Liste"
?  Welche Liste?
2  "w: Werkliste"                       werke zeigen
2  "e: Exemplarliste"                   exemplare zeigen
2  "b: Benutzerliste"                   benutzer zeigen
1  "q: Ende"                            break
1  "i: Initialisierung"                 bibinit
//*
```

Aus dieser Spezifikation wird ein Monitorprogramm generiert, das zunächst ein Menue anbietet aus dem einer der Punkte der Stufe 1 ausgewählt werden kann (durch Eingabe des ersten Zeichens oder durch Cursor – Positionierung und RETURN).
Bei 'q' auf Stufe 1 wird die Task unterbrochen,
bei 'z' wird die Meldung 'Welche Liste?' ausgegeben und das zugehörige Untermenue der Stufe 2 angeboten.
Bei Anwahl eines anderen Menuepunkts wird die angegebene (vorher insertierte oder hinter //* beschriebene) Prozedur ausgeführt.

Die externe Darstellung des Menuebaums wird durch eine Zeile mit //* beendet. Danach kann ELAN – Programm folgen.

Ein angebotenes Menue kann vom Benutzer auch durch HOP verlassen werden (jedoch nur bis Stufe 1). Dieses Zeichen kann durch die Prozedur 'return symbol' anders definiert werden.

Menue beschreiben

```
edit("source.name")  (* 'name' ist frei wählbar *)
```
Darin gemäß obigem Beispiel den Menuebaum beschreiben.

Monitor – Programm generieren

```
fetch("MORCHEL std",task("REMIS"))
transform("name")
```
erzeugt: tabelle.name
- name (interne Baumdarstellung)
- monitor.name (Baum – Interpreter)

```
insert("monitor.name")
```

Dialog – Programm starten

```
monitor name        (* kann nicht mehr verlassen werden *)
```

Menue in anderer Task implementieren

name zur Verfügung stellen
monitor.name insertieren
starten

Menue – Baum ändern

source.name ändern
`transform ("name")`
Falls tabelle.name vorhanden ist, wird versucht name zu ändern.
Falls möglich, braucht monitor.name nicht neu insertiert zu werden,
sonst tabelle.name löschen und ab transform wiederholen.

7. Abfragesprache sql

sql generiert aus einem Kommando, das in einer Datei steht, ein ELAN-Programm, das Datenobjekte auf dem Bildschirm zeigt. Dafür muß ein Rahmenprogramm in der Datei "join mac" existieren (fetch ("join mac",/"REMIS")).

Hinter dem Schlüsselwort SELECT werden die Merkmale aufgezählt, die ausgegeben werden sollen.
Nach dem Schlüsselwort FROM folgt die Angabe der Gruppe(n), deren Objekte betrachtet werden sollen.
Um die Merkmale eindeutig einem Objekt zuzuordnen, kann ihnen ein Gruppenbezeichner vorangestellt werden (Gruppenname oder ein im FROM-Teil vereinbarter Gruppenbezeichner).

Autoren aller in 'wgr' enthaltenen Werke

```
SELECT autor FROM wgr          oder
SELECT wgr_autor FROM wgr
```

Name aller Benutzer:

```
SELECT name FROM bgr : pers    oder
SELECT pers_name FROM bgr : pers
```

Außerdem kann, um das Ausgabebild zu beeinflussen, dem Merkmal noch eine Längenangabe zugefügt werden (Standardwertwert: 20).

name (max. 30 Stellen) und klasse (max. 6 Stellen) aller Bibliotheksbenutzer:

```
SELECT name L 30, klasse L 6 FROM bgr
```

Die Ergebnismenge kann durch Angabe von Auswahlbedingungen im WHERE – Teil der SELECT – Anweisung eingeschränkt werden.

Titel und Autor aller Bücher des Autors "May",
in deren Titel das Wort 'Winnetou' auftritt
und die spätestens 1892 erschienen sind:

```
SELECT titel, autor FROM wgr : buch
WHERE buch_autor = "May"
  AND "Winnetou" IN buch_titel
  AND buch_erschj <= 1892
```

Anmerkung: Im WHERE – Teil der Anweisung müssen alle Merkmale durch den Gruppenbezeichner spezifiziert werden, da dieser Teil vom ELAN – Compiler abgearbeitet wird, der keine automatischen Ergänzungen durchführt. Hierdurch kann aber der volle Elan – Sprachumfang zur Formulierung von Bedingungen genutzt werden, einschließlich (auch nachträglich) selbst definierter Operatoren. Für das Beispiel muß der Operator IN insertiert sein, z.Bsp.:

```
BOOL OP IN (TEXT CONST muster, text):
  pos (text, muster) > 0
END OP IN
```

Die Angabe mehrerer Gruppen im FROM – Teil führt zu einem JOIN, d.h. jedes Element der einen Gruppe wird mit jedem Element der anderen zu einem neuen Objekt gekoppelt.
Eine Gruppe kann auch mit sich selbst gekoppelt werden. Man erhält dann die Menge aller Kombination zweier Objekte dieser Gruppe.

Alle Autoren, die auch Benutzer sind:

```
SELECT name, klasse FROM wgr : werk, bgr : benutzer
WHERE werk_autor = benutzer_name
```

8. Weitere Funktionen

Datumsbearbeitung

Das Datumspaket stellt Prozeduren zur Verfügung, die einen TEXT, der der Syntax eines gültigen Datums zwischen 1.1.1900 und 31.12.2050 entspricht, eindeutig einen INTEGER zuweist. Dieser Bereich ist in der Regel für Datenbankanwendungen ausreichend (Geburtsdatum, Einstellungsdatum, Termine etc.).

Die Codierung hat gegenüber der Speicherung des Textes folgende Vorteile:

- Der Speicherplatzbedarf beträgt nur ein Achtel
- Die Vergleichsoperationen wirken wie gewünscht, während sie auf der TEXT – Darstellung gemäß alphabetischer Ordnung arbeiten und unsinnige Ergebnisse liefern.
- Operationen wie Bestimmen der Tagesdifferenz zwischen zwei Daten, Bestimmung des Wochentages, Berechnung des Folgedatums, Sortieren nach Datum etc. sind wesentlich schneller.

Beispiel:

```
datum ("28.01.84")              -->  - 1434
datum (- 1434)                  -->  "28.01.84"
datum (datum("28.2.84") + 1)    -->  "29.02.84"
datum (datum("01.3.83") - 1)    -->  "28.02.83"
```

Die durch dieses Paket definierten Prozeduren sind im Anhang aufgeführt.

Task – Kommunikation

REMIS enthält eigene Prozeduren für die Abwicklung der Inter – Task – Kommunikation, die zur Koordinierung von Programmen, die in verschiedenen Tasks laufen, genutzt werden können.

Andere Task rufen

`call (name der gerufenen task)`
Das Terminal wird an die gerufene Task übergeben. Die rufende Task wartet auf Rückgabe.

`call (taskname, PROC (code, datenraum) auftragsbearbeitung)`
Zusätzlich ist während der Wartezeit die Prozedur 'autragsbearbeitung' bereit, Aufträge der gerufenen Task zu bearbeiten, die durch einen code und einen (zu bearbeitenden) Datenraum beschrieben sind. Als Ergebnis wird ein Datenraum und in code eine Antwort zurückgegeben. Die Prozedur hat kein Terminal. Falls erforderlich muß sie es durch die parameterlose Prozedur 'continue' holen und durch 'return control' wieder abgeben.

Terminal abgeben

`return control`
Falls die Task mit 'call' gerufen wurde, wird das Terminal an die rufende Task zurückgegeben. Die Task selbst wartet, bis sie das Terminal wiederbekommt.

Aufträge an die rufende Task erteilen (z. Bsp. Fragen stellen)

`ask (auftragsart, datenraum, antwort)`
erteilt durch eine Zahl in auftragsart einen Auftrag an die rufende Task und erhält eine Antwort. Diese ist positiv, wenn sie von der Auftragsbearbeitung gegeben wurde, sonst negativ. Im Datenraum können Daten übergeben und übernommen werden.

Starten der Kommunikation

Eine Task, die durch 'call' aufgerufen werden soll, muß mit 'return control' anstelle von 'break' verlassen werden.

9. Mehrfachbenutzung

9.1 Überblick

Wenn eine Datenbank von mehreren Tasks gleichzeitig benutzt werden soll, dann ist die Multi – User – Version von REMIS zu verwenden. Sie arbeitet nach folgendem Konzept:

Es wird eine Task als Datenbankmanager eingerichtet, in der die Datenbank liegt. Jede Task eröffnet die Kommunikation mit dem DB – Manager durch

```
OPEN db-manager-name anstelle von 'load base'.
```

Anschließend können Lese – und Schreibtransaktionen auf der Datenbank ausgeführt werden. Dazu werden parameterlose Prozeduren zur Ausführung an REMIS übergeben, z. Bsp.

```
read transaction (PROC werke zeigen).
```

Anmerkung: 'load base' und 'save base' gibt es in der Multi – User – Version nicht mehr.

Gleichzeitig können mehrere Lesetransaktionen und eine Schreibtransaktion ausgeführt werden. Weitere Schreibtransaktionen müssen bis zum Ende der gerade laufenden warten.

Über die Transaktionen – Verwaltung hinaus enthält die Multi – User – Version von REMIS ein eigenes Archiv – System, das insbesondere die Archivierung von DB – Datenräumen, die nicht auf eine Floppy passen, ermöglicht sowie einige Prozeduren für die Inter – Task – Kommunikation.

9.2 REMIS starten

Anmerkung: Bei der Generierung des Datenbaksystems wird eine Task "REMIS" eingerichtet. Falls die schon existiert (z. Bsp. Single – User – Version) muß sie zunächst umbenannt werden, z. Bsp.

```
continue ("REMIS")
rename myself ("REMIS single")
break
```

Task – Struktur

PUBLIC
- DB – UR
  - Datenbank – Manager
  - REMIS
    - Arbeitstask 1
    - Arbeitstask 2

und bei Bedarf:

SYSUR
- DB – SYSUR
  - DB – ARCHIVE
  - shutup

Die Tasks DB – UR, REMIS und DB – SYSUR, DB – ARCHIVE, shutup werden vom Generator selbständig eingerichtet.

REMIS installieren

```
begin ("xxx")
                                  Archiv – Floppy einlegen
archive ("REMISm")                archiv anwählen
fetch all(archive)
edit ("REMIS.info")               Information lesen
ESC q                             edit verlassen
run                               "REMIS.info" ausführen
```

richtet DB – UR und REMIS ein und insertiert REMIS – Pakete

Archiv – System installieren

```
begin ("xxx","SYSUR")
fetch ("REMIS.sys",/"REMIS")
insert
```

Datenbank – Manager einrichten

```
begin ("BIBM","DB-UR")
fetch all                         leere Datenbasis holen
```

hier kann auch eine andere durch 'save base' erzeugte Datenbasis verwendet werden

```
multi user base                   Datenbasis einrichten und Manager
                                  starten
```

Arbeitstask einrichten

```
begin ("bib","REMIS")
OPEN "BIBM"                       Kommunikation mit BIBM eröffnen
```

Nun können Transaktionen auf der Datenbank ausgeführt werden.

9.3 Bearbeiten der Datenbasis

Alle Arbeiten mit der Datenbasis müssen als Transaktion ausgeführt werden, z. Bsp.

```
read transaction (PROC dblist)
```

oder, nachdem das Programm auf S. 9 mit 'edit' in eine Datei geschrieben wurde

```
read transaction (PROC run).
```

Das Bibliothekssystem kann in einfacher Weise durch Änderung des Menue – Baums (S. 41) auf die Mehrbenutzer – Version umgestellt werden.
Beispiel:

```
l  "n: Titel aufnehmen"                tr titel aufnehmen
l  "ä: Titel ändern"                   tr titel aendern
l  "w: weitere Exemplare aufnehmen"    tr weitere exemplare
l  "a: Ausleihe"                       tr ausleihe
l  "r: Rückgabe"                       tr rueckgabe
l  "l: Exemplar löschen"               tr exemplar loeschen
l  "b: Benutzeranzeige"                tr benutzeranzeige
l  "t: Titelanzeige"                   tr titelanzeige
l  "z: zeige Liste"
?  Welche Liste?
2  "w: Werkliste"                      tr werke zeigen
2  "e: Exemplarliste"                  tr exemplare zeigen
2  "b: Benutzerliste"                  tr benutzer zeigen
l  "q: Ende"                           break
l  "i: Initialisierung"                tr bibinit
//*
tr titel aufnehmen:   write transaction (PROC titel aufnehmen).
tr titel aendern:     write transaction (PROC titel aendern).
tr weitere exemplare: write transaction (PROC weitere exemplare).
tr ausleihe:          write transaction (PROC ausleihe).
tr rueckgabe:         write transaction (PROC rueckgabe).
tr exemplar loeschen: write transaction (PROC exemplar loeschen).
tr benutzeranzeige:   read transaction (PROC benutzeranzeige).
```

```
tr titelanzeige:      read transaction (PROC titelanzeige).
tr werke zeigen:      read transaction (PROC werke zeigen).
tr exemplare zeigen:  read transaction (PROC exemplare zeigen).
tr benutzer zeigen:   read transaction (PROC benutzer zeigen).
tr bibinit:           write transaction (PROC bibinit).
```

Anmerkung: Bei den Lesetransaktionen 'benutzeranzeige' und 'titelanzeige' wird am Ende ein Fehler auftreten ("Schreibzugriff in Lesetransaktion nicht erlaubt").
Erklärung: 'benutzeranzeige' enthält die Anweisung

```
read (benutzer,entliehen,exgr).
```

Dabei wird der Wert von 'entliehen' gelesen und auf den in der Datenbasis liegenden Platz von 'exgr' geschrieben. Die Lese – Anweisung enthält also auch eine Schreib – Operation.
Der Fehler kann vermieden werden, indem 'benutzeranzeige' als Schreibtransaktion ausgeführt wird, oder abgefangen werden, indem die Lesetransaktion in geeigneter Weise mit 'disable stop', 'clear error' und 'enable stop' umgeben wird.

9.4 Programmieren von Transaktionen

Der DB – Manager muß gestartet sein durch

```
multi user base.
```

In einer Arbeitstask wird die Kommunikation mit dem DB – Manager eröffnet durch

```
OPEN db-manager-name
```

und geschlossen durch

```
kill base.
```

Prozeduren, die auf der Datenbasis ausgeführt werden sollen, sind durch

```
read transaction (PROC prozedur)  bzw.
write transaction (PROC prozedur)
```

auszuführen. Wenn die Prozeduren Parameter enthalten, müssen sie in parameterlose Prozeduren verpackt werden.

Lesetransaktionen dürfen keine Schreibzugriffe enthalten. Dies wird bei Transaktionsende überprüft. Wurden Schreibzugriffe ausgeführt, dann wird die Transaktion beendet mit

```
errorstop ("Schreibzugriff in Lesetransaktion nicht erlaubt").
```

Schreibtransaktionen dürfen nicht innerhalb von Lesetransaktionen ausgeführt werden.

Außerhalb von Transaktionen ist der Zustand der Datenbasis in der Arbeitstask nicht definiert (und ebenso das Ergebnis von DB – Operationen).

Zwischen Transaktionen kann die Datenbasis durch Schreibtransaktionen anderer Tasks geändert werden. Insbesondere kann der Inhalt von Gruppen verändert sein und Referenzen im Programm können ungültig sein. Variable für Werte aus der Datenbasis (insbesondere REF – Variable) müssen daher in jeder Transaktion neu besetzt werden.

Für den, der's genauer wissen will:

Um den Aufwand bei Transaktionsanfang und -ende zu verringern wurde das Systemverhalten optimiert.

Transaktionen aus einer Task werden von REMIS zusammengefaßt bis zum fehlerhaften Abbruch einer Transaktion oder der Behandlung einer Schreibtransaktion aus einer anderen Task. Innerhalb einer Transaktionenfolge wird der Zustand der Datenbasis zwischen den Transaktionen nicht verändert. Die Programmvariablen müssen daher nur in jeder Transaktionenfolge neu besetzt werden.
Ob eine neue Transaktionenfolge aufgesetzt wird bzw. wurde, kann festgestellt werden aus dem Bestehen eines Fehlerzustands ('is error') nach Transaktionsende bzw. aus der Änderung der DB-Versionsnummer. Diese wird jeweils bei Ende einer Schreibtransaktion geändert. Sie wird geliefert durch die Prozedur

```
trversion.
```

9.5 Datenbasis archivieren

Durch die Prozedur

```
archive operation (welche, dbmanager-name)
```

kann die Archiv – Task mit der Druchführung einer Archiv – Funktion beauftragt werden. Folgende Operationen können ausgeführt werden:

welche

1 Floppies formatieren
2 Anzeige des Inhalts einer Floppy.
3 Datenbasis vom Archiv laden.
4 Datenbasis archivieren.

Datenbasis archivieren

Der Benutzer wird aufgefordert (formatierte!) Floppies einzulegen, bis die Datenbasis vollständig archiviert ist oder der Vorgang abgebrochen wird.
Eine unvollständige Archivierung ist wertlos.
Große Datenräume werden ggfs. auf mehrere Floppies verteilt.

Datenbasis vom Archiv laden

Der Benutzer wird aufgefordert Floppies einzulegen bis eine vollständige Datenbasis eingelesen ist. Die zu einer Kopie gehörenden Floppies können in beliebiger Reihenfolge eingelegt werden.

Um die Gefahr eines Speicherüberlaufs zu vermeiden, wird der Benutzer evtl. zu Beginn gefragt, ob die aktuelle Datenbasis vor dem Einlesen der Kopie gelöscht werden soll.

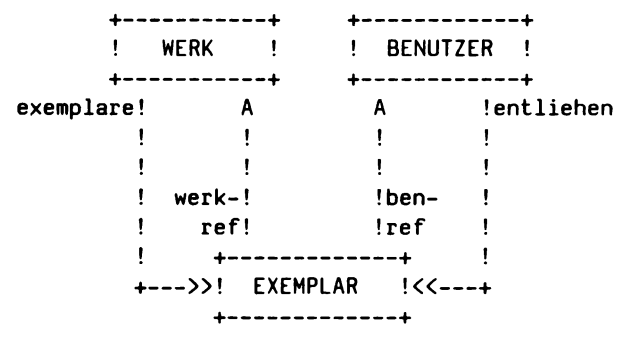
10. Anhang

10.1 Beispiel Schulbibliothek

Am folgenden Beispiel soll die Verwendung von REMIS gezeigt werden.
Das Programm stellt für die Verwaltung einer Schulbibliothek folgende Dialogfunktionen zur Verfügung:

1 Eingabe von Buchtiteln
2 Änderung von Titeln
3 Eingabe von Zusatzexemplaren
4 Ausleihe
5 Rückgabe
6 Anzeige aller Exemplare eines Titels
7 Anzeige aller ausgeliehenen Bücher eines Benutzers
8 Anzeige aller Werke, Exemplare, Benutzer
in alphabetischer Reihenfolge ab einer bestimmten Stelle

Die Datenbank enthält Daten folgender Struktur:



--> zeigt auf ein Objekt, -->> zeigt auf mehrere Objekte

WERK hat folgende Segmente (Felder, Attribute):

autor	Text	Autor
titel	Text	Titel
verlag	Text	Verlag
ejahr	Zahl (INT)	Erscheinungsjahr
exemplare	Gruppe	zugehörige Exemplare

BENUTZER hat folgende Segmente:

name	Text	Nachname
vorname	Text	Vorname
klasse	TEXT	Klasse bzw. Benutzergruppe
entliehen	Gruppe	entliehene Exemplare

EXEMPLAR hat folgende Segmente:

exbez	Text	Exemplarbezeichnung
adat	Zahl (INT)	Ausleihdatum
werkref	REF	Verweis auf Titel
benref	REF	Verweis auf Ausleiher

Datendefinition
(wird in einer Datei erstellt und mit 'run' ausgeführt)

```
new class ("WERK");
new class ("EXEMPLAR");
new class ("BENUTZER");

GROUP VAR
   wgr    := new group ("WERK","wgr"),              (* alle Werke *)
   bgr    := new group ("BENUTZER","bgr"),          (* alle Benutzer *)
   egr    := new group ("EXEMPLAR","egr");          (* alle Exemplare *)
(* Beispielgruppen *)
   exgr   := new group ("EXEMPLAR","exemplare"); (* Exemplare eine Werks *)
   entgr := new group ("EXEMPLAR","entliehen"); (* Ausleihem eines Benutzers *)

(* WERK - Segmente *)
   new segment ("autor",          "WERK",      "");
   new segment ("titel",          "WERK",      "");
   new segment ("verlag",         "WERK",      "");
   new segment ("ejahr",          "WERK",      0);
   new segment ("exemplare", "WERK",           exgr);

(* EXEMPLAR - Segmente *)
   new segment ("exbez",          "EXEMPLAR", "");
   new segment ("adat",           "EXEMPLAR", nildatum);
   new segment ("werkref",        "EXEMPLAR", "WERK",     nilref);
   new segment ("benref",         "EXEMPLAR", "BENUTZER", nilref);

(* BENUTZER - Segmente *)
   new segment ("name",           "BENUTZER", "");
   new segment ("vorname",        "BENUTZER", "");
   new segment ("klasse",         "BENUTZER", "");
   new segment ("entliehen", "BENUTZER", entgr);

all class segments (wgr);
all class segments (egr);
all class segments (bgr);

new methode (SEG "adat", 2);                  (* Datums - Konversion *)
```

dblist zeigt folgende DB – Struktur:

Stand: 04.07.84 19:47 TASK : bib
Es gibt die folgenden Objektklassen :

CONVERSION
WERK
EXEMPLAR
BENUTZER

Strukturen der Datenbankobjekte :

CONVERSION – Objekte bestehen aus den folgenden Segmenten :
INT tablenr
INT intcode
TEXT textcode

WERK – Objekte bestehen aus den folgenden Segmenten :
TEXT autor
TEXT titel
TEXT verlag
INT ejahr
GROUP exemplare

EXEMPLAR – Objekte bestehen aus den folgenden Segmenten :
TEXT exbez
INT adat
REF werkref
REF benref

BENUTZER – Objekte bestehen aus den folgenden Segmenten :
TEXT name
TEXT vorname
TEXT klasse
GROUP entliehen

Folgende Gruppen sind permanent in der Datenbank :

conversiontable enthaelt 1 CONVERSION – Objekte. Substruktur :
- tablenr
- intcode
- textcode

wgr enthaelt 0 WERK – Objekte. Substruktur :
- autor
- titel
- verlag
- ejahr
- exemplare

egr enthaelt 0 EXEMPLAR – Objekte. Substruktur :
- exbez
- adat
- werkref
- benref

bgr enthaelt 0 BENUTZER – Objekte. Substruktur :
- name
- vorname
- klasse
- entliehen

exemplare enthaelt 0 EXEMPLAR – Objekte. Substruktur :

entliehen enthaelt 0 EXEMPLAR – Objekte. Substruktur :

Folgende Formulare wurden definiert:

morchelrahmen (geändert, für Statusanzeigen und Meldungen)

```
 +-------------------------------------------------------------+
 1!feld 2                              feld 3                  !
 2!feld 4 und feld 5 (4711)                                    !
 3!-------------------------------------------------------------!
```

werk (Werkanzeige, Standardformular für wgr)

```
 4!Autor:  autor_____________________________________________!
 5!Titel:  titel_____________________________________________!
 6!__________________________________________________________!
 7!Verlag: verlag____________________________________________!
 8!Erscheinungsjahr: 4...                                      !
                     4: ejahr
```

exemplar (Exemplar – Anzeige, Standardformular für egr)

```
10!Exemplarbezeichnung: exbez_____________                     !
```

ausleihe (Ausleiher – Anzeige, Standardformular für bgr)

```
12!Name:     name_____________________                         !
13!Vorname:  vorname__________________                         !
14!Klasse:   klasse___________________                         !
```

Anwendungssystem

Das Anwendungssystem wird mit
archive ("REMIS.bib")
fetch all (archive)
run ("bibstart")
gestartet.

```
PACKET bib DEFINES
  titel aufnehmen,
  titel aendern,
  weitere exemplare,
  ausleihe,
  rueckgabe,
  exemplar loeschen,
  benutzeranzeige,
  titelanzeige,
  werke zeigen,
  exemplare zeigen,
  benutzer zeigen,
  bibinit:
```

```
TEXT VAR s:="", salt;
INT VAR adatum, i, erste zeile;
BOOL VAR erfolg:=FALSE;
GROUP VAR exgr, egr, wgr, bgr;
REF VAR werk, exemplar, ex, benutzer, bref, wref;
TAG VAR tagst, tagw, tage, taga;

PROC bibinit:
  exgr:=arbeitsgruppe ("exgroup", "EXEMPLAR");
  egr:=group ("egr"); sort (egr, 1);
  wgr:=group ("wgr"); sort (wgr, 2);
  bgr:=group ("bgr"); sort (bgr, 2);
  werk:=first (wgr);
  exemplar:=first (egr);
  benutzer:=first (bgr);
  tagst INITBY "morchelrahmen";
  tagw INITBY "werk";
  tage INITBY "exemplar";
  taga INITBY "ausleihe";
  erste zeile:=4
END PROC bibinit;
```

```
PROC titel aufnehmen:
  werk:=first (wgr);
  exemplar:=first (egr);
  REP werk aufnehmen
  UNTIL NOT erfolg PER;
  sort (wgr, 2).

werk aufnehmen:
  meldung ("Daten oder ESC");
  show (tagw);
  insert and reset (werk);
  get (werk);
  IF   aufnahme ok
  THEN exemplare aufnehmen
  ELSE erfolg:=FALSE; delete (werk)
  FI.
aufnahme ok: (werk_titel) > "".
exemplare aufnehmen:
  clear (exgr);
  exemplar aufnehmen;
  IF   erfolg
  THEN REP exemplar aufnehmen
       UNTIL NOT erfolg PER;
       write (werk, exemplare, exgr);
       erfolg:=TRUE
  ELSE delete (werk);
       meldung ("ohne Exemplarbezeichnung keine Aufnahme");
       pause (100)
  FI
END PROC titel aufnehmen;
```

```
PROC titel aendern:
  werk:=first (wgr);
  show (tagw);
  REP meldung ("");
      titel suchen;
      IF   erfolg CAND ja ("diesen Titel ändern")
      THEN putget (werk) FI
  UNTIL nein ("noch ein Titel") PER;
  sort (wgr, 2)
END PROC titel aendern;

PROC weitere exemplare:
  werk:=first (wgr);
  exemplar:=first (egr);
  show (tagw);
  titel suchen;
  IF   erfolg
  THEN read (werk, exemplare, exgr);
       REP exemplar aufnehmen;
       UNTIL NOT erfolg PER;
       write (werk, exemplare, exgr)
  FI
END PROC weitere exemplare;
```

```
PROC ausleihe:
  werk:=first (wgr);
  exemplar:=first (egr);
  benutzer:=first (bgr);
  show (taga);
  adatum:=datum (date);
  benutzer suchen;
  IF NOT erfolg COR nein ("an diesen Benutzer ausleihen")
  THEN IF   ja ("Benutzer aufnehmen")
       THEN benutzer aufnehmen
       ELSE erfolg:=FALSE
       FI
  FI;
  IF erfolg THEN ausleihen FI.
ausleihen:
  show (tagw); show (tage);
  read (benutzer, entliehen, exgr);
  REP exemplar suchen;
      IF   erfolg CAND ja ("Dieses Buch ausleihen")
      THEN ruecknahme; buchen FI
  UNTIL NOT erfolg PER;
  write (benutzer, entliehen, exgr).
buchen:
  hold (exgr, exemplar);
  write (exemplar, benref, benutzer);
  write (exemplar, adat, adatum).
benutzer aufnehmen:
  insert and reset (benutzer);
  write (benutzer, name, s);
  putget (benutzer);
  IF   aufnahme ok
  THEN sort (bgr, 2);
       erfolg:=TRUE
  ELSE erfolg:=FALSE;delete (benutzer)
  FI.
aufnahme ok:  (benutzer_name) > ""
END PROC ausleihe;
```

```
PROC rueckgabe:
  exemplar:=first (egr);
  show (tagw); show (tage);
  REP exemplar suchen;
      IF   erfolg CAND ja ("Dieses Buch zurücknehmen")
      THEN ruecknahme FI
  UNTIL nein ("noch eine Rückgabe") PER
END PROC rueckgabe;

PROC exemplar loeschen:
  exemplar:=first (egr);
  show (tagw); show (tage);
  exemplar suchen;
  IF   erfolg
  THEN IF   ausgeliehen
       THEN meldung ("nicht möglich, ausgeliehen"); pause (100)
       ELIF ja ("Dieses Exemplar löschen")
       THEN loeschen
       FI
  FI.
ausgeliehen: another (exemplar_benref).

loeschen:
  read (exemplar, werkref, wref);
  delete (exemplar);
  werk ohne exemplar loeschen.

werk ohne exemplar loeschen:
  read (wref, exemplare, exgr);
  IF groupcount (exgr) = 0 THEN delete (wref) FI
END PROC exemplar loeschen;
```

```
PROC benutzeranzeige:
  frage antwort ("Benutzername oder q");
  IF   s <> "q"
  THEN benutzer:=first (bgr);
       first (benutzer, name = s);
       WHILE another (benutzer)
       REP benutzer anzeigen;
           next (benutzer)
       UNTIL nein ("nächster Benutzer") PER
  FI.
benutzer anzeigen:
  putze ab zeile (erste zeile);
  put (benutzer_name);
  put (benutzer_vorname);
  putline (benutzer_klasse);
  read (benutzer, entliehen, exgr);
  ex:=first (exgr);
  WHILE another (ex)
  REP read (ex, werkref, wref);
      put (ex_exbez);
      out (wref_autor); out (": ");
      out (wref_titel); out ("; ");
      out (datum (ex_adat));line;
      next (ex)
  PER
END PROC benutzeranzeige;
```

```
PROC titelanzeige:
  frage antwort ("Autor oder q");
  IF   s <> "q"
  THEN werk:=first (wgr);
       first (werk, autor = s);
       WHILE another (werk)
       REP titel anzeigen;
           next (werk);
       UNTIL nein ("nächster Titel") PER
  FI.
titel anzeigen:
  putze ab zeile (erste zeile);
  out (werk_autor); out (": ");
  out (werk_titel); line;
  read (werk, exemplare, exgr);
  ex:=first (exgr); i:=0;
  WHILE another (ex)
  REP naechste zeile;
      put (ex_exbez);
      read (ex, benref, bref);
      IF   another (bref)
      THEN put (bref_name);
           put (bref_klasse);
           out (datum (ex_adat));
      FI; line;
      next (ex)
  PER.
naechste zeile:
  IF   i > 15
  THEN IF   no ("weiter")
       THEN LEAVE titelanzeige
       ELSE i:=1;
            putze ab zeile (erste zeile + 1)
       FI
  ELSE i INCR 1
  FI
END PROC titelanzeige;
```

```
PROC werke zeigen:
  werk:=first (wgr);
  frage antwort ("Anhalten mit bel. Taste; Bitte Autor");
  first (werk, autor=s);
  WHILE another (werk)
  REP zeigen; next (werk) PER;
  put ("*** ENDE ***"); pause (1000).
zeigen:
  put (werk_autor);
  putline (werk_titel);
  IF   incharety > ""
  THEN IF   no ("weiter")
       THEN LEAVE werke zeigen
       ELSE eine zeile hoch FI
  FI
END PROC werke zeigen;

PROC exemplare zeigen:
  exemplar:=first (egr);
  frage antwort ("Anhalten mit bel. Taste; Bitte Exemplarbez.");
  first (exemplar, exbez=s);
  WHILE another (exemplar)
  REP zeigen; next (exemplar) PER;
  put ("*** ENDE ***"); pause (1000).
zeigen:
  put (exemplar_exbez);
  putline (exemplar_werkref_titel);
  IF   incharety > ""
  THEN IF   no ("weiter")
       THEN LEAVE exemplare zeigen
       ELSE eine zeile hoch FI
  FI
END PROC exemplare zeigen;
```

```
PROC benutzer zeigen:
  benutzer:=first (bgr);
  frage antwort ("Anhalten mit bel. Taste; Bitte Benutzername");
  first (benutzer, name=s);
  WHILE another (benutzer)
  REP zeigen; next (benutzer) PER;
  put ("*** ENDE ***"); pause (1000).
zeigen:
  put (benutzer_name);
  put (benutzer_vorname);
  putline (benutzer_klasse);
  IF   incharety > ""
  THEN IF   no ("weiter")
       THEN LEAVE benutzer zeigen
       ELSE eine zeile hoch FI
  FI
END PROC benutzer zeigen;

PROC benutzer suchen:
  erfolg:=FALSE;
  meldung ("Benutzername oder ESC");
  s:="";
  putget (taga, s, namensfeld);
  IF   s > ""
  THEN meldung ("");
       first (benutzer, name = s);
       WHILE NOT erfolg AND another (benutzer)
       REP put (benutzer);
           IF   nein ("nächster Benutzer")
           THEN erfolg:=TRUE
           ELSE next (benutzer)
           FI
       PER
  FI.
namensfeld: field with name (taga, SEG name)
END PROC benutzer suchen;
```

```
PROC exemplar suchen:
  erfolg:=FALSE;
  meldung ("Exemplarbezeichnung oder ESC");
  s:=""; putget (tage, s, 1);
  IF   s > ""
  THEN meldung ("");
       first (exemplar, exbez = s);
       IF   found
       THEN put (exemplar_werkref);
            erfolg:=TRUE
       ELSE meldung ("Exemplar nicht gefunden"); pause (100)
       FI
  FI
END PROC exemplar suchen;

PROC titel suchen:
  erfolg:=FALSE;
  meldung ("Bitte Autor oder ESC");
  s:=""; putget (tagw, s, 1);
  IF   s > ""
  THEN meldung ("");
       first (werk, autor = s);
       WHILE NOT erfolg AND another (werk)
       REP put (werk);
           IF   nein ("nächster Titel")
           THEN erfolg:=TRUE
           ELSE next (werk)
           FI
       PER;
       IF NOT erfolg
       THEN meldung ("Titel nicht gefunden"); pause (100)
       FI
  FI
END PROC titel suchen;
```

```
PROC exemplar aufnehmen:
  meldung ("Daten oder ESC");
  show (tage);
  erfolg:=FALSE;
  salt:=""; s:=salt;
  REP putget (tage, s, 1);
      IF   s = salt
      THEN LEAVE exemplar aufnehmen FI;
      first (exemplar, exbez = s);
      IF   found
      THEN meldung ("Exemplar vorhanden, bitte ändern oder ESC");
           salt:=s
      ELSE insert (exemplar);
           erfolg:=TRUE
      FI
  UNTIL erfolg PER;
  hold (exgr, exemplar);
  exemplarwerte eintragen.
exemplarwerte eintragen:
  write (exemplar, exbez, s);
  reset (exemplar, adat);
  reset (exemplar, benref);
  write (exemplar, werkref, werk)
END PROC exemplar aufnehmen;
```

```
PROC ruecknahme:
  read (exemplar, benref, bref);
  IF   another (bref)
  THEN bei benutzer abbuchen;
       reset (exemplar, adat);
       reset (exemplar, benref)
  FI.
bei benutzer abbuchen:
  GROUP VAR g:=new group("EXEMPLAR");
  read (bref, entliehen, g);
  ex:=first (g);
  first (ex, exemplar);
  IF   another (ex)
  THEN unhold (ex);
       write (bref, entliehen, exgr)
  FI; forget (g)
END PROC ruecknahme;

GROUP PROC arbeitsgruppe (TEXT CONST groupname, classname):
  IF   group exists (groupname)
  THEN group (groupname)
  ELSE new group (classname, groupname)
  FI
END PROC arbeitsgruppe;

PROC insert and reset (REF VAR r):
  BOOL VAR vorhanden;
  SEGMENT VAR seg;
  insert (r);
  begin segment list (r);
  next segment (seg, vorhanden);
  WHILE vorhanden
  REP reset (r, seg);
      next segment (seg, vorhanden)
  PER
END PROC insert and reset;
```

```
PROC putze ab zeile (INT CONST z):
  cursor (1, z); out (""4"")
END PROC putze ab zeile;

PROC meldung (TEXT CONST t):
  put (tagst, t, 4)
END PROC meldung;

BOOL PROC ja (TEXT CONST t):
  out (""1""10""5""); yes (t)
END PROC ja;

BOOL PROC nein (TEXT CONST t):
  out (""1""10""5""); no (t)
END PROC nein;

PROC frage antwort (TEXT CONST fragetext):
  cursor (1, 2); put (fragetext); get (s);
  cursor (1, erste zeile)
END PROC frage antwort;

bibinit.
eine zeile hoch: out (""3"")
END PACKET bib
```

10.2 REMIS – Funktionen nach Gruppen

Bearbeitung der Datenbank

base exists - - > BOOL
kill base
load base (TASK CONST manager task)
load base
save base (TASK CONST manager task)
save base

Bearbeitung von Klassen

class exists (TEXT CONST name) - - > BOOL
clear (TEXT CONST classname)
delete class (TEXT CONST name)
delete segment (TEXT CONST segmentname, classname)
new class (TEXT CONST name)
new segment (TEXT CONST segmentname, classname, INT CONST wert)
new segment (TEXT CONST segmentname, classname, REAL CONST wert)
new segment (TEXT CONST segmentname, classname, wert)
new segment (TEXT CONST segmentname, classname, targetclassname,
REF CONST wert)
new segment (TEXT CONST segname, classname, GROUP VAR g)

Bearbeitung von Gruppen

TYPE GROUP

:= (GROUP VAR a, GROUP CONST b)

add group segment (GROUP VAR g, SEGMENT CONST segment)
all class segments (GROUP VAR g)
class of (GROUP CONST g) - - > TEXT
clear (GROUP VAR g)
copy of (GROUP CONST g) - - > GROUP
exists (GROUP CONST g) - - > BOOL
forget (GROUP VAR g)
group (TEXT CONST name) - - > GROUP
group exists (TEXT CONST name) - - > BOOL
groupcount (GROUP CONST g) - - > INT
keep group (GROUP CONST g, TEXT CONST name)
new group (TEXT CONST classname) - - > GROUP
new group (TEXT CONST classname, groupname) - - > GROUP
no group segment (GROUP CONST g)
reorganize grouptable
reset sorted (GROUP CONST g)
sort (GROUP CONST g, INT CONST n)
sorted (GROUP CONST g) - - > INT

Bearbeitung von Objekten

delete (REF VAR r)
hold (GROUP VAR g, REF CONST r)
hold (REF VAR wohin, REF CONST r)
insert (REF VAR r)
unhold (REF VAR r)

Bearbeitung von Referenzen

TYPE REF

:= (REF VAR a, REF CONST b)
= (REF CONST a, REF CONST b) - - > BOOL

another (REF CONST r) - - > BOOL
first (REF VAR r)
first (REF VAR a, REF CONST b)
first (REF VAR r, segment = konstante)
first (GROUP CONST g) - - > REF
found - - > BOOL
group (REF CONST r) - - > GROUP
is ref of (REF CONST r, GROUP CONST g) - - > BOOL
is ref of (REF CONST r, TEXT CONST classname) - - > BOOL
last (REF VAR r)
last (GROUP CONST g) - - > REF
next (REF VAR r)
next (REF VAR r, segment = konstante)
nilref - - > REF
prior (REF VAR r)

Bearbeitung von Segmenten

TYPE GROUPSEGMENT
TYPE INTSEGMENT
TYPE REALSEGMENT
TYPE REFSEGMENT
TYPE SEGMENT
TYPE TEXTSEGMENT

_ (REF CONST r, INTSEGMENT CONST s) - - > INT
_ (REF CONST r, REALSEGMENT CONST s) - - > REAL
_ (REF CONST r, REFSEGMENT CONST s) - - > REF
_ (REF CONST r, SEGMENT CONST s) - - > TEXT
_ (REF CONST r, TEXTSEGMENT CONST s) - - > TEXT

```
:= (INTSEGMENT VAR a, SEGMENT CONST b)
:= (REALSEGMENT VAR a, SEGMENT CONST b)
:= (TEXTSEGMENT VAR a, SEGMENT CONST b)
:= (REFSEGMENT VAR a, SEGMENT CONST b)
:= (GROUPSEGMENT VAR a, SEGMENT CONST b)
:= (SEGMENT VAR a, SEGMENT CONST b)
= (SEGMENT CONST a, b) - - > BOOL
SEG (TEXT CONST name) - - > SEGMENT
SEG (INTSEGMENT CONST s) - - > SEGMENT
SEG (REALSEGMENT CONST s) - - > SEGMENT
SEG (TEXTSEGMENT CONST s) - - > SEGMENT
SEG (REFSEGMENT CONST s) - - > SEGMENT
SEG (GROUPSEGMENT CONST s) - - > SEGMENT

clear (SEGMENT CONST s)
generate declarations
insert declarations
is segment of (SEGMENT CONST s, GROUP CONST g) - - > BOOL
is segment of (SEGMENT CONST s, REF CONST r) - - > BOOL
is segment of (SEGMENT CONST s, TEXT CONST classname) - - > BOOL
last conversion done - - > BOOL
read (REF CONST r, SEGMENT CONST s, TEXT VAR w)
read (REF CONST r, INTSEGMENT CONST s, INT VAR wert)
read (REF CONST r, REALSEGMENT CONST s, REAL VAR wert)
read (REF CONST r, TEXTSEGMENT CONST s, TEXT VAR wert)
read (REF CONST r, REFSEGMENT CONST s, REF VAR wert)
read (REF CONST r, GROUPSEGMENT CONST s, GROUP VAR wert)
reset (REF CONST r, SEGMENT CONST s)
segment (TEXT CONST name) - - > SEGMENT
segment (TEXT CONST segmentname, classname) - - > SEGMENT
segment exists (TEXT CONST name) - - > BOOL
segment name (SEGMENT CONST s) - - > TEXT
valid value written - - > BOOL
write (REF CONST r, SEGMENT CONST s, TEXT CONST w)
write (REF CONST r, INTSEGMENT CONST s, INT CONST wert)
write (REF CONST r, REALSEGMENT CONST s, REAL CONST wert)
write (REF CONST r, TEXTSEGMENT CONST s, TEXT CONST wert)
write (REF CONST r, REFSEGMENT CONST s, REF CONST wert)
write (REF CONST r, GROUPSEGMENT CONST s, GROUP CONST wert)
```

Datenbank – Verwaltung

```
aktual mode  - - >  INT
begin classlist
begin grouplist
begin segmentlist (TEXT CONST class)
begin segmentlist (GROUP CONST g)
begin segmentlist (REF CONST r)
convert by (SEGMENT CONST conversion methode)
db list (TEXT CONST dateiname)
db list
init conversion handling
new conversion table entry (SEGMENT CONST s, TEXT CONST wert,
                                    INT CONST code)
new methode (SEGMENT CONST s, INT CONST mode)
next class list entry (TEXT VAR name)
next group list entry (TEXT VAR groupname, classname)
next segment (SEGMENT VAR s, BOOL VAR gefunden)
reorganize (SEGMENT CONST segment)
segment type (SEGMENT CONST s)  - - >  INT
segment type (SEGMENT CONST s, TEXTSEGMENT VAR t,
                  INTSEGMENT VAR i, REALSEGMENT VAR r,
                  REFSEGMENT VAR rr, GROUPSEGMENT VAR g)  - - >  INT
type text (INT CONST segmenttyp)  - - >  TEXT
version  - - >  INT
```

Sonstige DB – Prozeduren

```
pack (TEXT CONST wert, INT VAR code, BOOL VAR okay)
pack (TEXT CONST wert, INT VAR code)
unpack (TEXT VAR wert, INT CONST code, BOOL VAR okay)
unpack (TEXT VAR wert, INT CONST code)
```

Verwendung von Formularen

TYPE TAG

:= (TAG VAR a, TAG CONST b)
INITBY (TAG VAR t, TEXT CONST name)
SCROLL (TAG VAR t, INT CONST n)

auskunfts nr (TAG CONST t, INT CONST feldnr) - - > INT
auskunfts nr - - > INT
execute command code (TAG CONST t, INT VAR feldnr)
field exists (TAG CONST t, INT CONST feldnr) - - > BOOL
field with name (TAG CONST t, INT CONST name) - - > INT
field with name (TAG CONST t, SEGMENT CONST s) - - > INT
fields (TAG CONST t) - - > INT
first field (TAG CONST t) - - > INT
form line (TAG CONST t, INT CONST n) - - > TEXT
get (TAG CONST t, TEXT VAR wert, INT CONST feldnr)
get (TAG CONST t, ROW 100 TEXT VAR werte, INT VAR feldnr)
get (TAG CONST t, INT CONST feldnr) - - > TEXT
leaving code - - > INT
length (TAG CONST t, INT CONST feldnr) - - > INT
next field (TAG CONST t, INT CONST feldnr) - - > INT
prior field (TAG CONST t, INT CONST feldnr) - - > INT
protect (TAG VAR t, INT CONST feldnr, BOOL CONST schreibschutz)
protected (TAG CONST t, INT CONST feldnr) - - > BOOL
put (TAG CONST t, TEXT CONST wert, INT CONST feldnr)
put (TAG CONST t, ROW 100 TEXT VAR werte)
putget (TAG CONST t, TEXT VAR wert, INT CONST feldnr)
putget (TAG CONST t, ROW 100 TEXT VAR werte, INT VAR feldnr)
reorganize screen
set field infos (TAG VAR t, INT CONST feldnr,
BOOL CONST closed, protected, secret)
set last edit values
show (TAG CONST t)
symbolic name (TAG CONST t, INT CONST feldnr) - - > INT
tag (TEXT CONST name) - - > TAG
tag (REF CONST r) - - > TAG

tag exists (TEXT CONST name) - - > BOOL
x size (TAG CONST t) - - > INT
y size (TAG CONST t) - - > INT

Verwendung von Formularen mit Datenbankdaten

get (REF VAR r, TAG CONST t)
get (REF VAR r, TAG CONST t, INT VAR feldnr)
get (REF VAR r)
link tag to group (GROUP VAR g, TEXT CONST tagname)
put (REF CONST r, TAG CONST t)
put (REF CONST r)
put (TAG CONST t, INT CONST symb name, TEXT CONST wert)
put (TAG CONST t, SEGMENT CONST s, TEXT CONST wert)
putget (REF VAR r)
putget (REF VAR r, TAG CONST t)
putget (REF VAR r, TAG CONST t, INT VAR feldnr)
show (REF CONST r)

Verwendung von Formularen für Ausgabe

CLEARBY (TAG VAR ziel, TAG CONST quelle)

clear board
fill (TAG VAR t, TEXT CONST wert, INT CONST feldnr)
get board (FILE VAR datei)
next boardline - - > INT
pin (TAG CONST t, INT CONST spalte,zeile)
pin (TAG CONST t, INT CONST spalte)
pin (TAG CONST t)
pin (TAG CONST t, TEXT CONST symbol)
pin (TAG CONST t, TEXT CONST symbol, BOOL VAR erfolg)
pin (TEXT CONST text, INT CONST spalte,zeile)
put board
put board (FILE VAR datei)
set boardline (INT CONST zahl)

Ausgabe von Datenbankdaten

FILLBY (TAG VAR t, REF CONST r)
GENBY (TAG VAR t, REF VAR r)

fill (TAG VAR t, SEGMENT CONST s, TEXT CONST wert)
fill (TAG VAR t, INT CONST feldname, TEXT CONST wert)

Formular – Verwaltung

TO (TAG CONST t, FILE VAR datei)
TO (FILE VAR datei, TAG VAR t)

begin taglist
clear field (TAG VAR t, INT CONST feldnr)
copy tag (TEXT CONST alt, neu)
design (TAG VAR t)
design fields (TAG VAR t)
design form (TAG VAR t)
forget tag (TEXT CONST name)
list tags
next tag list entry (TEXT VAR name)
nil (TAG VAR t)
rename tag (TEXT CONST alt, neu)
restore tag (TAG CONST t, TEXT CONST name)
store tag (TAG CONST t, TEXT CONST name)
trans (TEXT CONST pseudoblank inversan inversaus)
transform (TAG CONST t, FILE VAR datei)
transform (FILE VAR datei, TAG VAR t)

Menue – Generator

clear menue
menuecursor (TEXT CONST t)
minimenue
next screen
put fehler (TEXT CONST msg)
put old commands
return symbol (TEXT CONST z)
return symbol – – > TEXT
return to father
screen version – – > INT
transform (TEXT CONST name)

Multi – User DB – Manager

multi user base

Multi – User Arbeitstask

OPEN (TEXT CONST dbmanager name)

read transaction (PROC prozedur)
write transaction (PROC prozedur)

sql

sql (TEXT CONST dateiname)

Task – Kommunikation

ask (INT CONST order, DATASPACE VAR ds, INT VAR returncode)
call (TEXT CONST taskname)
call (TEXT CONST taskname,
    PROC (INT VAR code, DATASPACE VAR ds) prozedur)
continue
return control

Arbeiten mit Datum

datum (TEXT CONST d) - - > INT
datum (INT CONST t,m,j) - - > INT
datum (INT CONST d) - - > TEXT
datumjh (INT CONST d) - - > TEXT
jahr (INT CONST d) - - > INT
jahrestag (INT CONST d) - - > INT
monat (INT CONST d) - - > INT
nildatum - - > INT
tag (INT CONST d) - - > INT
tmj (INT CONST d, INT VAR t,m,j)
wochentag (INT CONST d) - - > INT

10.3 REMIS – Funktionen alphabetisch

GROUP
TYPE GROUP
Zweck: interner GROUP – Bezeichner

GROUPSEGMENT
TYPE GROUPSEGMENT
Zweck: interner GROUPSEGMENT – Bezeichner

INTSEGMENT
TYPE INTSEGMENT
Zweck: interner INTSEGMENT – Bezeichner

REALSEGMENT
TYPE REALSEGMENT
Zweck: interner REALSEGMENT – Bezeichner

REF
TYPE REF
Zweck: Referenz; enthält internen GROUP – Bezeichner, Position in der Gruppe, Referenz auf Objekt, internen Klassen – Bezeichner, Referenz auf Standardformular

REFSEGMENT
TYPE REFSEGMENT
Zweck: interner REFSEGMENT – Bezeichner

SEGMENT
TYPE SEGMENT
Zweck: interner SEGMENT – Bezeichner

TAG
TYPE TAG
Zweck: Datentyp für ein Formular

TEXTSEGMENT
TYPE TEXTSEGMENT
Zweck: interner TEXTSEGMENT – Bezeichner

_ (Unterstrich)

INT OP _ (REF CONST r, INTSEGMENT CONST s)
REAL OP _ (REF CONST r, REALSEGMENT CONST s)
REF OP _ (REF CONST r, REFSEGMENT CONST s)
TEXT OP _ (REF CONST r, TEXTSEGMENT CONST s)
Zweck: liefert den Wert des Segments s des durch r referierten Objekts; Warnung: s muß zur referierten Klasse gehören, sonst falsches Ergebnis
Fehler: Wrong segment type: <typ>

TEXT OP _ (REF CONST r, SEGMENT CONST s)
Zweck: liefert den Wert des Segments s des durch r referierten Objekts, konvertiert nach der s zugeordneten Methode
Fehler: keine Textumwandlung für diesen Typ (REF – oder GROUP-SEGMENT)

:=

OP := (INTSEGMENT VAR a, SEGMENT CONST b)
OP := (REALSEGMENT VAR a, SEGMENT CONST b)
OP := (TEXTSEGMENT VAR a, SEGMENT CONST b)
OP := (REFSEGMENT VAR a, SEGMENT CONST b)
OP := (GROUPSEGMENT VAR a, SEGMENT CONST b)
Zweck: Zuweisung mit Typkontrolle; falls möglich bezeichnen anschliessend a und b dasselbe Segment
Fehler: segmenttyp ist nicht INT (oder nicht vorhanden)
segmenttyp ist nicht REAL (oder nicht vorhanden)
segmenttyp ist nicht TEXT (oder nicht vorhanden)
segmenttyp ist nicht REF (oder nicht vorhanden)
segmenttyp ist nicht GROUP (oder nicht vorhanden oder keine gültige Beispielgruppe)

OP := (SEGMENT VAR a, SEGMENT CONST b)
Zweck: Zuweisung; a und b bezeichnen anschließend dasselbe Segment (oder nichts)

OP := (GROUP VAR a, GROUP CONST b)
Zweck: Nach der Zuweisung bezeichnen a und b dieselbe Gruppe (oder nichts)

OP := (REF VAR a, REF CONST b)
Zweck: nach der Zuweisung referieren a und b dasselbe Objekt in derselben Gruppe

OP := (TAG VAR a, TAG CONST b)
Zweck: b wird auf a übertragen

=

BOOL OP = (SEGMENT CONST a, b)
Zweck: liefert TRUE, wenn a und b dasselbe Segment bezeichnen

BOOL OP = (REF CONST a, REF CONST b)
Zweck: liefert TRUE, wenn a und b dasselbe Objekt referieren; a und b brauchen nicht zur selben Gruppe zu gehören.

CLEARBY

OP CLEARBY (TAG VAR ziel, TAG CONST quelle)
Zweck: überschreibt alle (ymax (ziel)) Zeilen in ziel mit den ersten (ymax (ziel)) Zeilen von quelle;
sinnvoll für: schnelles Leeren eines mit Daten gefüllten Formulars aus einer aufbewahrten leeren Kopie

FILLBY

OP FILLBY (TAG VAR t, REF CONST r)
Zweck: trägt die Werte des durch r referierten Objekts in t ein

GENBY

OP GENBY (TAG VAR t, REF VAR r)
Zweck: erzeugt in t das zu r gehörende Formular und trägt die Werte des durch r referierten Objekts ein

INITBY

OP INITBY (TAG VAR t, TEXT CONST name)
Zweck: erzeugt in t das Formular name
Fehler: TAG <name> unbekannt

OPEN

OP OPEN (TEXT CONST dbmanager name)

Zweck: eröffnet die Kommunikation mit dem Datenbank – Manager; (schließen mit 'kill base')

Fehler: Datenbank "<dbmanager name>" noch offen
Keine DB – Manager – Task
DB – Version zu alt

SCROLL

OP SCROLL (TAG VAR t, INT CONST n)

Zweck: verschiebt t um n Zeilen auf dem Bildschirm (n > 0: nach unten, n < 0: nach oben); t kann mit den neuen Positionen auch gespeichert werden

Fehler: Feld ausserhalb Bildschirm durch SCROLL

SEG

SEGMENT OP SEG (TEXT CONST name)

Zweck: liefert den internen Bezeichner des existierenden Segments name

Fehler: segment not existing: <name>

SEGMENT OP SEG (INTSEGMENT CONST s)
SEGMENT OP SEG (REALSEGMENT CONST s)
SEGMENT OP SEG (TEXTSEGMENT CONST s)
SEGMENT OP SEG (REFSEGMENT CONST s)
SEGMENT OP SEG (GROUPSEGMENT CONST s)

Zweck: liefert den internen Bezeichner von s

TO

OP TO (TAG CONST t, FILE VAR datei)

Zweck: überträgt das Bild von t in editierbarer Form (wie von 'design form' angeboten) in die Textdatei datei

OP TO (FILE VAR datei, TAG VAR t)

Zweck: erzeugt in t ein Formular aus dem in den ersten (maximal 24) Sätzen von datei abgelegten Formularbild

add group segment

PROC add group segment (GROUP VAR g, SEGMENT CONST segment)

Zweck: erweitert die Gruppe g um segment, falls es noch nicht dazugehört

Fehler: Segment gehört nicht zur Class

aktual mode

INT PROC aktual mode

Zweck: liefert die über 'convert by' eingestellte Konversionsmethode

all class segments

PROC all class segments (GROUP VAR g)

Zweck: ordnet g alle noch nicht zu g gehörenden Segmente der zugehörigen Klasse zu in der Reihenfolge, in der sie zu der Klasse hinzugefügt wurden

another

BOOL PROC another (REF CONST r)

Zweck: liefert TRUE, wenn r ein gültiges Objekt referiert

ask

PROC ask (INT CONST order, DATASPACE VAR ds,
INT VAR returncode)

Zweck: fragt mit order und ds und Terminalübergabe bei der aufrufenden Task nach und erwartet eine Antwort in ds und returncode

auskunfts nr

INT PROC auskunfts nr (TAG CONST t, INT CONST feldnr)

Zweck: liefert die zum Feld feldnr in t gehörende Auskunftsnummer

INT PROC auskunfts nr

Zweck: liefert die aktuelle Auskunftsnummer (wird beim Verlassen einer Maske durch ESC gesetzt)

base exists

BOOL PROC base exists

Zweck: liefert TRUE, wenn eine Bank in der Task existiert

begin classlist

PROC begin classlist

Zweck: initialisiert die Liste der Objektklassen

begin grouplist

PROC begin grouplist

Zweck: initialisiert die Liste der Gruppen

begin segmentlist

PROC begin segmentlist (TEXT CONST class)

Zweck: initialisiert die Liste der Segmente von class

PROC begin segmentlist (GROUP CONST g)

Zweck: initialisiert die Liste der Segmente von g

PROC begin segmentlist (REF CONST r)

Zweck: initialisiert die Liste der Segmente der zu r gehörenden Gruppe oder Klasse

begin taglist

PROC begin taglist

Zweck: initialisiert die Lister der Formular – Definitionen

call

PROC call (TEXT CONST taskname)

Zweck: übergibt die Kontrolle und das Terminal an Task taskname

Fehler: Task "<taskname>" wird benutzt (anwortet nicht)

PROC call (TEXT CONST taskname,
PROC (INT VAR code, DATASPACE VAR ds) prozedur)

Zweck: übergibt Kontrolle und Terminal an Task taskname; bei Rückkehr wird prozedur mit code und ds ausgeführt, wobei in code ein ordercode übergeben wird und ein returncode zurückzugeben ist. Falls prozedur das Terminal braucht, muß sie es durch 'continue' besorgen und am Ende durch 'return control' zurückgeben

Fehler: Task "<taskname>" wird benutzt (antwortet nicht)

class exists

BOOL PROC class exists (TEXT CONST name)

Zweck: liefert TRUE, wenn eine Klasse name vorhanden ist

class of

TEXT PROC class of (GROUP CONST g)

Zweck: liefert den Namen der zu g gehörenden Klasse

Fehler: subscript underflow (wenn g keine gültige Gruppe bezeichnet)

clear

PROC clear (TEXT CONST classname)

Zweck: löscht alle Objekte der Klasse classname

PROC clear (SEGMENT CONST s)

Zweck: löscht den Inhalt von s in allen Objekten

PROC clear (GROUP VAR g)

Zweck: löscht in g alle Referenzen (nicht die referierten Objekte)

Fehler: undefined group (falls g keine gültige Gruppe bezeichnet)

clear board

PROC clear board

Zweck: erzeugt ein leeres Pinboard

clear field

PROC clear field (TAG VAR t, INT CONST feldnr)

Zweck: löscht Feld feldnr aus Formular t

clear menue

PROC clear menue

Zweck: löscht den durch das Menue – Formular belegten Bildschirmbereich

continue

PROC continue

Zweck: holt das durch 'call' abgegebene Terminal zurück

convert by

PROC convert by (SEGMENT CONST s)

Zweck: schaltet die zu s gehörende Konversionsmethode ein; alle nachfolgenden Konvertierungen werden entsprechend durchgeführt

copy of

GROUP PROC copy of (GROUP CONST g)

Zweck: erzeugt und liefert eine Kopie von g, falls möglich

Fehler: grouptable overflow

copy tag

PROC copy tag (TEXT CONST alt, neu)

Zweck: erzeugt von der Formulardefinition alt eine Kopie mit dem Namen neu, falls möglich

Fehler: TAG <alt> unbekannt
tag already existing

datum

INT PROC datum (TEXT CONST datumstext)

Zweck: liefert den Datumscode für datumstext (in der Form tt.mm.jj)

INT PROC datum (INT CONST t, m, j)

Zweck: liefert den Datumscode

TEXT PROC datum (INT CONST d)

Zweck: liefert den Datumstext (in der Form tt.mm.jj) für Datumscode d

datumjh

TEXT PROC datumjh (INT CONST d)

Zweck: liefert den Datumstext mit Jahrhundert (in der Form tt.mm.jjjj)

dblist

PROC db list

Zweck: generiert ein Inhaltsverzeichnis der vorhandenen Datenbank in die Textdatei 'DB – list', zeigt die Liste und löscht sie nach Wunsch

PROC dblist (TEXT CONST dateiname)

Zweck: generiert ein Inhaltsverzeichnis der vorhandenen Datenbank in die Textdatei dateiname

delete

PROC delete (REF VAR r)

Zweck: löscht das durch r referierte Objekt aus seiner Klasse und damit aus allen Gruppen; anschließend referiert r das Objekt, das (vor dem Löschen) mit 'next (r)' erreicht worden wäre

delete class

PROC delete class (TEXT CONST name)

Zweck: loescht die Klasse name

delete segment

PROC delete segment (TEXT CONST segmentname, classname)

Zweck: löscht das Segment segmentname bei der Klasse classname (und allen ihren Objekten)

Fehler: class <classname> not existing.
Warnung: <segmentname> ist nicht in <classname>

design

PROC design (TAG VAR t)

Zweck: bietet t zum interaktiven design an; ruft design form und design fields, letzteres jedoch ohne symbolische Feldnamen (bzw. Segmentzuordnung)

design fields

PROC design fields (TAG VAR t)

Zweck: erlaubt interaktives Erzeugen, Ändern und Löschen von Datenfeld – Definitionen für t mit Segmentzuordnung und Feldeigenschaften (geschützt, Standardanzeige)

design form

PROC design form (TAG VAR t)

Zweck: erlaubt die interaktive Gestaltung des Bildes von t; wenn t nicht existiert, wird ein leeres TAG erzeugt

execute command code

PROC execute command code (TAG CONST t, INT VAR feldnr)

Zweck: interpretiert 'leaving code' (s.u.) und setzt feldnr auf einen entsprechenden neuen Wert

exists

BOOL PROC exists (GROUP CONST g)

Zweck: liefert TRUE, wenn g eine Gruppe bezeichnet, sonst FALSE

fields

INT PROC fields (TAG CONST t)

Zweck: liefert die höchste Feldnummer, die in t irgendwann existierte
Warnung: nicht zu jeder Nummer muß ein Feld existieren

field exists

BOOL PROC field exists (TAG CONST t, INT CONST feldnr)

Zweck: liefert TRUE, wenn Feld feldnr in t existiert, sonst FALSE

field with name

INT PROC field with name (TAG CONST t, INT CONST name)

Zweck: liefert die Nummer des Feldes in t mit dem symbolischen Namen name oder 0, falls es nicht existiert

INT PROC field with name (TAG CONST t, SEGMENT CONST s)

Zweck: liefert die Nummer des Feldes des in t zu s gehörenden Feldes

fill

PROC fill (TAG VAR t, TEXT CONST wert, INT CONST feldnr)

Zweck: füllt Feld feldnr in t mit wert

PROC fill (TAG VAR t, SEGMENT CONST s, TEXT CONST wert)

Zweck: trägt wert in das s zugeordnete Feld von t ein

PROC fill (TAG VAR t, INT CONST feldname, TEXT CONST wert)

Zweck: trägt wert in das Feld feldname von t ein

first

PROC first (REF VAR r)

Zweck: setzt r auf das erste der von der assoziierten Gruppe referierten Objekte zurück

PROC first (REF VAR a, REF CONST b)
Zweck: positioniert a auf den Gruppeneintrag, der das gleiche Objekt referiert wie b, oder auf Gruppenende; a und b brauchen nicht zur selben Gruppe zu gehören
bisheriger Name: next
Fehler: REFs of different classes

PROC first (REF VAR r, segment = konstante)
Zweck: positioniert r auf das erste Objekt, dessen Segment segment (Bezeichner eines TEXTSEGMENTs oder INTSEGMENTs) den Wert konstante hat. Wird kein solches Objekt gefunden, dann zeigt r bei sortierten Gruppen auf das nächstgrößere Objekt, bei nicht sortierten Gruppen hinter das letzte Objekt

REF PROC first (GROUP CONST g)
Zweck: liefert eine Referenz auf das erste der von g referierten Objekte

first field
INT PROC first field (TAG CONST t)
Zweck: liefert die Nummer des ersten Feldes von t

forget
PROC forget (GROUP VAR g)
Zweck: löscht die Gruppe g (falls permanent nach Anfrage); g bezeichnet anschließend keine Gruppe

forget tag
PROC forget tag (TEXT CONST name)
Zweck: löscht TAG name
Fehler: TAG <name> unbekannt

form line
TEXT PROC form line (TAG CONST t, INT CONST n)
Zweck: liefert Zeile n von t

found
BOOL PROC found
Zweck: liefert TRUE, wenn das letzte first oder next mit Suchkriterium erfolgreich war, sonst FALSE

generate declarations

PROC generate declarations

Zweck: erzeugt für die vorhandene Datenbank einen Deklarationsteil und eine Prozedur, in die ein Programm einkopiert werden kann, um die Datenbankdeklarationen ohne Insertierung in dem Programm zur Verfügung zu haben

get

PROC get (TAG CONST t, TEXT VAR wert, INT CONST feldnr)

Zweck: liest über Feld feldnr von t einen Text in die Variable wert

PROC get (TAG CONST t, ROW 100 TEXT VAR werte, INT VAR feldnr)

Zweck: bietet werte auf t zum Editieren an, ohne sie vorher auszugeben (sie können vorher ausgegeben worden sein (z.B. durch put)); feldnr bezeichnet das Feld von t, in das die Schreibmarke am Anfang gesetzt werden soll, bzw. in dem sie sich am Ende befand (Verlassen nur mit ESC)

Fehler: startfeld nicht im tag

PROC get (REF VAR r, TAG CONST t)

Zweck: liest die Werte für das durch r referierte Objekt über t ein

PROC get (REF VAR r, TAG CONST t, INT VAR feldnr)

Zweck: liest die Werte für das durch r referierte Objekt über t ein; feldnr bezeichnet das Feld , in das die Schreibmarke am Anfang gesetzt werden soll, bzw. in dem sie sich am Ende befand

Fehler: Einstiegsfeld gesperrt

PROC get (REF VAR r)

Zweck: liest Werte für das durch r referierte Objekt über das zugehörige Formular ein

Fehler: Kein freies Feld zugeordnet

TEXT PROC get (TAG CONST t, INT CONST feldnr)

Zweck: liest über das Feld feldnr von t einen Text ein und liefert ihn

get board

PROC get board (FILE VAR datei)

Zweck: liest das Pinboard aus Datei datei

group

GROUP PROC group (TEXT CONST name)

Zweck: liefert die permanente Gruppe name

Fehler: group does not exist

GROUP PROC group (REF CONST r)

Zweck: liefert die zu r gehörende Gruppe; ist undefiniert, wenn r aus einem Segment gelesen wurde

groupcount

INT PROC groupcount (GROUP CONST g)

Zweck: liefert die Anzahl der Objekte in g

group exists

BOOL PROC group exists (TEXT CONST name)

Zweck: liefert TRUE, wenn die permanente Gruppe name vorhanden ist, sonst FALSE

hold

PROC hold (GROUP VAR g, REF CONST r)

Zweck: hängt r an g; r und g müssen Objekte derselben Klasse referieren (wird nicht geprüft); das Sortierkennzeichen wird auf 'nicht sortiert' gesetzt. Ungültige Referenzen in g werden gelöscht.

PROC hold (REF VAR wohin, REF CONST r)

Zweck: fügt r vor wohin in die zu wohin gehörende Gruppe ein; r und wohin müssen Objekte derselben Klasse referieren (wird nicht geprüft); anschließend gilt wohin = r; das Sortierkennzeichen bleibt erhalten. Ungültige Referenzen werden gelöscht.

init conversion handling

PROC init conversion handling

Zweck: erzeugt (falls nicht bereits vorhanden) die CLASS 'CONVERSION', die die Informationen enthält, welche Segmente nach welchem Modus verschlüsselt sind, und die Code – Tabellen für im table – mode verschlüsselte Segmente

insert

PROC insert (REF VAR r)

Zweck: erzeugt ein neues Objekt in der zu group (r) gehörenden Klasse und fügt eine Referenz darauf in group (r) ein vor der Position der aktuellen Referenz r; anschließend referiert r das neue Objekt; das Sortierkennzeichen bleibt erhalten, d.h. der Programmierer ist verantwortlich für die Reihenfolge in der Gruppe. Ungültige Referenzen werden aus der Gruppe gelöscht.

Fehler: Zu viele Objekte in class <classname>

insert declarations

PROC insert declarations

Zweck: erzeugt und insertiert für die vorhandene Datenbank ein Paket, das für jedes Segment eine Konstante deklariert, die als Namen den des Segments hat und als Wert den internen Segmentbezeichner enthält

is ref of

BOOL PROC is ref of (REF CONST r, GROUP CONST g)

Zweck: liefert TRUE, wenn r mit g assoziiert ist

BOOL PROC is ref of (REF CONST r, TEXT CONST classname)

Zweck: liefert TRUE, wenn r ein Objekt der Klasse classname referiert

is segment of

BOOL PROC is segment of (SEGMENT CONST s, GROUP CONST g)

Zweck: liefert TRUE, wenn s zu g gehört

BOOL PROC is segment of (SEGMENT CONST s, REF CONST r)

Zweck: liefert TRUE, wenn s zu r gehört

BOOL PROC is segment of (SEGMENT CONST s, TEXT CONST clname)

Zweck: liefert TRUE, wenn s zu Klasse clname gehört

jahr

INT PROC jahr (INT CONST d)

Zweck: liefert das Jahr für den Datumscode d

jahrestag

INT PROC jahrestag (INT CONST d)

Zweck: liefert den Jahrestag für den Datumscode d (1.1. = 1 usw.)

keep group

PROC keep group (GROUP CONST g, TEXT CONST name)

Zweck: ordnet der von g bezeichneten Gruppe den Namen name zu

Fehler: duplicate groupname: <name>.
subscript underflow (wenn g keine Gruppe bezeichnet)

kill base

PROC kill base

Zweck: löscht die ganze Datenbank (z.B. vor Laden vom Manager)

last

PROC last (REF VAR r)

Zweck: setzt r auf das letzte der von der assoziierten Gruppe referierten Objekte

REF PROC last (GROUP CONST g)

Zweck: liefert eine Referenz auf das letzte der von g referierten Objekte

last conversion done

BOOL PROC last conversion done

Zweck: liefert TRUE, wenn die letzte Konvertierung möglich war, sonst FALSE

leaving code

INT PROC leaving code

Zweck: liefert den Code des Steuerzeichens, mit dem das letzte get oder putget vom Benutzer beendet wurde

bei Mehrfelderbearbeitung 27: ESC

bei Einfeldbearbeitung

0: auskunft (HOP ?)	13: feld vor (RETURN)
3: OBEN	19: feld rück (HOP RETURN)
9: TAB	27: ESC
10: UNTEN	

length
INT PROC length (TAG CONST t, INT CONST feldnr)
Zweck: liefert die Gesamtlänge des Feldes feldnr von t

link tag to group
PROC link tag to group (GROUP VAR g, TEXT CONST tagname)
Zweck: ordnet tagname der Gruppe g zu
Fehler: TAG <tagname> unbekannt

list tags
PROC list tags
Zweck: zeigt eine Liste der vorhandenen TAGs auf dem Bildschirm

load base (* nur single-user-Version *)
PROC load base (TASK CONST manager task)
Zweck: lädt die Datenbank von der manager task
Fehler: load impossible, base already existing;
DB-Version zu alt

PROC load base
Zweck: lädt die Datenbank von der Vatertask
Fehler: load impossible, base already existing;
DB-Version zu alt

menuecursor
PROC menuecursor (TEXT CONST t)
Zweck: ersetzt die Darstellung des Menue-Cursors (Standard: = = >)

minimenue
PROC minimenue
Zweck: zeigt das Minimenue (Kommandobuchstaben der nächsten Menue-Anzeige) im Feld 4712 der Rahmenmaske

monat
INT PROC monat (INT CONST d)
Zweck: liefert den Monat für den Datumscode d

multi user base

PROC multi user base

Zweck: startet den Datenbankmanager; eine Datenbasis muß vorhanden sein

new class

PROC new class (TEXT CONST name)

Zweck: richtet neue Klasse mit dem Namen name ein

Fehler: CLASS einrichten unmöglich (kein Platz mehr);
CLASS existiert schon

new conversion table entry

PROC new conversion table entry (SEGMENT CONST s,
TEXT CONST wert, INT CONST code)

Zweck: trägt in die zu s gehörende Konversionstabelle die Kombination wert – code ein

new group

GROUP PROC new group (TEXT CONST classname)

Zweck: erzeugt eine leere temporäre Gruppe (ohne Namen) zur Objektklasse classname

Fehler: grouptable overflow.
class <classname> not existing

GROUP PROC new group (TEXT CONST classname, groupname)

Zweck: erzeugt die leere permanente Gruppe groupname zur Objektklasse classname

Fehler: group table overflow.
class <classname> not existing.
duplicate groupname: <groupname>

new methode

PROC new methode (SEGMENT CONST s, INT CONST mode)

Zweck: erzeugt einen neuen Konversions – Methoden – Eintrag mit der Nummer s und dem Modus mode
0: Zahl – Text;
1: Code (INT) – Text;
2: INT – Datum (in der Form tt.mm.jj)

Fehler: Unknown mode: <mode>

new segment

PROC new segment (TEXT CONST segmentname, classname,
INT CONST wert)

Zweck: fügt zur Klasse classname ein neues INTSEGMENT mit dem Standardwert wert hinzu

Fehler: <segmentname> bereits vorhanden

PROC new segment (TEXT CONST segmentname, classname,
REAL CONST wert)

Zweck: fügt zur Klasse classname ein neues REALSEGMENT mit dem Standardwert wert hinzu

Fehler: <segmentname> bereits vorhanden

PROC new segment (TEXT CONST segmentname, classname, wert)

Zweck: fügt zur Klasse classname ein neues TEXTSEGMENT mit dem Standardwert wert hinzu

Fehler: <segmentname> bereits vorhanden

PROC new segment (TEXT CONST segmentname, classname, targetclassname, REF CONST wert)

Zweck: fügt zur Klasse classname ein neues REFSEGMENT segmentname hinzu, das REFs auf Objekte der Klasse targetclassname aufnehmen kann. Der Standardwert wert muß auf die Zielklasse (targetclass) oder keine Klasse (nilref) zeigen.

Fehler: <segmentname> bereits vorhanden;
Falscher Beispielref

PROC new segment (TEXT CONST segname, classname, GROUP VAR g)

Zweck: fügt zur Klasse classname ein neues GROUPSEGMENT hinzu mit den Eigenschaften von g. Der Name von g und segname müssen übereinstimmen. g darf keinem anderen Segment zugeordnet sein. g wird auf 'leer' gesetzt und sollte leer bleiben. Spätere Änderungen der Eigenschaften von g (z.B. zugehörige Segmente) gelten für alle Einträge in das Segment.

Fehler: <segname> bereits vorhanden.
Falsche Beispielgruppe

next

PROC next (REF VAR r)

Zweck: setzt r auf das nächste Objekt seiner Gruppe oder Gruppenende

PROC next (REF VAR r, segment = konstante)

Zweck: positioniert r weiter auf das nächste Objekt, dessen Segment segment (Bezeichner eines TEXTSEGMENTs oder INTSEGMENTs) den Wert konstante hat. Wird kein solches Objekt gefunden, dann zeigt r bei sortierten Gruppen auf das nächstgrößere Objekt, bei nicht sortierten Gruppen hinter das letzte Objekt

next boardline

INT PROC next boardline

Zweck: liefert die Zeilennummer nach der letzten nicht leeren Zeile auf dem Board (Wert des Zeilenmerkers)

next class list entry

PROC next class list entry (TEXT VAR name)

Zweck: liefert den nächsten Eintrag aus der Liste der Objektklassen. Ist kein weiterer Eintrag vorhanden, wird name auf niltext gesetzt.

next field

INT PROC next field (TAG CONST t, INT CONST feldnr)

Zweck: liefert die Nummer des nächsten Feldes nach Feld feldnr in t

next group list entry

PROC next group list entry (TEXT VAR groupname, classname)

Zweck: liefert den nächsten Eintrag aus der Liste der zu Klasse classname gehörenden Gruppen. Ist kein weiterer Eintrag vorhanden, werden groupname und classname auf niltext gesetzt.

next screen

PROC next screen

Zweck: erhöht die Bildschirm – Versions – Nummer um 1 (mod 32000)

next segment

PROC next segment (SEGMENT VAR s, BOOL VAR gefunden)

Zweck: liefert den nächsten Eintrag aus der aktuellen Segmentliste. Diese Liste wird durch 'begin segment list' initialisiert. Wenn ein Eintrag gefunden wurde, wird gefunden auf TRUE gesetzt, sonst FALSE.

next tag list entry

PROC next tag list entry (TEXT VAR name)

Zweck: liefert den nächsten Eintrag aus der aktuellen TAG – Liste; bei Listenende ist name leer

nil

PROC nil (TAG VAR t)

Zweck: erzeugt in t ein leeres Formular

nildatum

INT PROC nildatum

Zweck: liefert das leere Datum

nilref

REF PROC nilref

Zweck: liefert den leeren Zeiger

no group segment

PROC no group segment (GROUP CONST g)

Zweck: entfernt alle Segmentzuordnungen von g

pack

PROC pack (TEXT CONST wert, INT VAR code, BOOL VAR okay)

Zweck: konvertiert wert nach der eingestellten Konversionsmethode, liefert den Code in code und TRUE in okay genau dann, wenn die Konvertierung möglich war

PROC pack (TEXT CONST wert, INT VAR code)

Zweck: konvertiert wert nach der eingestellten Konversionsmethode, liefert den Code in code

pin

PROC pin (TAG CONST t, INT CONST spalte, zeile)
Zweck: heftet t mit der linken oberen Ecke auf das Pinboard an die Stelle (spalte, zeile)

PROC pin (TAG CONST t, INT CONST spalte)
Zweck: wirkt wie pin (t, spalte, next boardline)

PROC pin (TAG CONST t)
Zweck: wirkt wie pin (t, 1, next boardline)

PROC pin (TAG CONST t, TEXT CONST symbol)
Zweck: heftet t mit der linken oberen Ecke auf das erste Zeichen von symbol auf dem Pinboard

PROC pin (TAG CONST t, TEXT CONST symbol, BOOL VAR erfolg)
Zweck: heftet t mit der linken oberen Ecke auf das erste Zeichen von symbol auf dem Pinboard; wenn symbol gefunden wurde, ist erfolg TRUE, sonst FALSE

PROC pin (TEXT CONST text, INT CONST spalte, zeile)
Zweck: heftet text auf die Position (spalte, zeile) des Pinboards

prior

PROC prior (REF VAR r)
Zweck: setzt r auf das vorhergehende Objekt seiner Gruppe oder Gruppenende

prior field

INT PROC prior field (TAG CONST t, INT CONST feldnr)
Zweck: liefert die Nummer des Feldes vor Feld feldnr in t

protect

PROC protect (TAG VAR t, INT CONST feldnr, BOOL CONST schreibschutz)
Zweck: bei schreibschutz = TRUE wird das Feld feldnr in t gegen Schreiben geschützt, andernfalls wird es zum Schreiben geöffnet

protected

BOOL PROC protected (TAG CONST t, INT CONST feldnr)

Zweck: liefert TRUE, wenn Feld feldnr in t schreibgeschützt ist, sonst FALSE

put

PROC put (TAG CONST t, TEXT CONST wert, INT CONST feldnr)

Zweck: gibt wert auf dem Bildschirm in Feld feldnr von t aus; ggf. wird wert gekürzt

PROC put (TAG CONST t, ROW 100 TEXT VAR werte)

Zweck: gibt werte auf dem Bildschirm in den Feldern von t aus; ggf. werden einzelne werte gekürzt

PROC put (REF CONST r, TAG CONST t)

Zweck: gibt die Werte des durch r referierten Objekts auf t aus

PROC put (REF CONST r)

Zweck: gibt die Werte des durch r referierten Objekts auf dem zugeordneten Formular aus

PROC put (TAG CONST t, INT CONST symb name, TEXT CONST wert)

Zweck: gibt den wert auf dem Feld symb name von t aus

PROC put (TAG CONST t, SEGMENT CONST s, TEXT CONST wert)

Zweck: gibt den wert auf dem s zugeordneten Feld von t aus

putget

PROC putget (TAG CONST t, TEXT VAR wert, INT CONST feldnr)

Zweck: bietet wert auf dem Feld feldnr von t zum Editieren an

PROC putget (TAG CONST t, ROW 100 TEXT VAR werte,
INT VAR feldnr)

Zweck: bietet werte auf t zum Editieren an; ggf. werden werte bei der Anzeige gekürzt; feldnr bezeichnet das Feld von t, in das die Schreibmarke am Anfang gesetzt werden soll, bzw. in dem sie sich am Ende befand (Verlassen nur mit ESC)

Fehler: startfeld nicht im tag

PROC putget (REF VAR r)
Zweck: bietet die Werte des durch r referierten Objekts auf dem zugehörigen tag zum Editieren an

PROC putget (REF VAR r, TAG CONST t)
Zweck: bietet die Werte des durch r referierten Objekts auf t zum Editieren an

PROC putget (REF VAR r, TAG CONST t, INT VAR feldnr)
Zweck: bietet die Werte des durch r referierten Objekts auf t zum Editieren an; ggf. werden Werte bei der Anzeige gekürzt; feldnr bezeichnet das Feld von t, in das die Schreibmarke am Anfang gesetzt werden soll, bzw. in dem sie sich am Ende befand (Verlassen nur mit ESC)
Fehler: Einstiegsfeld gesperrt

put board
PROC put board
Zweck: kopiert board auf die aktuelle Ausgabeeinheit

PROC put board (FILE VAR datei)
Zweck: schreibt das Pinboard in die offene output – Datei datei

put fehler
PROC put fehler (TEXT CONST msg)
Zweck: gibt msg auf das Fehlerfeld der Rahmenmaske aus

put old commands
PROC put old commands
Zweck: rekonstruiert die Menue – Rahmenmaske auf dem Bildschirm

read
PROC read (REF CONST r, SEGMENT CONST s, TEXT VAR w)
Zweck: liest den Wert des Segments s des durch r referierten Objekts, konvertiert ihn nach der s zugeordneten Methode und liefert das Ergebnis in w
Fehler: keine Textumwandlung für diesen Typ (REF – oder GROUP-SEGMENT)

PROC read (REF CONST r, INTSEGMENT CONST s, INT VAR w)
PROC read (REF CONST r, REALSEGMENT CONST s, REAL VAR w)
PROC read (REF CONST r, TEXTSEGMENT CONST s, TEXT VAR w)
PROC read (REF CONST r, REFSEGMENT CONST s, REF VAR w)
Zweck: liest den Wert des Segments s des durch r referierten Objekts in die Variable w;
Warnung: s muß zur referierten Klasse gehören, sonst falsches Ergebnis
Fehler: Wrong segment type: <typ>

PROC read (REF CONST r, GROUPSEGMENT CONST s, GROUP VAR w)
Zweck: liest den Wert des Segments s des durch r referierten Objekts in die Variable w; die Gruppe wert hat die Eigenschaften der zu s gehörenden Beispielgruppe;
Warnung: s muß zur referierten Klasse gehören, sonst falsches Ergebnis
Fehler: Wrong segment type: <typ>

read transaction
PROC read transaction (PROC prozedur)
Zweck: führt prozedur auf einer Datenbank – Kopie als Lesetransaktion aus; Schreibzugriffe erzeugen bei Transaktionsende einen errorstop
Fehler: Schreibzugriffe während Lese – Transaktion

rename tag
PROC rename tag (TEXT CONST alt, neu)
Zweck: ersetzt den Namen von TAG alt durch neu
Fehler: TAG <alt> unbekannt

reorganize
PROC reorganize (SEGMENT CONST segment)
Zweck: reorganisiert den Speicherplatz für segment

reorganize grouptable
PROC reorganize grouptable
Zweck: löscht alle unbenutzten (nach forget) oder temporären Gruppen

reorganize screen

PROC reorganize screen

Zweck: rekonstruiert den Bildschirm aus dem Bildwiederholspeicher

reset

PROC reset (REF CONST r, SEGMENT CONST s)

Zweck: schreibt in das Segment s des durch r referierten Objekts den Standardwert

reset sorted

PROC reset sorted (GROUP CONST g)

Zweck: kennzeichnet eine Gruppe als 'nicht sortiert'

restore tag

PROC restore tag (TAG CONST t, TEXT CONST name)

Zweck: ersetzt TAG name durch t

Fehler: TAG <name> unbekannt

return control

PROC return control

Zweck: gibt Kontrolle und Terminal (an aufrufende Task) zurück

return symbol

PROC return symbol (TEXT CONST z)

Zweck: definiert z als das Zeichen, bei dessen Eingabe im Menuebaum zum Vater zurückverzweigt werden soll

TEXT PROC return symbol

Zweck: liefert das aktuelle Rückkehr – Zeichen

return to father

PROC return to father

Zweck: geht im Menuebaum zum Vater – Menue zurück; wird normalerweise vom Monitorrahmenprogramm nach jeder Ausführung eines Endknotens selbständig ausgeführt

save base (* nur sigle – user – Version *)
PROC save base (TASK CONST manager task)
Zweck: speichert die Datenbank beim Manager manager task
Fehler: Inkonsistente Sicherung wegen: <grund>

PROC save base
Zweck: speichert die Datenbank bei der Vater – Task
Fehler: Inkonsistente Sicherung wegen: <grund>

screen version
INT PROC screen version
Zweck: liefert die Bildschirm – Versions – Nummer

segment
SEGMENT PROC segment (TEXT CONST name)
Zweck: liefert den internen Bezeichner des existierenden Segments name
Fehler: segment not existing: <name>

SEGMENT PROC segment (TEXT CONST segmentname, classname)
Zweck: liefert das bereits existierende Segment segmentname der Klasse classname
Fehler: <segmentname> ist kein Segment von Objekten der Klasse <classname>

segment exists
BOOL PROC segment exists (TEXT CONST name)
Zweck: liefert TRUE, wenn Segment name existiert, sonst FALSE

segment name
TEXT PROC segment name (SEGMENT CONST s)
Zweck: liefert den Namen von s

segment type

INT PROC segment type (SEGMENT CONST s)

Zweck: liefert den Typ des Segments s

1: TEXT 4: REF
2: INT 5: GROUP
3: REAL 0: sonst

INT PROC segment type (SEGMENT CONST s, TEXTSEGMENT VAR t,
INTSEGMENT VAR i, REALSEGMENT VAR r,
REFSEGMENT VAR rr, GROUPSEGMENT VAR g)

Zweck: liefert den Typ des Segments s (s.o.) und den Segmentbezeichner in dem entsprechenden Parameter

set boardline

PROC set boardline (INT CONST zahl)

Zweck: setzt den Zeilenmerker auf zahl

set field infos

PROC set field infos (TAG VAR t, INT CONST feldnr,
BOOL CONST closed, protected, secret)

Zweck: setzt die angegebenen Eigenschaften für Feld feldnr von t;
z.Zt. wird nur protected und secret unterstützt

set last edit values

PROC set last edit values

Zweck: stellt für die nächste Bearbeitung eines Maskenfeldes (get, putget) die Situation (Cursor – Position) beim Verlassen der Maske wieder her

show

PROC show (TAG CONST t)

Zweck: zeigt t auf dem Bildschirm

PROC show (REF CONST r)

Zweck: zeigt zu r gehörendes Formular auf dem Bildschirm

sort

PROC sort (GROUP CONST g, INT CONST n)

Zweck: sortiert die Gruppe g nach den ersten n Segmenten (nur INT, REAL, TEXT)

Fehler: keine Ordnungsrelation für segmenttype;
Sortfehler

sorted

INT PROC sorted (GROUP CONST g)

Zweck: liefert 0, falls g 'unsortiert', sonst die Anzahl der Segmente, nach denen g sortiert ist

sql

PROC sql (TEXT CONST dateiname)

Zweck: übersetzt ein SQL – Programm aus der Textdatei dateiname, generiert daraus ein ELAN – Progamm und führt es aus; sql benötigt dazu ein Rahmenprogramm in einer Textdatei mit dem Namen 'join mac'

store tag

PROC store tag (TAG CONST t, TEXT CONST name)

Zweck: speichert t unter dem Namen name

Fehler: tag already existing

symbolic name

INT PROC symbolic name (TAG CONST t, INT CONST feldnr)

Zweck: liefert den symbolischen Namen des Feldes feldnr von t

tag

INT PROC tag (INT CONST d)

Zweck: liefert den Tag für den Datumscode d

TAG PROC tag (TEXT CONST name)

Zweck: liefert das Formular name

Fehler: TAG <name> unbekannt

TAG PROC tag (REF CONST r)

Zweck: liefert das zu r gehörende Formular

tag exists

BOOL PROC tag exists (TEXT CONST name)

Zweck: TRUE, wenn die Formulardefinition name existiert

tmj

PROC tmj (INT CONST d, INT VAR t, m, j)

Zweck: liefert Tag, Monat und Jahr in t, m, j für den Datumscode d

trans

PROC trans (TEXT CONST pseudoblank inversan inversaus)

Zweck: ersetzt die entsprechenden Standardzeichen für die Formulargestaltung (design form)

Fehler: falsche Eingabe (nicht 3 Zeichen)

transform

PROC transform (TAG CONST t, FILE VAR datei)

Zweck: schreibt Formular t in transformierter Form in die offene output – Datei datei

PROC transform (FILE VAR datei, TAG VAR t)

Zweck: erzeugt t aus der offenen input – Datei datei

PROC transform (TEXT CONST name)

Zweck: erzeugt aus der Textdatei 'source.name' das Monitorprogramm in die Textdatei 'monitor.name' und den Menuebaum in den Datenraum 'name'. Zur Generierungszeit muß die Textdatei 'MORCHEL std' mit dem Rahmen für das Monitorprogramm vorhanden sein. Außerdem werden die beiden Standardmasken 'morchelrahmen' und 'morchelmenue' verwendet. Zur Ausführung wird 'monitor.name' insertiert und mit 'monitor name' gestartet.

trversion

INT PROC trversion

Zweck: liefert die Versionsnummer der Datenbasis; wird bei jeder Schreibtransaktion geändert

type text

TEXT PROC type text (INT CONST segmenttyp)

Zweck: liefert die Textbezeichnung eines Segmenttyps

Fehler: unknown type nr

unhold

PROC unhold (REF VAR r)

Zweck: löscht die Referenz r aus der zugehörigen Gruppe; anschließend referiert r das Objekt, das (vor dem Löschen) mit 'next (r)' erreicht worden wäre

unpack

PROC unpack (TEXT VAR wert, INT CONST code, BOOL VAR okay)

Zweck: konvertiert code nach der eingestellten Konversionsmethode, liefert den Text in wert und TRUE in okay genau dann, wenn die Konvertierung möglich war

PROC unpack (TEXT VAR wert, INT CONST code)

Zweck: konvertiert code nach der eingestellten Konversionsmethode, liefert den Text in wert

valid value written

BOOL PROC valid value written

Zweck: liefert TRUE genau dann, wenn beim letzten 'write' mit Konvertierung ein im Sinne der angewendeten Konversionsmethode gültiger Wert geschrieben wurde

version

INT PROC version

Zweck: liefert die Versionsnummer des Datenbanksystems

wochentag

INT PROC wochentag (INT CONST d)

Zweck: liefert den Wochentag für den Datumscode d (So = 0, ..., Sa = 6)

write

PROC write (REF CONST r, SEGMENT CONST s, TEXT CONST w)

Zweck: konvertiert w nach der dem Segment s zugeordneten Methode und schreibt das Ergebnis in das Segment s des durch r referierten Objekts

Fehler: write with illegal type (REF – oder GROUPSEGMENT)

PROC write (REF CONST r, INTSEGMENT CONST s, INT CONST w)
PROC write (REF CONST r, REALSEGMENT CONST s, REAL CONST w)
PROC write (REF CONST r, TEXTSEGMENT CONST s, TEXT CONST w)
PROC write (REF CONST r, REFSEGMENT CONST s, REF CONST w)
PROC write (REF CONST r, GROUPSEGMENT CONST s, GROUP CONST w)

Zweck: schreibt w in das Segment s des durch r referierten Objekts;
Warnung: s muß zur referierten Klasse gehören, sonst falsches Ergebnis

Fehler: Wrong segment type: <typ>
ref of wrong object class (bei REFSEGMENT)

write transaction

PROC write transaction (PROC prozedur)

Zweck: führt prozedur auf einer Datenbank – Kopie als Schreibtransaktion aus; die geänderten Datenräume werden nach fehlerloser Ausführung zum Datenbank – Manager übertragen

Fehler: Schreib – Transaktion während Lese – Transaktion

x size

INT PROC x size (TAG CONST t)

Zweck: liefert die Breite (Anzahl der Spalten) von t

y size

INT PROC y size (TAG CONST t)

Zweck: liefert die Höhe (Anzahl der Zeilen) von t

10.4 Syntax des SELECT – Kommandos (sql)

erforderlich: Rahmenprogramm in Datei "join mac"

<Kommando> ::= <SELECT – teil> <FROM – teil> /
<SELECT – teil> <FROM – teil> <WHERE – teil>

<SELECT – teil> ::= "SELECT" <segmentliste>

<segmentliste> ::= <segmentid> / <segmentid> "," <segmentliste>

<segmentid> ::= <merkmal> / <merkmal> <outputform>

<merkmal> ::= <merkmalsname> /
<groupbezeichner> "_" <merkmalsname>

<merkmalsname> ::= Name eines Merkmals einer der im FROM – Teil angegebenen GROUPs.

<outputform> ::= "L" <ausgabelänge als ganze Zahl>

<FROM – teil> ::= "FROM" <groupliste>

<groupliste> ::= <groupid> / <groupid> "," <groupliste>

<groupid> ::= <groupname> /
<groupname> ":" <groupbezeichner>

<groupname> ::= Name einer in der DB vorhandenen GROUP

<WHERE – teil> ::= "WHERE" <boolscher ausdruck>

<boolscher ausdruck> ::= beliebiger boolscher ELAN – Ausdruck,
Merkmale müssen wie folgt angegeben werden:
<groupbezeichner> "_" <merkmalsname>