L3
Referenzhandbuch

Autor: Werner Metterhausen

ACCOMMODAT GmbH

Rathausallee 10
53757 Sankt Augustin
Telefon: (02241) 92530
Telefax: (02241) 206026

Inhaltsverzeichnis

1 Einführung

Das L3 Referenzhandbuch ist das zweite Handbuch zum Betriebssystem L3. Während das Benutzerhandbuch durchaus als Lesebuch angelegt ist, ist das Referenzhandbuch eher ein Nachschlagewerk. Es mag auch interessant sein, dieses Handbuch durchzublättern, um auf Neues zu stoßen, es ist jedoch nicht darauf angelegt, es von vorn bis hinten durchzulesen. Dieses Buch ist nicht frei von Vorwärtsverweisen, es werden teilweise Dinge erwähnt, deren Erklärung erst anschließenden Kapiteln folgt. Vor allem wird dieses Handbuch nur dann Hilfe sein, wenn es beim Lernen und bei der Arbeit neben dem Rechner liegt. Ein Betriebssystem vom Schlage des L3 ist zu komplex und detailreich, um zuerst zu lesen und dann zu arbeiten. Für einen Einstieg in die Handhabung und Programmierung des Betriebssystems sollten Sie dieses erste Kapitel lesen, der Rest des Buches möge dann bei Bedarf folgen.

Genauso wichtig wie das Buch selbst ist die beiliegende Programmdiskette. Auf ihr befinden sich Teile des L3 Systems im Quellcode. Ausgesucht sind aus dem Gesamtsystem einerseits Programme, die gut nachvollziehbar die Programmierung in ELAN demonstrieren, andererseits aber auch Programme, die Basis für anspruchsvollere Vorhaben sein mögen.

Schließlich noch eine Bitte an alle Leser: Geben Sie Laut! Nur Ihre Meinung zu Stärken und Schwächen dieses Handbuchs kann eine Weiterentwicklung zum Besseren veranlassen.

Die aktuelle Auflage des Handbuchs ist in Januar 1991 abgeschlossen. Der Stand der einzelnen Kapitel:

Kap. 11: 3/90

Kap. 2, 3.3, 7: 10/90

Kap. 3, 4, 5, 8, 9, 10, 12: 11/90

Kap. 6: 12/90

Kap. 1, 8, 13, 14: 1/91

1.1 Tips zur schnellen Übersicht

Das L3 Referenzhandbuch beschreibt die sicheren Leistungen des Betriebssystems L3. Sichere Leistungen sind diejenigen, die auch in zukünftigen Versionen des Systems in der hier beschriebenen Form verfügbar sein werden oder für die Portierungswerkzeuge bereitgestellt würden, falls in kommenden Versionen Änderungen vorgenommen würden.

Nach diesem einführenden Kapitel ist in Kapitel 2 einiges zum ELAN Compiler ausgeführt. Die Kapitel 3 bis 13 sind Nachschlageteil für den ELAN Programmierer, das Kapitel 14 erörtert allgemein die Schnittstellen von L3 als offenes Betriebssystem für Rechner mit dem Intel 80386/80486 Prozessor.

Beim Aufbau der »ELAN Teile« wurde versucht vom den einfachen Dingen, wie arithmetischen Operationen zum Schwierigen, etwa der Managerprogrammierung vorzuschreiten.

Von allgemeinem Interesse sind Kapitel 5, das sich mit den L3 Basistypen Datenraum und Task beschäftigt, Kapitel 7 zur Intertask-Kommunikation und eventuell auch das Kapitel 8, das versucht das Managerkonzept nahezubringen.

1.1.1 Diskette mit Beispielprogrammen

Die Diskette »ELAN Beispielprogramme« enthält eine größere Anzahl von Programmen aus dem L3 System im Quellcode. Es gibt weder in diesem Buch noch auf der Diskette Querverweise zwischen Stichworten im Handbuch und Programmen auf der Diskette. Stattdessen befindet sich auf der Diskette das Programm `grep`, mit dem leicht Stichworte in einzelnen Dateien oder Gruppen von Dateien gesucht werden können.

Der Inhalt dieser Diskette kann mit der Zeit variieren, sie wird mit kommenden Versionen aktualisiert.

1.2 Notation in diesem Buch

Die Schreibweise in diesem Buch ist nach folgenden Regeln gestaltet:

- Besondere Tasten werden durch das Tastensymbol   dargestellt.
- Prozedurnamen etc. werden durch **Fettdruck** hervorgehoben. Derart hervorgehobene Namen in Erklärungen anderer Prozeduren sind also als Querverweis aufzufassen.
- Falls im Text Parameter erwähnt werden, sind sie in 'Hochkommata' gesetzt.
- Äußerungen, etwa Umschreibungen, die in Anführungsstriche zu setzen wären, sind durch » « geklammert.
- Programme oder Programmstücke sind in `äquidistanter` Schrift gedruckt.

Die Zusammenstellung der L3 Systemleistungen ist im weitestem Sinne an den beteiligten Datentypen orientiert oder nach Themenkreisen sortiert. Das heißt, ein Kapitel beschreibt z.B. die im Zusammenhang mit TEXT Objekten relevanten Systemleistungen, also Prozeduren/Operatoren, deren Parameter vom Typ TEXT sind oder die Texte liefern, ein anderer Abschnitt beschreibt die Prozeduren, die in unmittelbarem Zusammenhang mit der Intertask Kommunikation stehen. Im zweiten Fall ist keine Ordnung nach auftretenden Datentypen möglich, die Zusammenstellung aber thematisch naheliegend.

Jedes Kapitel beginnt mit einer kurz gehaltenen Einführung. Anschließend folgt die Beschreibung der Prozeduren, Operatoren oder auch Systemcalls. Innerhalb eines Kapitels gibt es keine zwingend eingehaltene Reihenfolge. Während es manchmal ratsam schien, die Leistungen in alphabetischer Reihung zu ordnen, sind in anderen Abschnitten nichtalphabetische Zusammenstellungen passender, weil sie eher der tatsächlichen Programmierung entsprechen.

Die Beschreibung einer einzelnen Komponente kann folgende Teile enthalten:

PROZEDUR

PROC PROZEDUR (PARAMETER ...)

Wirkung: Mindestens eine kurze Beschreibung zu Sinn und Zweck ist immer vorhanden.

Bemerkung: Falls Erläuterungen nötig sind, die zwar nicht den eigentlichen Zweck der Prozedur beschreiben, aber zum Verständnis der Funktionsweise beitragen, sind diese separat gefaßt.

Beispiel:

```
(* Statt langer Bemerkungen oder Verweise *)
(* ist manchmal ein kurzer                *)
(* Programmabschnitt informativer.        *)
```

Parameter: Falls Parameter Werte von ganz besonderer Bedeutung enthalten können, sind diese hier aufgeführt.

Fehler: Fehler, die mit der Funktionsweise der Prozedur in Zusammenhang stehen, nicht etwa mögliche Syntaxfehler oder dergleichen.

1.3 Ein Wort zur Sprache

Die Informatik als Wissenschaft und vor allem die Informatik als Populärwissenschaft findet natürlich auch ihren Niederschlag in der Sprache. »Die Sprache« ist einerseits die, mit der sich die Informatik selbst zu Beschreiben sucht. Der Informatiker, angefangen beim Systemdesigner bis hin zum subalternen Handbuchschreiber, neigt dabei zu Anglizismen. Andererseits ist »die Sprache« auch die, mit der über die Informatik und ihre Themen gesprochen wird. Hier ist besonders der Aspekt der Populärwissenschaft Informatik spürbar. Eine Fülle von Büchern und Zeitschriften beschäftigt sich mit unterschiedlichem Tiefgang und verschiedenem Vokabular mit dem Thema und pflegt manches Mal einen wunderlichen Soziolekt. Dieses alles ist vor allem der Tatsache zu verdanken, daß die entscheidenden Impulse der Informatik in kommerzieller und damit breitenwirksamer Hinsicht aus den USA kommen. So wie das Lateinische die Sprache der Wissenschaften der religiös geprägten Jahrhunderte war, ist das Englische die Sprache des Zeitalters der Information. Der Autor möchte zwar nicht Rom und Armonk in ihrer Bedeutung für diese oder jene Welt völlig gleichsetzen, aber vielleicht doch ein bißchen?

Zumindest jedoch reicht ihm das als Begründung, daß die Benennung von Leistungen von L3, z.B. in Form der vielen ELAN Prozeduren, an der englischen Sprache orientiert ist. Folgerichtig wurde auch für die Benennung von Parametern die englische Bezeichnung gewählt. Zwar sind damit Formulierungen entstanden, die einen deutschen Leser befremden mögen, dafür jedoch könnte diese Wahl der Übersetzung in andere Sprachen hilfreich sein.

1.4 Woher stammt L3?

L3 ist in einem langjährigen Forschungsprojekt »Betriebssysteme für Mikrocomputer«, das in Kooperation der Gesellschaft für Mathematik und Datenverarbeitung (GMD) und der Universität Bielefeld durchgeführt wird, entstanden. Leiter dieses Projekts und »Chefdesigner« ist Jochen Liedtke. Der »Kern« der L3 Entwicklungsmannschaft besteht aus:

Ulrich Bartling, Uwe Beyer, Dieter Breuer, Achim Heidrich, Dietmar Heinrichs, Dietmar Martinek, Andreas Reichpietsch, Rudolf Ruland, Gyula Szalay und Marcus Zens.

Der diesjährige Preis für äußerste Gerissenheit im Schaffen mit L3 gebührt Axel Kroll und Martin Schönbeck.

Ihnen, allen anderen Mitwirkenden an diesem Projekt und vor allem auch den »Piloten«, die die Fortschritte, aber auch manchen Fehltritt, von den Anfängen an mit begleiteten, sei an dieser Stelle gedankt.

Werner Metterhausen, Siegburg im Januar 1991.

1.5 Der Aufbau von L3

L3 ist ein Multitasking-/ Multiuser- Betriebssystem für Rechner mit mit INTEL 386 Prozessor. L3 läuft grundsätzlich im Protected Mode mit eingeschaltetem Paging. Es handelt sich bei L3 um ein prozeßorientiertes Betriebssystem. Damit unterscheidet es sich in seiner inneren Struktur grundsätzlich von tabellenorientierten (= »dateiorientierten«) Systemen, wie z.B. UNIX. L3 besteht aus den Komponenten **Systemkern**, den **Gerätetreibern**, dem **Systemmantel** und der **Systemschale**. Der Aufbau von L3 läßt sich in folgendem Bild skizzieren:

```
L3 Systemschale  (= Benutzeroberfläche)

L3 Systemmantel (Tasksystem)

L3 Eingabe Ausgabetreiber (Prozesse)
für I/O Devices (= Diskette, Tastatur, Bildschirm...)

      L3 Systemkern

      Geräte        80386
```

Jede Komponente setzt gewisse Fähigkeiten und Vorleistungen der darunterliegenden Komponenten voraus, es handelt sich um eine Hierarchie abstrakter Maschinen, aufbauend auf der konkreten Maschine Intel 80386. Die Beschreibung des Aufbaus und des Zusammenwirkens dieser Komponenten ist Anliegen dieses Abschnitts.

1.6 Der L3 Systemkern

Der L3 Systemkern ist ein Steuerprogramm, das Objekte der Klasse **Prozeß** und der Klasse **Datenraum** erzeugt und verwaltet.

Um diese Leistungen zu erbringen, nutzt der Systemkern insbesondere die Möglichkeit der virtuellen Adressierung (Paging) des Intel 80386 aus.

Der L3 Systemkern bringt die Leistungen:

- Systemverwaltung
- Speicherverwaltung
- Prozeßverwaltung
- Datenraumverwaltung
- Verwaltung des virtuellen 8086 Mode (V86)
- Verwaltung des Arithmetik Coprozessors INTEL 80387.

Die **Systemverwaltung** behandelt den Systemstart, die Behandlung der Betriebssystemaufrufe (Systemcalls) und Interna wie die Systemuhr.

Die **Speicherverwaltung** vewaltet Vordergrund- (RAM) und Hintergrundspeicher (Festplatte). L3 hat eine virtuelle Speicherverwaltung, d.h. es gibt keine Unterscheidung zwischen Schreiben in Puffer (temporäre Dateien im Arbeitsspeicher) und Schreiben auf die Festplatte. L3 kennt stattdessen (logische) Objekte des Typs **Datenraum**. Ein Datenraum ist ein einfaches Objekt des Typs Speicher, das als Folge von Byte betrachtet werden kann. Durch Angabe eines Offset kann in einem Datenraum beliebig adressiert werden. Physisch besteht ein Datenraum aus **Pages** (im folgenden auch als Seiten erwähnt), 4096 Byte großen Abschnitten auf der Festplatte. Die Ablage der Seiten und damit die Verwaltung der Festplatte sind ausschließliche Sache der Speicherverwaltung. Das Laden der durch die aktuell vorliegende Adresse geforderten Page in den Arbeitsspeicher und die damit verbundenen Adreßumsetzungen regelt die L3 Speicherverwaltung unter Benutzung der Memory Management Unit (MMU) des Intel 80386.

Die **Prozeßverwaltung** erschafft und verwaltet Objekte des Typs **Prozeß**. Prozesse sind die einzigen globalen Objekte in einem L3 System, d.h. nur Prozesse, nicht etwa Dateien oder Verzeichnisse, sind im Grundzustand des Systems sichtbar. Ein Prozeß »besteht« aus einem Stück Speicher, seinem Adreßraum, und dem sequentiellen Kontrollfluß in dem Prozeß, **Thread** genannt. Ein Prozeß kann auch aus mehreren Threads bestehen. In diesem Falle teilen sich also mehrere »kleine Prozesse« einen gemeinsamen Adreßraum. Die Programmierschnittstelle zu Threads wird aber erst in einer kommenden L3 Version veröffentlicht.

L3 unterscheidet **Realprozesse** und **virtuelle Prozesse**. Realprozesse sind resident im Arbeitsspeicher. Sie dienen als Gerätetreiber und unterliegen nicht dem Paging. Somit haben sie nur eine beschränkte Lebensdauer. Sie werden beim Abschalten des Systems beendet und beim Start neu etabliert. Virtuelle Prozesse dagegen unterliegen dem Paging und überstehen Systemabschaltungen und -abstürze. Sie sind also dauerhaft. Sie können lokale Datenobjekte verwalten und auf Grund ihrer prinzipiell unbeschränkten Lebensdauer damit natürlich auch speichern. Dateien sind also als lokale Datenobjekte prinzipiell ewig lebender Prozesse realisiert.

Aus Sicht des Systemkerns besteht ein Prozeß aus dem Prozeßkontrollblock (einer Datenstruktur im L3 Systemkern) und (mindestens) einem Datenraum. Dieser sogenannte **Standarddatenraum** (STDDS) des Prozesses enthält Code und Daten. Einfach dargestellt ist der Standarddatenraum eine Datei, die ein fertig übersetztes, lauffertiges Programm enthält, das bei Erzeugung des Prozesses gestartet wird.

Der **Prozeßkontrollblock** (PCB) ist der Anker eines Prozesses im Systemkern. Er enthält neben nur für den Systemkern relevanten Informationen insbesondere das Abbild aller Registerinhalte und somit des Befehlszählers.

Im Prozeßkontrollblock ist auch der **Botschaftsvektor** des Prozesses enthalten. Über diese Datenstruktur tauschen Prozesse Botschaften aus. Kontrolle und Ausführung des Botschaftstransfers unterliegen dem Systemkern.

Botschaften sind ein weiteres Grundkonstrukt von L3. Eine Botschaft ist ein zusammengesetztes Objekt, das aus mehreren Abschnitten besteht. Abschnitte können leer oder von Typ Integer, Bytestring oder Datenraum sein. Unabhängig vom Umfang der Botschaft, die durchaus aus mehreren Datenräume bestehen kann, also den (logischen) Umfang von zig Megabyte haben kann, ist der Transfer der Botschaft atomar.

Die Botschaftsübermittlung in L3 ist synchron, d.h. der Empfänger einer Botschaft muß empfangsbereit sein, damit der Sender seine Sendung erfolgreich absetzen kann. Alle möglichen Abläufe innerhalb von L3 basieren auf dem Botschaftstransfer zwischen Prozessen. Da auch die Ein-/Ausgabetreiber Prozesse sind, ist z.B. jede Ausgabe eines beliebigen Anwendungsprogramms auf dem Bildschirm eine Botschaft an den zugeordneten Bildschirmtreiber.

Die **Datenraumverwaltung** liefert und verwaltet Objekte des Typs **Datenraum**. Der Auftrag, einen Datenraum zu erzeugen oder der Anstoß zu Aktionen in Bezug auf einen existierenden Datenraum kommt stets von irgendeinem Prozeß. Jeder Datenraum ist genau einem Prozeß zugeordnet, hat also einen eindeutigen Besitzer. Datenräume sind also tasklokale Objekte. Die Datenraumverwaltung stellt auch den Mechanismus des faulen Kopierens zur Verfügung. Die Grundidee dabei ist, daß es hinreichend ist, beim Kopieren eines Datenraums zunächst nur einen weiteren Verweis auf den Datenraum zu erzeugen. Danach zeigen dann zwei verschiedene Verweise auf ein Datenobjekt. Erst bei Schreibzugriffen ist es nötig, Seiten, die verändert werden zu kopieren. Auch an dieser Stelle werden also nur Teile des logisch mehrfach vorhandenen Datenobjekts physisch kopiert.

Das **V86 Management** liefert die Leistungen, die Prozesse benötigen, die in den virtuellen 8086 Mode (V86) des Prozessors wechseln. Neben den Umschaltmechanismen zwischen V86 und 386 Mode finden sich hier insbesondere Emulationen für die kritischen 8086 Befehle `CLI`, `STI`, `PUSHF`, `POPF`, `INT` ,`IRET`.

Das **387 Management** verwaltet den Coprozessor, bzw. die Coprozessoremulation. Der Systemkern enthält ein Modul zur Emulation des Intel 80387. Falls kein numerischer Coprozessor des Typs 80387 oder 80287 in den Rechner eingebaut ist, wird die Emulationssoftware zur Ausführung der entsprechenden Operationen herangezogen.

Die Schnittstelle aller anderen Programme zum Systemkern sind L3 **Systemcalls**. Systemcalls sind also Botschaften von Prozessen an den Kern und initiieren Aktionen, die nicht von Prozessen geleistet werden können.

1.7 Treiberprozesse

Eng verbunden mit dem Systemkern existieren die Treiberprozesse für Eingabe und Ausgabe. Sie kontrollieren die Devices (Komponenten) des Rechners. Die Treiberprozesse sind Realprozesse, d.h. sie sind resident, also stets im Arbeitsspeicher und unterliegen nicht dem Paging. Treiberprozesse haben direkten Zugriff auf I/O Ports, sind jedoch adreßraummäßig abgeschottet, so daß sie andere Prozesse oder gar den Systemkern nicht korrumpieren können.

Treiberprozesse werden bei jedem Systemstart durch den Systemkern neu etabliert.

Ein Treiberprozeß besteht also aus einem Treiberprogramm der Art:

```
REPEAT
   warte auf auftrag;
   bearbeite auftrag
END REPEAT
```

Ein Auftrag kommt geräteseitig in Form eines Interrupts, systemseitig in Form eines Auftrags, der nach den Regeln des L3 Treiberprotokolls **GDP (General Driver Protocol)** formuliert ist. Die Bearbeitung des Auftrags ist gerätespezifisch.

1.8 Der L3 Systemmantel

Der L3 Systemmantel enthält den Teil des Gesamtsystems, der auf Leistungen des Kerns aufbaut, aber noch nicht modifizierbar ist. Der Systemmantel bildet die Schnittstelle zwischen der abstrakten Maschine Systemkern und den von ihr zur Verfügung gestellten Objekten vom Typ Prozeß und Datenraum und der auf den Benutzer ausgerichteten Oberfläche des Systems.

Diese Oberfläche, die Systemschale, wird erzeugt von einzig sichtbaren Teil des Systemmantels, der Task SUPERVISOR.

Eine Task im L3 System ist ein autonomer, benannter Prozeß. Der abstrakte Datentyp Prozeß wird auf der Ebene des Systemmantels zur Task ausgeformt. Dabei entsteht die erste Task im System, der SUPERVISOR, als »deus ex machina«, alle weiteren Tasks werden von dieser ersten Task erzeugt. Der SUPERVISOR ist Wurzel und Verwalter des baumartig organisierten Tasksystems.

Im laufenden System können andere Tasks den Auftrag an SUPERVISOR geben, eine neue Task anzulegen oder eine existierende zu löschen. Weitere Einzelheiten dazu finden sich in den Abschnitten 5 und 13 (das SUPERVISOR Protokoll).

1.9 Die L3 Systemschale

Der SUPERVISOR erschafft unter Benutzung der Leistungen des Systemkerns wird mit Hilfe von

- Laufzeitsystem (RTS) und
- ELAN Compiler

das L3 Tasksystem.

Das **ELAN Laufzeitsystem** ist vor allem als Systemcall Schnittstelle für Erzeugung und Betrieb des gesamten L3 Systems aufzufassen.

Der **ELAN Compiler** ist von Beginn seiner Entwicklung an für ein Betriebssystem mit virtueller Speicherverwaltung vorgesehen und unterscheidet sich von üblichen Compilern. Da die virtuelle Speicherverwaltung nicht von vornherein Überlegungen über Limitierungen von Segmenten, Dateigrößen oder Arbeitsspeicher erfordert, konnten vom üblichen Schema:

```
Quellcode     ->   Compiler     ->   Objektcode
  Datei                                Datei
```

abgewichen werden.

Stattdessen ist der Compiler in eine Umgebung eingebettet, die, neben dem Compiler an sich, noch das ELAN L3 Laufzeitsystem und die ELAN Modulbibliothek enthält, die Code und Daten übersetzter ELAN Programme enthält .

```
0 MB
 |
 |  Laufzeitsystem
```

```
    |
    |  Compiler
    |
    |  Modulbibliothek
    |
  32 MB
```

Dieses ganze komplexe Objekt wird bei L3 als Standarddatenraum für jeden virtuellen Prozeß benutzt. Startprozedur des Prozesses ist dann eine aus einem der Moduln herausgereichte ELAN Prozedur.

Die L3 Systemschale wird durch die Eigenschaften und Fähigkeiten der einzelnen Tasks des L3 Systems geprägt. Meistenteils sind die Tasks so angelegt, daß sie einen Kommandodialog anbieten, sobald an sie ein Terminal gekoppelt sind. Dieser Dialog kann eingeschränkt sein, wie bei einer Terminaltask, oder frei und kommandozeilenorientiert, wie in der Task 'PUBLIC'.

Zur Systemschale gehört auch das L3 Subsystem, das eine Umgebung für »DOS Programme« realisiert. In dieser Umgebung laufen DOS Programme virtuellen 8086 Modus.

Zu diesem Zweck wird nach dem Auftrag an das V86 Management des Systemkerns, die Task in den virtuellen 8086 Modus zu setzen, der Standarddatenraum der Task modifiziert. Der zunächst ungenutzte Adreßbereich des ersten Megabytes wird durch Einbettung des **VBIOS** (Virtual BIOS), Zuweisung des Adreßraums der Task als Laufwerk C: und Hereinholen des Dateien 'COMMAND.COM' und 'L3.EXE' in eine funktionstüchtige Umgebung für DOS Programme verwandelt.

```
0x00000h     Interruptvektoren
...
0x10FFFh     ELAN Laufzeitsystem
....
```

Die in jeder Task vorhandene »Lücke im Adreßraum«, die erst durch Initialisierung des L3 Subsystems geschlossen wird, kostet durch die virtuelle Speicherverwaltung keinen Platz im Arbeitsspeicher!

2 Der ELAN Compiler

2.1 Einführung

Der in das Betriebssystem L3 integrierte ELAN Compiler übersetzt ELAN Programme in ausführbaren 80386 Code. Er ist nicht nur ein Werkzeug zum Übersetzen von ELAN Quelltexten, sondern auch der Verwalter der ausführbaren ELAN Programme einer Task.

Stellung und Funktionsweise des ELAN Compilers sind Resultat des L3 Taskkonzepts. Eine Task wird angelegt, indem der L3 Systemkern den Auftrag erhält, einen Prozeß zu erzeugen. Dabei wird insbesondere ein Datenraum benötigt, der der **Standarddatenraum** (STDDS) des neuen Prozesses sein soll. Dieser Datenraum muß, zumindest in Teilen, ausführbaren Code enthalten. Durch Angabe der Adresse in diesem Standarddatenraum, auf die der Befehlszähler zu Beginn zeigen soll, kann eine »Startprozedur« festgelegt werden, die unmittelbar nach Erzeugung des Prozesses gestartet wird.

Im »Normalbetrieb« wird der Standarddatenraum einer neuen Task durch *faules Kopieren* des STDDS der Vatertask erzeugt. Die Startprozedur wird kann explizit vorgegeben werden, das System bietet auch eine Voreinstellung an. Wenn im Terminalmenü der Punkt `neue Task erzeugen` angewählt wird und als Vatertask `PUBLIC` bestätigt wird, so wird der STDDS von PUBLIC faul kopiert und eine Prozedur gestartet, die einen Kommandodialog anbietet. Offensichtlicher wird der Mechanismus durch die ELAN Syntax, wenn eine Task mit einer besonderen Startprozedur in einem Programm erzeugt wird: `begin ("teilnehmer", PROC startproc, tvar)`. Hier wird eine Sohntask mit der (nachträglich erzeugten) Prozedur »startproc« angelegt. Näheres dazu siehe Kap. 5.

In den im System vorhandenen und weiteren durch das ELAN System erzeugten Tasks ist der Standarddatenraum wie folgt aufgebaut:

Standarddatenraum

`0x00000000`	reserviert für
...	L3DOS
`0x0010ffff`	
`0x00110000`	
...	ELAN Laufzeitsystem
`0x0011ffff`	
`0x00120000`	
...	ELAN Compiler Code
`0x00180000`	
...	Stack
`0x001fffff`	
`0x00200000`	
...	ELAN Programmcode
`0x006fffff`	
`0x00700000`	
...	ffff
`0x007fffff`	
`0x00800000`	
...	Compilertabellen
`0x009affff`	
`0x009b0000`	
...	Graphikdaten
`0x009fffff`	
`0x00A00000`	
...	ELAN Paketdaten
`0x011fffff`	
`0x01200000`	
....	Textheap
`0x01ffffff`	

Der STDDS einer Task enthält also nicht etwa nur den ausführbaren Code von ELAN Programmen, sondern auch den Compiler, um weitere ELAN Programme zu übersetzen, und die Modulbibliothek (Compilerdatenbank), die der Compiler zur Verwaltung bereits übersetzter Programme braucht. Da die Moduln im STDDS, der für diese Zwecke hinreichend groß ist, immer zur Verfügung stehen, ist ein besonderer Dateityp »ausführbare Datei« und ein »Linker«, über den Referenzen eines in Übersetzung befindlichen Programms auf andere Moduln gesetzt werden, nicht erforderlich. Die im STDDS vom Compiler verwalteten Moduln werden direkt über ihren Namen adressiert. Die Bereiche »ELAN Programmcode« und »ELAN Paketdaten« des STDDS enthalten zunächst Code und Daten der bei der Systemerzeugung übersetzten ELAN Pakete. Die Programmdaten des aktuell laufenden Programms sind auf dem Stack gespeichert.

Die Startprozedur bei Anlage einer neuen Task kann nun dergestalt sein, daß sie eine »Commandshell« folgender Systematik anbietet:

```
REPEAT
   prompt ausgeben;
   zeichenkette empfangen;
   zeichenkette an compiler;
   IF fehlerfrei THEN ausführen
                 ELSE fehlermeldung ausgeben
   FI
END REPEAT
```

Der Compiler prüft, ob die Zeichenkette ein syntaktisch korrektes ELAN Programm darstellt und führt es im Falle der Korrektheit aus. Da gerade Namen von Objekten, die in der Schnittstelle eines ELAN Paketes stehen, als korrekte ELAN Programme erkannt und via Modulbibliothek direkt adressiert und ausgeführt werden können, steht der ELAN Compiler als »Command Interpreter« zur Verfügung. Alle sichtbaren Objekte (Prozeduren, Typen etc.) in den Schnittstellen der vorübersetzten ELAN Pakete im STDDS sind damit in Kommandos verwendbar. Die Liste der sichtbaren Objekte kann durch das Kommando **bulletin** angezeigt werden.

Desweiteren kann der Compiler natürlich auch als »normaler« Compiler benutzt werden, um ELAN Quellprogramme, die in einer Textdatei stehen, zu übersetzen. Ein solches Programm wird entweder übersetzt und, falls korrekt, direkt ausgeführt (**run**) oder insertiert (**insert**).

Insertieren bedeutet, daß das Programm, das mindestens ein PACKET enthalten muß, übersetzt wird und der Programmcode, wiederum die Korrektheit vorausgesetzt, in den Codebereich der Task eingetragen wird. Der Packetcode, also alle Anweisungen vor, zwischen und hinter PROC/OP Deklarationen wird unmittelbar nach der Übersetzung eines PACKETs ausgeführt. Die Objekte in der Schnittstelle des PACKET sind dann über die Modulbibliothek sichtbar. Auf diese Weise kann der »Kommandovorrat« einer Task erweitert werden.

Die Sichtbarkeitsregeln für die insertierten Objekte ergeben sich nach kurzer Betrachtung der L3 Kopierlogik: bei Erschaffung einer Task wird eine Kopie des STDDS der erzeugenden Task als STDDS für die neue Task genommen. Die neue Task »erbt« also den Kommandovorrat. Wenn in dieser neuen Task nun insertiert wird, erfolgt ein Schreibzugriff auf den STDDS. Dabei wird neuer Programmcode in den STDDS geschrieben, neue Seiten werden entkoppelt. Damit sind die neu insertierten Objekte nur in der Task sichtbar, in der sie insertiert wurden, und in deren Nachkommen, die diesen erweiterten STDDS erben.

2.2 Compiler Operationen:

Übersetzen	*Protokollieren*	*Information*
do	check	bulletin
insert	check on	help
run	check off	packet
	prot	
	prot off	
	warnings	
	warnings on	
	warnings off	

2.2.1 Übersetzung

do

PROC do (TEXT CONST elanprog)

Wirkung: Der Text 'elanprog' wird an den ELAN Compiler übergeben und übersetzt. Falls er ein fehlerfreies ELAN Programm darstellt, wird dieses sofort ausgeführt.

Bemerkung: Die Ausführung von **do** in einem Programm entspricht also der Wirkung der interaktiven Kommandoeingabe.

Fehler: Falls ein Syntaxfehler auftritt, wird ein **errorstop** ausgelöst.

run

PROC run (TEXT CONST filename)

Wirkung: Der Inhalt der Datei wird übersetzt. Falls die Übersetzung fehlerfrei verläuft, wird der erzeugte Code sofort ausgeführt.

Bemerkung: Fehlermeldungen werden grundsätzlich im Notizbuch vermerkt. Wird ein Fehler gefunden, wird auf der oberen Bildschirmhälfte ein Editorfenster auf den Quelltext in 'filename' und in der unteren Hälfte das Notizbuch mit den Fehlermeldungen geöffnet.

PROC run

Wirkung: Übersetzt die zuletzt benutzte Datei.

Siehe: PROC last param , TEXT PROC std

insert

PROC insert (TEXT CONST filename)

Wirkung: Der Inhalt der Datei 'filename' wird übersetzt. Die Datei muß also den Quelltext eines oder mehrerer PACKETs enthalten. Falls die Übersetzung fehlerfrei verläuft, sind alle Objekte, die in den Schnittstellen aufgeführt sind, im Standarddatenraum der Task verfügbar.

Bemerkung: Fehlermeldungen werden grundsätzlich im »Notizbuch« **notebook** vermerkt. Wird ein Fehler gefunden, wird auf der oberen Bildschirmhälfte ein Editorfenster auf den Quelltext in 'filename' und in der unteren Hälfte das Notizbuch mit den Fehlermeldungen geöffnet.

Fehler: Falls die Datei den Quelltext mehrerer PACKETs enthielt und in einem ein Syntaxfehler auftrat, so ist keines der PACKETs insertiert.

PROC insert

Wirkung: Insertiert die zuletzt benutzte Datei.

Siehe: PROC last param , TEXT PROC std

ACHTUNG: Insertieren bedeutet grundsätzlich Code und Daten anzufügen! Auch ständiges Neuinsertieren eines einzigen PACKETs, wie möglicherweise bei der Entwicklung nötig, hat diesen Effekt. Es ist möglich den Standarddatenraum einer Task durch fortlaufendes Insertieren eines PACKETs vollaufen zu lassen und so eine der *Überlauf-Fehlermeldungen* (siehe Kap. 2.5) zu erzeugen. Auch aus diesem Grunde ist es ratsam, die Programmentwicklung stets in einer Sohntask der Task, die die Programmquellen enthält, durchzuführen.

Bemerkung:

1. Ein PACKET wird also ausgeführt, sobald es ohne Syntaxfehler übersetzt wurde. Sind in einer Datei mehrere PACKETs, die nacheinander übersetzt werden sollen, so wird nach dem erfolgreichen Übersetzen und Ausführen eines PACKETs das darauf folgende übersetzt. Angenommen, eines der PACKETs enthält Fehler und der Code von vorher übersetzten PACKETs wurde bereits ausgeführt. Wird nun der Fehler in dem PACKET, dessen Übersetzung abgebrochen wurde, korrigiert und die Datei zum Insertieren sämtlicher PACKETs erneut aufgerufen, kann es zu weiteren Fehlermeldungen kommen. Wenn z.B. ein PACKET, das im ersten Versuch korrekt übersetzt und ausgeführt werden konnte, eine Task neu erzeugt, würde ein nochmaliges Insertieren mit der Fehlermeldung "Task existiert bereits" an dieser Stelle abbrechen. Derartige Effekte sollten in den entsprechenden Dateien, in denen PACKETs zur Übersetzung und Ausführung zusammengefaßt werden, z.B. Generatoren, berücksichtigt und abgefangen werden.
2. Rekursive Aufrufe des Compilers sind verboten. Als Anweisung im letzten PACKET führen Aufrufe des Compilers *nicht* zu Rekursion, da der Compiler das PACKET erst startet, wenn seine eigenen Aktionen abgeschlossen sind.

2.2.2 Protokollieren

Der ELAN Compiler kann mit ein- oder ausgeschaltetem **Checkmodus**, **Protokollmodus** und **Warnmodus** benutzt werden. Die Einstellungen sind tasklokal und werden beim Anlegen einer neuen Task vererbt.

Die tiefgreifendsten Auswirkungen hat der Checkmodus. Ist dieser Modus eingeschaltet, werden bei der Übersetzung Zeilennummern im Code des übersetzten Programms mitgeführt. Bei Auftreten eines Laufzeitfehlers kann dann die Zeilennummer in der Quelldatei ausgegeben werden, sowie der Name des PACKETs, in der/dem der Fehler bemerkt wurde.

Durch Einfügen von Zeilennummern wird der erzeugte Code um ca. 25% größer. Es ist deshalb eine übliche Vorgehensweise, bei Entwicklung und Test mit **check on** zu arbeiten und das PACKET mit den fertig ausgetesteten Dateien mit **check off** zu insertieren.

Die Zeilennummern des ELAN Programms, die von **check** und **prot** gesetzt werden, sind dieselben, die auch beim Ausdruck einer ELAN Programmdatei im Listing erscheinen.

check

BOOL PROC check

Wirkung: Liefert 'TRUE', falls der Checkmodus in der Task eingeschaltet ist.

check off

PROC check off

Wirkung: Checkmodus abschalten.

check on

PROC check on

Wirkung: Checkmodus anschalten.

prot

PROC prot (TEXT CONST filename)

Wirkung: Protokollmodus einschalten. Jede folgende Übersetzung wird auf dem Bildschirm protokolliert. Das Programmlisting und alle Meldungen des Compilers werden in die Datei 'filename' geschrieben.

Fehler: File Überlauf. Die Protokolldatei ist vollgeschrieben.

prot

BOOL PROC prot

Wirkung: Informationsprozedur, die TRUE liefert, falls der Protokollmodus eingeschaltet ist.

prot off

PROC prot off

Wirkung: Protokollmodus ausschalten.

Bemerkung: Normalmodus des Compilers.

warnings

BOOL PROC warnings

Wirkung: Informationsprozedur, die TRUE liefert, falls der Warnungsmodus eingeschaltet ist.

warnings on

PROC warnings on

Wirkung: Warnungsmodus einschalten. Bei nachfolgenden Übersetzungen werden auch Warnungen und nicht nur Fehlermeldungen ins Notizbuch eingetragen.

warnings off

PROC warnings off

Wirkung: Warnungsmodus ausschalten. Warnungen des Compilers werden am Bildschirm gezeigt, aber nicht in das Notizbuch protokolliert.

2.2.3 Informationen

bulletin

PROC bulletin (TEXT CONST packet name)

Wirkung: Listet die Namen aller Prozeduren, Operatoren und Typen, die in der Schnittstelle des Pakets 'packet name' stehen, in eine temporäre Datei und zeigt diese im »show« Modus.

Fehler: "packet name" ist kein Paketname.

PROC bulletin

Wirkung: Listet die Namen aller Objekte aus allen insertierten Paketen, die in dieser Task sichtbar sind, in eine temporäre Datei und zeigt diese Datei im »show« Modus.

help

PROC help (TEXT CONST pattern)

Wirkung: Listet die Namen aller Prozeduren / Operatoren mit dem Namen 'pattern'. Zum Namen wird das Paket, in dessen Schnittstelle die Prozedur steht, mit ausgegeben.

Bemerkung: Als 'pattern' kann auch ein Name angegeben werden, der nur annähernd bekannt ist. Dazu kann an den Anfang und/oder das Ende von 'pattern' das Zeichen/der Operator * (wildcard) gestellt werden. So lassen sich alle Prozeduren finden, die mit 'pattern' anfangen, darauf enden oder in denen 'pattern' vorkommt.

Beispiel: `help ("*info*")`

packets

PROC packets

Wirkung: Listet die Namen aller in der Task insertierten Pakete in eine temporäre Datei und zeigt sie im »show« Modus.

2.3 Speicherbedarf

Der Compiler übersetzt ELAN Programme in 80386 Code. Die elementaren (denotierbaren) Datentypen der Sprache ELAN sind BOOL, INT, REAL und TEXT. Weiterhin kennt ELAN zusammengesetzte Datentypen (ROW und STRUCT), die wiederum aus einfachen oder zusammengesetzten Typen bestehen.

Die elementaren Datentypen des INTEL 80386 sind Bytes, Worte und Doppelworte. Dementsprechend sind auch die Typen BYTE, WORD, DWORD als (quasi-elementare) Typen verfügbar.

Typ	Länge in Bytes
BYTE	1
WORD	2
INT	4
BOOL	4
DWORD	4
REAL	8

Die interne Darstellung der REALs ist die binäre 8 Byte Fließkommadarstellung nach IEEE 754.

TEXT

Jedes Datenobjekt vom Typ TEXT besteht aus einem festen Teil von 16 Bytes und ggf. einem flexiblen Teil auf dem *Textheap*. Texte mit einer Länge bis zu 11 Byte werden im festen Teil gespeichert. Der feste Teil enthält im ersten Byte die Textlänge, gefolgt vom Text selbst. Die letzten vier Byte sind in diesem Fall ohne Bedeutung. Längere Texte werden auf dem Textheap des Datenraums gespeichert. Im ersten Byte des Textes steht dann `0xff`, das zweite ist ohne Bedeutung. Die Bytes drei und vier enthalten die Länge des Textes auf dem Textheap, die anschließenden 8 Byte sind ohne Bedeutung. In den letzten vier Byte steht der **Heaplink**, also die Adresse des Textes im Textheap.

Aufbau des Textheap im Standarddatenraum

Im Standarddatenraum liegt der Textheap in den oberen 14 MB des (32 MB großen) Standarddatenraums (Adreßbereich `0x1200000` bis `0x1ffffff`). Der Textheap wird vom ELAN Laufzeitsystem verwaltet. Der Textheap ist in Container verschiedener Größe eingeteilt. Es wird stets versucht, einen Text in einem jeweils minimalen passenden Container zu speichern:

Textklasse	max. Nettogröße in Byte	Bemerkung
1	16 - 5	s.o.
2	32 - 4	ersten 4 Byte 'Backlink'
3	64 - 4	
4	128 - 4	
5	1 * 4 KB	Separater Backlink
6	2 * 4 KB	die Pages werden
..	..	
11	7 * 4 KB	vollständig für
12	8 * 4 KB	Text genutzt

Backlink ist der Zeiger auf den festen Textteil zu dem der Container gehört.

Aufbau des Textheap in anderen Datenräumen

In Datenräumen, die benutzereigene Struktur aufweisen, wird der Heap hinter dem statischen Teil (siehe unten) aufgebaut. Der Datenraum enthält zwei Zeiger auf Heapstart und aktuellen Heaptop. Platz auf dem Heap wird in Vielfachen von 16 Byte vergeben.

```
Benutzerdef. Datenraum:

        Zeiger auf ersten freien Container
        Zeiger auf ersten Container
         ...
        Statischer Teil des Datenraums
        ---------------
        Erster Textcontainer
        x.Textcontainer
         ...
        y.Textcontainer
        erster freier Container
```

Jeder Container hat in seinen ersten beiden Byte eine Kapazitätsangabe `c (1 <= c <= 2048)`. Die Kapazität des Containers errechnet sich durch `(16 * c) - 2`. Falls ein Text über die Größe seines Containers hinaus wächst, wird ein neuer Container an der Heapspitze zugewiesen. Eine »Müllabfuhr« auf dem Textheap findet nicht statt. Es kann also nützlich sein, Daten in einen neuen Datenraum umzukopieren, wenn sehr viel Speicherplatz durch »alte« Texte verbraucht wird.

Die Mindestgröße des Textheap in einem Datenraum kann mit der Prozedur **set min heapsize** festgesetzt werden.

ROW und STRUCT Objekte

ROW und STRUCT Objekte werden gemäß ihrer Länge ausgerichtet. Jedes ROW oder STRUCT Datenobjekt, dessen Länge mehr als 5 Byte beträgt, wird auf durch 8 teilbare Adressen ausgerichtet.

Beispiel:

Typ	Länge (Byte)
ROW 2 BYTE	2, ausgerichtet auf 4
STRUCT (INT, BYTE)	8
ROW 5 INT	24
ROW 100 STRUCT (INT, INT)	800
ROW 100 STRUCT (REAL, REAL)	1600
ROW 100 STRUCT (BYTE, INT)	800
ROW 100 STRUCT (TEXT, INT, INT)	2400
ROW 100 STRUCT (INT, TEXT, INT)	3200

2.3.1 PACKED Option

Um vorgegebene komplexe Datenstrukturen in ELAN exakt nachbilden zu können, kann man die Ausrichtungsstrategie des ELAN-Compilers beeinflussen. Wird einer Struktur- oder Row-Deklaration das Schlüsselwort PACKED vorangestellt, so werden

- die Elemente innerhalb dieser Struktur und die Länge der Struktur nicht mehr ausgerichtet
- die Elemente des Feldes und die Länge des Feldes nicht mehr ausgerichtet

Beispiel:

```
LET MCB = STRUCT (BYTE marker ,
                  WORD owner, length,
                  ROW 11 BYTE reserved) ;
```

Der Compiler richtet WORDs auf eine geradzahlige Adresse aus, Objekte mit einer Länge >= 8 auf eine durch 4 teilbare Länge. Ein MCB wird also wie folgt abgelegt:

- 1 BYTE marker
- 1 Byte Ausrichtung auf eine durch 2 teilbare Adresse
- 1 WORD owner
- 1 WORD length
- 2 Byte Ausrichtung auf eine durch 8 (!) teilbare Adresse
- 16 (!) Byte ROW reserved

Gesamtlänge 24 Bytes.

```
LET MCB = PACKED STRUCT (BYTE marker ,
                         WORD owner, length,
                         ROW 11 BYTE reserved) ;
```

Der Compiler richtet innerhalb der Struktur nicht mehr aus:

- 1 BYTE marker
- 1 WORD owner
- 1 WORD length
- 16 (!) Byte ROW reserved

Gesamtlänge 21 Byte.

```
LET MCB = PACKED STRUCT (BYTE marker ,
                         WORD owner, length,
                         PACKED ROW 11 BYTE reserved) ;
```

Jede Ausrichtung ist unterdrückt:

- 1 BYTE marker

- 1 WORD owner
- 1 WORD length
- 11 Byte ROW reserved

Gesamtlänge 16 Byte.

Vor allen anderen Typbezeichnern darf PACKED stehen, hat aber keine Wirkung:

```
BYTE VAR byte ; PACKED INT VAR i;
BYTE VAR byte ;        INT VAR i;
```

Beide Zeilen sind äquivalent.

Anmerkung: Der Compiler richtet - wenn nötig - Paketdaten so aus, daß Objekte mit einer Länge >= 4 (8) immer auf einer durch 4 (8) teilbaren Adresse beginnen; in dem obigen Beispiel werden also zwischen dem BYTE und dem INT 3 Byte freigelassen. Der 80386 kann zwar auf nicht ausgerichtete (D)WORDs zugreifen, benötigt dafür aber zusätzliche Takte. Bei einem Bereich von 8 MB für Paketdaten sind solche Ausrichtungs»verluste« wohl unerheblich...

2.3.2 Forcer

Bei der systemnahen Programmierung entsteht oft die Notwendigkeit, Typen explizit umzubetrachten. Hierzu dient der **Forcer**

```
TYPE forcer ::  TYPE declarer, square bracket open token,
                               TYPE1 expression,
                               square bracket close token .
```

Die Notation ist der des Konstruktors ähnlich, im Gegensatz dazu liefert der Forcer jedoch ein *Objekt* und keinen Wert. Wie im folgenden Beispiel gezeigt wird, findet keine Überprüfung bezüglich der Längen der Typen statt!

Beispiel:

```
ROW 100 INT VAR queue ;

shift queue down :
    que := tail .

que  : ROW 99 INT [queue head] .

tail : ROW 99 INT [new queue head] .

queue head     : queue [1] .

new queue head : queue [2] .
```

Für das Refinement `shift queue down` wird ein `REP MOVSD` generiert.

```
OP := (BYTE VAR b, INT CONST r) :

    b := BYTE [r]

ENDOP := ;
```

Es wird nur das niederwertige Byte des INTs kopiert.

Achtung:

```
OP := (INT VAR l, BYTE CONST b) :

    l := INT [b]

ENDOP := ;
```

Der in diesem Zuweisungsoperator benutzte Forcer »verlängert« das Byte zu einem DWORD, die höherwertigen 3 Byte sind undefiniert!

```
REF INT VAR bios var := REF INT [0x 404]
```

Hier wird der Forcer benötigt, um eine Referenz (s.u.) auf einen absoluten Wert zu setzen.

2.3.3 INLINE

Um die Programmierung effizienter Software zu unterstützen, bietet der L3 ELAN Compiler die **INLINE** Anweisung für Prozeduren (und Operatoren). Der Code einer als INLINE deklarierten Prozedur wird direkt an der Aufrufstelle eingesetzt. Die INLINE Eigenschaft gilt selbstverständlich nicht nur im definierenden PACKET, sondern auch, wenn die Prozedur aus dem vorinsertierten Standard kommt.

Beispiel:

```
TYPE COMPLEX = STRUCT (REAL real, imag):

OP := (COMPLEX VAR dest, COMPLEX CONST source):
   INLINE ;
   CONCR (dest) := CONCR (source)
END OP := ;

REAL PROC real (COMPLEX CONST x):
   INLINE ;
   x.real
END PROC real ;
```

```
REAL PROC imag (COMPLEX CONST x):
   INLINE ;
   x.imag
END PROC imag ;
```

Außerhalb des PACKETs würde der Zugriff auf die Feinstruktur eines Objektes dieses Typs zu einer »teuren« Aktion, da nur über spezielle Prozeduren zugegriffen werden kann. Durch die INLINE Anweisung wird der Code der Zugriffsprozedur an der Aufrufstelle generiert. Der Zugriff ist also in der aufrufenden Prozedur genauso effizient wie innerhalb des definierenden PACKETs.

Zwischen dem Prozedurkopf und dem Schlüsselwort INLINE dürfen nur Blanks und Kommentare stehen.

2.4 Referenzen in ELAN

Die DOS-Emulation und andere Anwendungen haben gezeigt, daß es notwendig ist, in ELAN die Behandlung von Adressen zu ermöglichen. Dazu wird die Typ-Deklaration in ELAN wie folgt geändert:

```
type declarer ::  plain type ;
                  row type ;
                  struct type ;
                  ref symbol token, type declarer.

ref symbol :: REF .
```

Der extended range wird um ein, vom Compiler realisiertes, Ref-Paket erweitert. Es stellt folgende »super«generischen Operationen zur Verfügung:

OP := (REF TYPE VAR l, REF TYPE CONST r)

Wirkung: Kopie der Referenz.

Ausserdem gibt es eine - dem CONCR ähnliche - Compileranweisung zum Dereferenzieren:

VAL (REF TYPE ACCESS obj)

Wirkung: liefert ein TYPE ACCESS obj .

Beispiel:

```
a)  INT VAR x ;
    REF INT VAR i :: address (x) ;
    VAL (i) := 0       (* setzt 'x' auf 0 *)

b)  INT CONST x ;
    REF INT VAR i :: address (x) ;
    VAL (i) := 0       (* geht nicht! 'x' ist eine Konstante *)

c)  REF INT VAR a, b;  (* macht der Compiler *)
    ...
    a := b

d)  LET X = REF UBOOT ;
    X VAR a, b ;
    ...
    a := b ;
```

Als unsicheres Feature wird eine nicht kopierende ref-Proc eingeführt. Sie ist unabdingbar für Systemzwecke.

REF TYPE PROC address (TYPE VAR obj)

Wirkung: Liefert die Adresse von 'obj'.

Bemerkung: 'obj' muß das Accessrecht VAR haben.

Bemerkung: Diese »super«generischen Prozeduren/Operatoren werden bei der 'help'-Funktion nicht angezeigt.

2.5 Fehlermeldungen des ELAN Compilers

Neben den Fehlermeldungen, die sich auf die übersetzten ELAN Programm beziehen, gibt es weitere Fehlermeldungen des ELAN Compilers. Diese beziehen sich auf den Compiler selbst oder auf den vom ELAN Compiler verwalteten Standarddatenraum. Diese Fehlermeldungen erscheinen stets in der Form:

COMPILER ERROR: <zahl>

wobei <zahl> folgende Werte annehmen kann:

Bedeutungen und eventuelle Abhilfe

`101` Überlauf der Namenstabelle

Die Anzahl der Namen aller sichtbaren Pakete ist zu groß. Der Fehler kann auftreten, wenn die Anführungstriche eines TEXT- Denoters fehlen. Andernfalls gibt es keine Abhilfe. Es muß versucht werden, das Programmsystem auf mehrerere Tasks zu verteilen. Häufige Ursache ist ständiges Neuinsertieren eines PACKETs.

`102` Überlauf der Symboltabelle

Die Anzahl der deklarierten Objekte im gerade übersetzten PACKET ist zu groß. Das PACKET muß in kleinere PACKETs unterteilt werden.

`103` Überlauf des Zwischencodebereiches

Das PACKET muß in kleinere PACKETs unterteilt werden.

`104` Überlauf der Permanenttabelle

Zu viele Pakete insertiert. Kann insbesondere dann auftreten, wenn in einer Task ein Paket zigfach zwecks Test insertiert wurde. Neue Task beginnen.

`105` Überlauf der INLINE Tabelle.

Die Tabelle enthält Informationen über die PROCs/OPs, die in der PACKET Schnittstelle stehen und als INLINE deklariert sind. Die INLINE Prozeduren müssen über mehrere PACKETs verteilt werden.

`107` Lokale Datenadresse zu groß

Die benutzerdefinierten lokalen Daten einer Prozedur belegen mehr als 128K. Keine Abhilfe.

`301` Modulnummern-Überlauf

Zu viele sichtbare Pakete, Prozeduren und Operatoren (> 20480). Keine Abhilfe.

`303` **und** `304` Zu viele Ansprungadressen

In dem gerade übersetzten Modul (Prozedur, Operator oder Paketrumpf) werden vom Compiler zu viele Marken benötigt. Marken werden z.B. für die Codegenerierung von Auswahl (`IF ...`) und Wiederholung (`REP ...`) gebraucht. Insbesondere bei SELECT-Anweisungen werden `casemax - casemin + 2` Marken benötigt, wobei »casemax« der INT-Wert des maximalen, »casemin« der des minimalen CASE-Wertes ist. Dieser Fehler ist somit fast immer auf zu viele und/oder zu weit gespannte SELECT- Anweisungen zurückzuführen. SELECT-Anweisungen über mehrere Prozeduren verteilen oder Spannweiten verringern.

305 Codeüberlauf

Der insgesamt erzeugte sichtbare Code ist zu umfangreich (> 5 MB). Keine Abhilfe.

307 Die benutzerdefinierten und vom Compiler (für Zwischenergebnisse) angelegten lokalen Daten einer Prozedur belegen zu viel Platz. Keine Abhilfe.

309 Zuviele Paketdaten

(Insgesamt mehr als 8192K Paketdaten) Keine Abhilfe möglich.

314 Zu viele »aktive« Label.

Es gibt zu viele Vorwärtssprünge, die noch nicht abgesättigt wurden. Es gibt eventuell zu viele geschachtelte `IF`, `SELECT` oder `REP` Anweisungen. Das Programm sollte in mehr Prozeduren gegliedert werden.

Falls eine hier nicht aufgeführte Fehlermeldungsnummer von Compiler ausgegeben wird, liegt ein interner Fehler vor. Setzen Sie sich in diesem Falle bitte mit der ACCOMMODAT GmbH in Verbindung.

3 Einfache Datentypen

3.1 Ganze Zahlen

3.1.1 Darstellung ganzer Zahlen

Der Datentyp INTEGER, der eine Untermenge der ganzen Zahlen repräsentiert, ist ein elementarer Typ der Sprache ELAN. Der Wertebereich der 32 Bit Repräsentation liegt zwischen −2 147 483 648 und 2 147 483 647.

Neben dem elementaren Typ Integer gibt es weitere Ganzzahltypen mit eingeschränktem oder erweitertem Wertebereich:

TYPE BYTE: Vorzeichenloses Byte (8 Bit).

TYPE WORD: Vorzeichenloses Wort (16 Bit).

TYPE HINT: Vorzeichenbehaftetes Wort (16 Bit). Wertebereich `(-32768 bis 32767 )`.

TYPE INT: Vorzeichenbehaftetes Doppelwort (32 Bit). Wertebereich `(-2147483648 bis 2147483647 )`.

TYPE DWORD: Vorzeichenloses Doppelwort (32 Bit).

TYPE DINT: Vorzeichenbehaftetes Quadwort (64 Bit). Wertebereich: `(-9223372036854775808 bis 9223372036854775807 )`.

Die interne Darstellung dieser Typen erfolgt durch die 80386 Datentypen BYTE, WORD, DWORD und QUADWORD. Der Speicherplatzbedarf ist somit für jeden Datentyp der minimal erforderliche.

Für die »üblichen« Ganzzahltypen H(alf) Integer, Integer und D(ouble) Integer sind alle üblichen arithmetischen Operationen bereitgestellt. Sollen Bytes, Worte oder Doppelworte für Berechnungen herangezogen werden, sind die passenden Konvertierungsprozeduren zu benutzen, um sie in Integer umzuwandeln.

Die Notation von Integern kann zur Basis 2, zur Basis 10 oder hexadezimal erfolgen. Binärnotation wird durch das Präfix `0b`, hexadezimale Notation durch das Präfix `0x` angezeigt.

Beispiel:

```
INT CONST i := 0x ff ,
          j := 255 ,
          k := 0b 1111 1111 ;
```

Hexadezimal- und Binärwerte werden nicht auf Integer Überlauf geprüft.

Nur INTEGER können in Zählschleifen (FOR Konstrukte), Selektionen (SELECT) oder zur Subskription herangezogen werden.

3.1.2 Operationen für ganze Zahlen

Integer (32 Bit)

INT	Typ
:=	Zuweisung
=	Vergleich
<>	Vergleich
<	Vergleich
<=	Vergleich
>	Vergleich
>=	Vergleich
+	Addition
-	Subtraktion
	Multiplikation
*	Exponentiation
abs	Absolutbetrag
ABS	Absolutbetrag
byte	Konvertierung
DECR	Subtraktion
dint	Konvertierung
DIV	Division
hint	Konvertierung
INCR	Addition
initializerandom	Zufallszahl initialisieren
max	Maximum
max int	Konstante
min	Minimum
min int	Konstante
MOD	Divisionsrestklasse
random	Zufallszahl liefern
REM	Divisionsrest
sign	Vorzeichen
SIGN	Vorzeichen
text	Konvertierung
word	Konvertierung

Double Integer (64 Bit)

DINT	Typ
:=	Zuweisung
=	Vergleich
<>	Vergleich
<	Vergleich
<=	Vergleich
>	Vergleich
>=	Vergleich
+	Addition
-	Subtraktion
	Multiplikation
abs	Absolutbetrag
DECR	Subtraktion
dint1	Einselement
dint0	Nullelement
DIV	Division
INCR	Addition
int	Konvertierung
max	Maximum
max dint	Konstante
min	Minimum
min dint	Konstante
MOD	Divisionsrestklasse
sign	Vorzeichen
text	Konvertierung

Half Integer (16 Bit)

HINT	Typ
:=	Zuweisung
=	Vergleich
<>	Vergleich
<	Vergleich
<=	Vergleich
>	Vergleich
>=	Vergleich
+	Addition
-	Subtraktion
	Multiplikation
DECR	Subtraktion
DIV	Division
INCR	Addition
int	Konvertierung
max	Maximum
max hint	Konstante
min	Minimum
min hint	Konstante
MOD	Divisionsrestklasse

Doppelworte

DWORD	Typ
:=	Zuweisung
dword	Konvertierung

Worte

WORD	Typ
:=	Zuweisung
int	Konvertierung

Bytes

BYTE	Typ
:=	Zuweisung
int	Konvertierung

3.1.3 INTEGER

INT

TYPE INT

:=

INT OP := (INT VAR a, INT CONST b)

Wirkung: Zuweisung. Kopie des Wertes von 'b' in die Variable 'a'.

=

BOOL OP = (INT CONST a, b)

Wirkung: Liefert 'TRUE' , falls 'a' gleich 'b' ist.

<>

BOOL OP <> (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' ungleich 'b' ist.

<

BOOL OP < (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' kleiner als 'b' ist.

<=

BOOL OP <= (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' kleiner oder gleich 'b' ist.

>

BOOL OP > (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer als 'b' ist.

>=

BOOL OP >= (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer oder gleich 'b' ist.

+

INT OP + (INT CONST a)

Wirkung: Monadischer Operator (Vorzeichen, ohne Wirkung).

INT OP + (INT CONST a, b)

Wirkung: Addition.

Fehler: "INT-Überlauf". Das Ergebnis der Addition ist größer als **max int** oder kleiner als **min int**.

-

INT OP - (INT CONST a)

Wirkung: Vorzeichen-Umkehrung.

INT OP - (INT CONST a, b)

Wirkung: Subtraktion.

Fehler: "INT-Überlauf". Das Ergebnis der Subtraktion ist größer als **max int** oder kleiner als **min int**.

*****

INT OP * (INT CONST a, b)

Wirkung: Multiplikation.

Fehler: "INT-Überlauf". Das Ergebnis der Multiplikation ist größer als **max int** oder kleiner als **min int**.

******

INT OP ** (INT CONST arg, exp)

Wirkung: Exponentiation mit 'exp' >= 0

Bemerkung: Ein negativer Exponent liefert das Ergebnis 0.

Fehler: `0**0` ist undefiniert. "INT-Überlauf".

abs

INT PROC abs (INT CONST argument)

Wirkung: Absolutbetrag des INT-Wertes.

ABS

INT OP ABS (INT CONST argument)

Wirkung: Absolutbetrag des INT-Wertes.

byte

BYTE PROC byte (INT CONST int)

Wirkung: Liefert den Wert des niederwertigsten Bytes von 'int'.

DECR

OP DECR (INT VAR left, INT CONST right)

Wirkung: Subtraktion. Entspricht `left := left - right`.

dint

DINT PROC dint (INT CONST int)

Wirkung: Liefert den entsprechenden DINT Wert zu 'int'.

DIV

INT OP DIV (INT CONST a, b)

Wirkung: Ganzzahldivision.

Fehler: DIV durch 0

dword

DWORD PROC dword (INT CONST int)

Wirkung: Liefert den entsprechenden DWORD Wert zu 'int'.

hint

HINT PROC hint (INT CONST int)

Wirkung: Liefert den entsprechenden HINT Wert zu 'int'.

Fehler: "INT Überlauf". Der Wert von 'int' liegt nicht zwischen -32768 und 32767.

INCR

OP INCR (INT VAR left, INT CONST right)

Wirkung: Addition. Entspricht `left := left + right`.

initialize random

PROC initialize random (INT CONST n)

Wirkung: Initialisierung des Generators für (Pseudo) Zufallszahlen. Die Zufallszahlen werden durch sukzessiven Aufruf der Prozedur **random** geliefert. Falls der Wert des Parameters 'n' erneut für **initialize random** benutzt wird, wird die vorher gelieferte Reihe von Zufallszahlen reproduziert.

max

INT PROC max (INT CONST left, right)

Wirkung: Liefert das Maximum der beiden Werte.

max int

INT CONST max int

Wirkung: Liefert den größten Wert, den der Typ INTEGER im L3 System annehmen kann (2 147 483 647).

min

INT PROC min (INT CONST left, right)

Wirkung: Liefert das Minimum der beiden Werte.

min int

INT CONST min int

Wirkung: Liefert den kleinsten Wert, den der Typ INTEGER im L3 System annehmen kann (-2 147 483 648).

MOD

INT OP MOD (INT CONST left, right)

Wirkung: Liefert die Divisionsrestklasse, vergl. **REM**.

Beispiel: `3 MOD 2 ==> 1`
`-3 MOD 2 ==> 1`

Fehler: DIV durch 0

random

INT PROC random (INT CONST lower bound, upper bound)

Wirkung: Liefert eine (Pseudo) Zufallszahl im Intervall zwischen 'lower bound' und 'upper bound' und wird mittels 'initialize random' initialisiert.

REM

INT OP REM (INT CONST left, right)

Wirkung: Liefert den Divisionsrest, vergl.**MOD**.

Beispiel: `3 REM 2 ==> 1`
`-3 REM 2 ==> -1`

sign

INT PROC sign (INT CONST argument)

Wirkung: Feststellen des Vorzeichens eines INT-Wertes.

```
argument > 0 ==> 1
argument = 0 ==> 0
argument < 0 ==> -1
```

SIGN

INT OP SIGN (INT CONST argument)

Wirkung: Feststellen des Vorzeichens eines INT-Wertes.

```
argument > 0 ==> 1
argument = 0 ==> 0
argument < 0 ==> -1
```

text

TEXT PROC text (INT CONST number)

Wirkung: Liefert die Darstellung von 'number' als ASCII Zeichenfolge. Falls 'number' einen negativen Wert hat, wird als Vorzeichen '-' vorangestellt.

TEXT PROC text (INT CONST number, length)

Wirkung: Konvertierung von 'number' in einen Text der Länge 'length'. Falls der Text kürzer als 'length' ist, wird er links (vorne) mit Leerzeichen aufgefüllt, falls er länger ist wird 'length' mal "*" geliefert.

Beispiel:

```
out ("X:"); out(text(12345,7)) ; line;
out ("Y:"); out(text(12345,3)) ;
(* ergibt *)
X:  12345
Y:***
```

word

WORD PROC word (INT CONST int)

Wirkung: Liefert den Wert des niederwertigsten Wortes zu 'int'.

Fehler: "INT Überlauf". Der Wert von 'int' liegt nicht zwischen 0 und 65535.

3.1.4 DOUBLE INTEGER

DINT

TYPE DINT

:=

DINT OP := (DINT VAR a, DINT CONST b)

Wirkung: Zuweisung. Kopie des Wertes von 'b' in die Variable 'a'.

=

BOOL OP = (DINT CONST a, b)

Wirkung: Liefert 'TRUE' , falls 'a' gleich 'b' ist.

<>

BOOL OP <> (DINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' ungleich 'b' ist.

<

BOOL OP < (DINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' kleiner als 'b' ist.

<=

BOOL OP <= (DINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' kleiner oder gleich 'b' ist.

>

BOOL OP > (DINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer als 'b' ist.

>=

BOOL OP >= (DINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer oder gleich 'b' ist.

+

DINT OP + (DINT CONST a, b)

Wirkung: Addition.

Fehler: "INT-Überlauf". Das Ergebnis der Addition ist größer als **max dint** oder kleiner als **min dint**.

-

DINT OP - (DINT CONST a, b)

Wirkung: Subtraktion.

Fehler: "INT-Überlauf". Das Ergebnis der Subtraktion ist größer als **max dint** oder kleiner als **min dint**.

*****

DINT OP * (DINT CONST a, b)

Wirkung: Multiplikation.

Fehler: "INT-Überlauf". Das Ergebnis der Multiplikation ist größer als **max dint** oder kleiner als **min dint**.

abs

DINT PROC abs (DINT CONST argument)

Wirkung: Absolutbetrag des DINT-Wertes.

DECR

OP DECR (DINT VAR left, DINT CONST right)

Wirkung: Subtraktion. Entspricht `left := left - right`.

dint0

DINT PROC dint0

Wirkung: Liefert DINT Wert '0'.

dint1

DINT PROC dint1

Wirkung: Liefert DINT Wert '1'.

DIV

DINT OP DIV (DINT CONST a, b)

Wirkung: Ganzzahldivision.

Fehler: DIV durch 0

INCR
OP INCR (DINT VAR left, DINT CONST right)

Wirkung: Addition. Entspricht `left := left + right`.

int
INT PROC int (DINT CONST dint)

Wirkung: Liefert den Wert des niederwertigsten DWORDS von 'dint'.

max
DINT PROC max (DINT CONST left, right)

Wirkung: Liefert das Maximum der beiden Werte.

max dint
DINT CONST maxdint

Wirkung: Liefert den größten Wert, den der Typ DINT im L3 System annehmen kann.

min
DINT PROC min (DINT CONST left, right)

Wirkung: Liefert den kleineren der beiden Werte.

min dint
DINT CONST min dint

Wirkung: Liefert den kleinsten Wert, den der Typ DINT im L3 System annehmen kann.

MOD
DINT OP MOD (DINT CONST left, right)

Wirkung: Liefert die Divisionsrestklasse, vergl. **REM**.

Beispiel:

```
dint( 3) MOD dint (2) ==> 1
dint(-3) MOD dint (2) ==> 1
```

Fehler: DIV durch 0

sign

INT PROC sign (DINT CONST value)

Wirkung: Vorzeichen eines DINT Wertes.

text

TEXT PROC text (DINT CONST number)

Wirkung: Liefert die Darstellung von ’number’ als ASCII Zeichenfolge. Falls ’number’ einen negativen Wert hat, wird als Vorzeichen ’-’ vorangestellt.

3.1.5 HALF INTEGER

HINT

TYPE HINT

:=

HINT OP := (HINT VAR a, HINT CONST b)

Wirkung: Zuweisung. Kopie des Wertes von ’b’ in die Variable ’a’.

=

BOOL OP = (HINT CONST a, b)

Wirkung: Liefert ’TRUE’ , falls ’a’ gleich ’b’ ist.

<>

BOOL OP <> (HINT CONST a, b)

Wirkung: Liefert ’TRUE’, falls ’a’ ungleich ’b’ ist.

<

BOOL OP < (HINT CONST a, b)

Wirkung: Liefert ’TRUE’, falls ’a’ kleiner als ’b’ ist.

<=

BOOL OP <= (HINT CONST a, b)

Wirkung: Liefert ’TRUE’, falls ’a’ kleiner oder gleich ’b’ ist.

>

BOOL OP > (HINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer als 'b' ist.

>=

BOOL OP >= (HINT CONST a, b)

Wirkung: Liefert 'TRUE', falls 'a' größer oder gleich 'b' ist.

+

HINT OP + (HINT CONST a, b)

Wirkung: Addition.

Fehler: "INT-Überlauf". Das Ergebnis der Addition ist größer als **maxhint** oder kleiner als **minhint**.

-

HINT OP - (HINT CONST a, b)

Wirkung: Subtraktion.

Fehler: "INT-Überlauf". Das Ergebnis der Subtraktion ist größer als **maxhint** oder kleiner als **minhint**.

*****

HINT OP * (HINT CONST a, b)

Wirkung: Multiplikation.

Fehler: "INT-Überlauf". Das Ergebnis der Multiplikation ist größer als **maxhint** oder kleiner als **minhint**.

DECR

OP DECR (HINT VAR left, HINT CONST right)

Wirkung: Subtraktion. Entspricht `left := left - right`.

DIV

HINT OP DIV (HINT CONST a, b)

Wirkung: Ganzzahldivision.

Fehler: DIV durch 0

INCR

OP INCR (HINT VAR left, HINT CONST right)

Wirkung: Addition. Entspricht `left := left + right`.

int

INT PROC int (HINT CONST hint)

Wirkung: Liefert den entsprechenden INT Wert zu 'hint'.

max

HINT PROC max (HINT CONST left, right)

Wirkung: Liefert das Maximum der beiden Werte.

maxhint

HINT CONST maxhint

Wirkung: Liefert den größten Wert, den der Typ HINT im L3 System annehmen kann.

min

HINT PROC min (HINT CONST left, right)

Wirkung: Liefert das Minimum der beiden Werte.

minhint

HINT CONST minhint

Wirkung: Liefert den kleinsten Wert, den der Typ HINT im L3 System annehmen kann.

MOD

HINT OP MOD (HINT CONST left, right)

Wirkung: Liefert die Divisionsrestklasse, vergl **MOD**.

Beispiel:
```
hint( 3) MOD hint (2) ==> hint (1)
    hint(-3) MOD hint (2) ==> hint (1)
```

Fehler: DIV durch 0

3.1.6 DWORD

DWORD

TYPE DWORD

:=

OP := (DWORD VAR a, DWORD CONST b)

Wirkung: Zuweisung. Kopie des Wertes von 'b' in die Variable 'a'.

int

INT PROC int (DWORD CONST word)

Wirkung: Liefert den entsprechenden INT Wert zu 'dword'.

3.1.7 WORD

WORD

TYPE WORD

:=

OP := (WORD VAR a, WORD CONST b)

Wirkung: Zuweisung. Kopie des Wertes von 'b' in die Variable 'a'.

int

INT PROC int (WORD CONST word)

Wirkung: Liefert den entsprechenden INT Wert zu 'word'.

3.1.8 BYTE

BYTE

TYPE BYTE

:=

OP := (BYTE VAR a, BYTE CONST b)

Wirkung: Zuweisung. Kopie des Wertes von 'b' in die Variable 'a'.

int

INT PROC int (BYTE CONST byte)

Wirkung: Liefert den entsprechenden INT Wert zu 'byte'.

3.2 Boolesche Verknüpfung und Bitmasken

3.2.1 Einführung

Der Wertebereich für Datenobjekte vom Type BOOL besteht aus TRUE und FALSE. Werte des Typs BOOL können mit folgenden Operatoren verknüpft werden.

AND	Logisches UND
CAND	Bedingtes UND
COR	Bedingtes ODER
false	Konstante
FALSE	Denoter
NOT	Negation
OR	Logisches ODER
true	Konstante
TRUE	Denoter
XOR	Exklusives ODER

3.2.2 Elementare Verknüpfungen

AND

BOOL OP AND (BOOL CONST a, b)

Wirkung: Logisches UND. Die Reihenfolge der Auswertung ist unbestimmt. Wenn der Operand, der zuerst ausgewertet wird, bereits FALSE liefert, wird der zweite Operand nicht mehr ausgewertet.

CAND

BOOL OP CAND (BOOL CONST a, b)

Wirkung: Bedingtes logisches UND. Der zweite Operand wird nicht ausgewertet, falls er für das Ergebnis nicht relevant ist. Entspricht: `IF a THEN b ELSE false FI`.

Bemerkung: Durch die festgelegte Reihenfolge der Auswertung kann z.B zuerst überprüft werden, ob ein Index innerhalb gewisser Grenzen liegt, bevor er zur Subscription herangezogen wird.

Beispiel:

```
i > 0 CAND row [i] = 1
```

COR

BOOL OP COR (BOOL CONST a, b)

Wirkung: Bedingtes logisches ODER. Der zweite Operand wird nicht ausgewertet, falls er für das Ergebnis nicht relevant ist. Entspricht: `IF a THEN true ELSE b FI.`

false

BOOL CONST false

Wirkung: Boolesche Konstante

NOT

BOOL OP NOT (BOOL CONST a)

Wirkung: `IF a THEN FALSE ELSE TRUE FI`

OR

BOOL OP OR (BOOL CONST a, b)

Wirkung: Logisches ODER. Die Reihenfolge der Auswertung ist unbestimmt. Wenn der Operand, der zuerst ausgewertet wird, bereits TRUE liefert, wird der zweite Operand nicht mehr ausgewertet.

true

BOOL CONST true

Wirkung: Boolesche Konstante

XOR

BOOL OP XOR (BOOL CONST a, b)

Wirkung: Exklusives ODER, liefert TRUE, wenn 'a' und 'b' ungleich, und FALSE, wenn 'a' und 'b' gleich sind.

3.2.3 Operationen auf Bitmasken

Bit-Operationen arbeiten in L3 auf Integer-Objekten. Sie können benutzt werden, um Bitmasken zu erzeugen. Ein Integer hat 32 Bit. Das höchstwertige Bit hat die Nummer 31, das niederwertigste die Nummer 0.

Ebenfalls vordefiniert sind einige dieser Operationen für DINTs, so daß auch Bitmasken der Länge 64 benutzt werden können.

AND	Bitweise UND Verknüpfung
bit	Bit Test
lowest set	Unterstes Eins Bit
lowest reset	Unterstes Null Bit
off	Test
on	Test
OR	Bitweise ODER Verknüpfung
SHL	Linksshift
SHR	Rechtsshift
reset bit	Bit auf Null setzen
rotate	Ringshift
set bit	Bit auf Eins setzen
XOR	Bitweise Exklusiv ODER Verknüpfung

AND

INT OP AND (INT CONST a, b)

Wirkung: Bitweise UND-Verknüpfung von 'a' mit 'b'.

DINT OP AND (DINT CONST a, b)

Wirkung: Bitweise UND-Verknüpfung von 'a' mit 'b'.

bit

BOOL PROC bit (INT CONST bits, bit no)

Wirkung: Liefert TRUE, wenn das Bit mit der Nummer 'bit no' in dem INT 'bits' gesetzt ist.

Beispiel:

```
IF bit (1024,10) THEN put("TRUE") FI  --->  TRUE
```

lowest set

INT PROC lowest set (INT CONST bits)

Wirkung: Liefert die Nummer des niederwertigsten auf 1 gesetzten Bits in 'bits'. Ist kein Bit auf 1 gesetzt, wird -1 geliefert.

lowest reset

lowest reset (INT CONST bits)

Wirkung: Liefert die Nummer des niederwertigsten auf 0 gesetzten Bits in 'bits'. Ist kein Bit auf 0 gesetzt, wird -1 geliefert.

off

BOOL PROC off (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls `(a AND b) = 0` ist.

on

BOOL PROC on (INT CONST a, b)

Wirkung: Liefert 'TRUE', falls `(a AND b) <> 0` ist.

OR

INT OP OR (INT CONST a, b)

Wirkung: Bitweise ODER-Verknüpfung von 'a' mit 'b'.

DINT OP OR (DINT CONST a, b)

Wirkung: Bitweise ODER-Verknüpfung von 'a' mit 'b'.

SHL

INT OP SHL (INT CONST a, b)

Wirkung: Linksshift von 'a' um 'b' Bit.

Beispiel: Schnelle Multiplikation mit 4:`17 SHR 4`

SHR

INT OP SHR (INT CONST a, b)

Wirkung: Rechtsshift von 'a' um 'b' Bit.

Beispiel: Schnelle Division durch 4:`17 SHR 4`

reset bit

PROC reset bit (INT VAR bits, INT CONST bit no)

Wirkung: Das Bit mit der Nummer 'bit no' wird in 'bits' auf 0 gesetzt.

rotate

PROC rotate (INT VAR bits, INT CONST factor)

Wirkung: Bits werden mit dieser Prozedur zyklisch geshiftet. Die Richtung der Bitverschiebung wird durch das Vorzeichen von 'factor' bestimmt. Ein positives Vorzeichen bewirkt Linksshift, ein negatives Rechtsshift.

Bemerkung: Da zyklisch geshiftet wird, bewirkt ein Wert von 'factor' größer als 32 einen mehrfachen Zyklus.

Beispiel:

```
rotate (1,1)     ---> 2
rotate (1,2)     ---> 4
rotate (1,-1)    ---> minint
rotate (1,65)    ---> 2
```

set bit

PROC set bit (INT VAR bits, INT CONST bit no)

Wirkung: Das Bit mit der Nummer 'bit no' wird in 'bits' auf 1 gesetzt.

XOR

INT OP XOR (INT CONST a, b)

Wirkung: Bitweise EXCLUSIV-ODER-Verknüpfung von 'a' mit 'b'.

DINT OP XOR (DINT CONST a, b)

Wirkung: Bitweise EXCLUSIV-ODER-Verknüpfung von 'a' mit 'b'.

3.3 Realzahlen

3.3.1 Einführung

Realzahlen sind ein elementarer Datentyp der Sprache ELAN. Realzahlen werden in ELAN Programmen in Punktnotation (z.B. 12.345), ggf. in Exponentialnotation (z.B. 1.2345e10) aufgeschrieben. Die Realzahlen stellen eine Untermenge der reellen Zahlen dar. Die interne Darstellung ist die binäre 8 Byte-Fließkommadarstellung gemäß der Norm IEEE 754. Sie entspricht im Hinblick auf interne Zahlendarstellung damit dem für den INTEL 80387 geforderten Format. Dieses Format erlaubt die Darstellung von Zahlen im ungefähren Bereich von `2.2e-308` bis `1.8e308`. Die Darstellung ist auf 15 bis 16 Dezimalstellen genau.

Wenn vorhanden, wird ein Coprozessor 80287/80387 benutzt. Falls dieser nicht vorhanden ist, wird die standardmäßig in den L3 Systemkern eingebundene Coprozessoremulation benutzt.

3.3.2 Operationen für Realzahlen

:=	Zuweisung
=	Vergleich (gleich)
<>	Vergleich (ungleich)
<	Vergleich (kleiner)
<=	Vergleich (kleiner oder gleich)
>	Vergleich (größer)
>=	Vergleich (größer oder gleich)
+	Addition
-	Subtraktion
	Multiplikation
/	Division
*	Exponentiation
abs	Absolutbetrag
ABS	Absolutbetrag
arctan	Arcus Tangens (Radiant)
arctand	Arcus Tangens (Grad)
cos	Cosinus (Radiant)
cosd	Cosinus (Grad)
decimal exponent	Dezimalexponent
DECR	Dekrementierung
e	Konstante
exp	Exponentiation
floor	Vorkommastellen
frac	Nachkommastellen
INCR	Inkrementierung
initialize random	Zufallszahl initialisieren
int	Konvertierung
ln	Natürlicher Logarithmus
log2	Logarithmus zur Basis 2
log10	Logarithmus zur Basis 10
maxreal	Konstante
max	Maximum
min	Minimum
MOD	Restklasse
pi	Konstante
random	Zufallszahl
round	Rundung
sign	Vorzeichen
SIGN	Vorzeichen
sin	Sinus (Radiant)
sind	Sinus (Grad)
smallreal	Konstante
sqrt	Quadratwurzel
tan	Tangens (Radiant)
tand	Tangens (Grad)
text	Konvertierung
underflow	Wert nicht darstellbar

:=

REAL OP := (REAL VAR a, REAL CONST b)

Wirkung: Zuweisung.

=

BOOL OP = (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a = b

<>

BOOL OP <> (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a <> b

<

BOOL OP < (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a < b

<=

BOOL OP <= (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a <= b

>

BOOL OP > (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a > b

>=

BOOL OP >= (REAL CONST a, b)

Wirkung: Liefert den Wahrheitwert der Aussage a >= b

+

REAL OP + (REAL CONST r)

Wirkung: Monadischer Operator

REAL OP + (REAL CONST a, b)

Wirkung: Liefert die Summe a + b (Addition).

-

REAL OP - (REAL CONST r)

Wirkung: Vorzeichen-Umkehrung.

REAL OP - (REAL CONST a, b)

Wirkung: Liefert die Differenz a - b (Subtraktion).

*****

REAL OP * (REAL CONST a, b)

Wirkung: Liefert das Produkt a * b (Multiplikation).

/

REAL OP / (REAL CONST a, b)

Wirkung: Liefert den Quotienten a / b (Division).

Fehler: Division durch 0

******

REAL OP ** (REAL CONST base, power)

Wirkung: Exponentiation.

Fehler: base < 0.0 0.0 ** 0.0

REAL OP ** (REAL CONST base, INT CONST power)

Wirkung: Exponentiation.

Fehler: 0.0 * 0

abs

REAL PROC abs (REAL CONST a)

Wirkung: Absolutbetrag eines REAL-Wertes.

ABS

REAL OP ABS (REAL CONST a)

Wirkung: Absolutbetrag eines REAL-Wertes.

arctan

REAL PROC arctan (REAL CONST r)

Wirkung: Arcus Tangens Funktion. Liefert einen Wert im Bogenmaß.

arctand

REAL PROC arctand (REAL CONST r)

Wirkung: Arcus Tangens Funktion. Liefert einen Wert in Grad.

cos

REAL PROC cos (REAL CONST r)

Wirkung: Cosinus Funktion. 'r' muß im Bogenmaß angegeben werden.

cosd

REAL PROC cosd (REAL CONST r)

Wirkung: Cosinus Funktion. 'r' muß in Grad angegeben werden.

decimal exponent

INT PROC decimal exponent (REAL CONST r)

Wirkung: Liefert für den REAL-Wert 'r' den dezimalen Exponenten 'z', so daß gilt: `r = y * 10 ** z` und `1.0 <= y < 10.0`.

DECR

OP DECR (REAL VAR a, REAL CONST b)

Wirkung: Wirkt wie `a := a - b`

e

REAL PROC e

Wirkung: Liefert die Eulersche Zahl (ca. 2.718282).

exp

REAL PROC exp (REAL CONST x)

Wirkung: Exponentialfunktion.

floor

REAL PROC floor (REAL CONST real)

Wirkung: Schneidet die Stellen des REAL-Wertes 'real' nach dem Dezimalpunkt ab.

frac

REAL PROC frac (REAL CONST r)

Wirkung: Liefert die Stellen eines REAL-Wertes hinter dem Dezimalpunkt.

INCR

OP INCR (REAL VAR a, REAL CONST b)

Wirkung: Wirkt wie `a := a + b`

initialize random

PROC initialize random (REAL CONST z)

Wirkung: Initialisierung des Generators für (Pseudo) Zufallszahlen im Intervall zwischen 0.0 und 1.0. Die Zufallszahlen werden durch sukzessiven Aufruf der Prozedur **random** geliefert. Falls derselbe Wert des Parameters 'z' erneut für **initialize random** benutzt wird, wird die vorher gelieferte Reihe von Zufallszahlen reproduziert.

int

INT PROC int (REAL CONST x)

Wirkung: Konvertierung in eine ganze Zahl. Die Nachkommastellen werden abgeschnitten.

Beispiel: `int (3.9) ==> 3`

ln

REAL PROC ln (REAL CONST x)

Wirkung: Natürlicher Logarithmus.

Fehler: x <= 0.0

log2

REAL PROC log2 (REAL CONST x)

Wirkung: Logarithmus zur Basis 2.

Fehler: x <= 0.0

log10

REAL PROC log10 (REAL CONST x)

Wirkung: Logarithmus zur Basis 10.

Fehler: x <= 0.0

max

REAL PROC max (REAL CONST a, b)

Wirkung: Liefert das Maximum der beiden REAL-Werte.

maxreal

REAL CONST maxreal

Wirkung: Liefert den größten REAL-Wert im L3-System.

min

REAL PROC min (REAL CONST a, b)

Wirkung: Liefert das Minimum der beiden REAL-Werte.

MOD

REAL OP MOD (REAL CONST a, b)

Wirkung: Liefert den Rest einer REAL Division (Restklasse).

Beispiel: `5.0 MOD 2.0 ==> 1.0`
```
4.5 MOD 4.0 ==> 0.5
4.5 MOD 4.4 ==> 0.1
```

pi

REAL CONST pi

Wirkung: Liefert die Zahl pi (ca. 3.141593).

random

REAL PROC random

Wirkung: Liefert eine (Pseudo) Zufallszahl im Intervall zwischen 0.0 und 1.0 und wird zuvor mit **initialize random** initialisiert.

round

REAL PROC round (REAL CONST real, INT CONST digits)

Wirkung: Runden eines REAL-Wertes auf 'digits' Stellen. Für positive Werte wird auf Stellen hinter dem Dezimalpunkt, für negative Werte von 'digits' auf Stellen vor dem Dezimalpunkt gerundet.

Beispiel: `round (3.14159, 3) ==> 3.142`
`round (123.456, -2) ==> 100.0`

sign

INT PROC sign (REAL CONST argument)

Wirkung: Feststellen des Vorzeichens eines REAL-Wertes.

INT OP SIGN (REAL CONST argument)

Wirkung: Feststellen des Vorzeichens eines REAL-Wertes.

`argument > 0 ==> 1 argument = 0 ==> 0 argument < 0 ==> -1`

sin

REAL PROC sin (REAL CONST r)

Wirkung: Sinus Funktion. 'r' muß im Bogenmaß angegeben werden.

sind

REAL PROC sind (REAL CONST r)

Wirkung: Sinus Funktion. 'r' muß im Grad angegeben werden.

smallreal

REAL PROC smallreal

Wirkung: Kleinster REAL-Wert im L3 System für den gilt:

```
1.0 - smallreal <> 1.0
1.0 + smallreal <> 1.0
```

sqrt

REAL PROC sqrt (REAL CONST x

Wirkung: Wurzel Funktion.

Fehler: x < = 0.0

tan

REAL PROC tan (REAL CONST r)

Wirkung: Tangens Funktion. 'r' muß im Bogenmaß angegeben werden.

tand

REAL PROC tand (REAL CONST r)

Wirkung: Tangens Funktion. 'd' muß in Grad angegeben werden.

text

TEXT PROC text (REAL CONST real)

Wirkung: Konvertierung eines REAL-Wertes in einen TEXT der maximal 14 Zeichen umfaßt in Genauigkeit von maximal 7 Stellen hinter dem Dezimalpunkt. Ggf. wird der TEXT in Exponentialnotation geliefert.

TEXT PROC text (REAL CONST real, INT CONST length)

Wirkung: Konvertierung eines REAL-Wertes in einen TEXT der Länge 'length' in Exponentialnotation. 'length' muß zwischen 8 und 25 liegen. Bei Werten kleiner 8 wird 'length' mal "*" ausgegeben, bei zu großen Werten wird 25 angenommen.

TEXT PROC text (REAL CONST real, INT CONST length, fracs)

Wirkung: Konvertierung eines REAL-Wertes in einen TEXT in Punktnotation. Dabei gibt 'length' die Länge des Resultats einschließlich des Dezimalpunktes und des Vorzeichens, 'fracs' die Anzahl der Stellen nach dem Dezimalpunkt an. Kann der REAL Wert nicht wie gewünscht dargestellt werden, wird 'length' mal "*" geliefert.

underflow

BOOL PROC underflow

Wirkung: Liefert TRUE, falls die letzte REAL Operation einen Wertunterlauf erzeugte. Ein Unterlauf entsteht, wenn das Resultat einen so kleinen Wert annimmt, daß eine korrekte interne Darstellung nicht mehr möglich ist. Wenn dieser Fall eintritt, gilt `underflow = true`. Das Resultat der Operation wird auf 0.0 gesetzt.

3.4 Texte

3.4.1 Einführung

TEXT ist ein elementarer ELAN Datentyp, der Textobjekte variabler Länge aufnehmen kann. ELAN kennt keine Unterscheidung der Typen "Character" (einzelnes Zeichen) und "String" (Zeichenkette). Sowohl der leere Text (**niltext**) als auch einzelne Zeichen ("A", "B" usw.) bis hin zur Zeichenkette der Länge 32000 (**maxtextlength**) sind TEXTe.

Die Verwaltung von Textobjekten in Programmen zur Laufzeit ist dynamisch. Interna über diese Verwaltung finden Sie am Ende dieses Abschnitts.

3.4.2 Vergleiche von Texten

Texte werden nach den üblichen lexikalischen Regeln verglichen. Die Ordnung der Zeichen ist durch die interne Darstellung, also die ASCII Codierung der Zeichen bestimmt. Unter der Annahme 'a' < 'b' sind weiterhin folgende Ausdrücke wahr:

```
"" < "a"
"a" < "b"
"a" < "ab"
"aa" < "b"
"aa" < "ab"
```

Für den lexikalischen Vergleich (die Telefonbuchordnung) werden Regeln gemäß DIN 5007 benutzt. Die Operatoren **LEXEQUAL**, **LEXGREATER** und **LEXGREATEREQUAL** prüfen nach folgenden Regeln:

- Buchstaben haben die aufsteigende Reihenfolge 'A' bis 'Z'. Dabei werden kleine und große Buchstaben gleich behandelt.
- Umlaute werden behandelt als wären sie ausgeschrieben. (Ä = Ae usw.) (ß = ss)
- Alle Sonderzeichen (auch Ziffern) außer ' ' (Leerzeichen) und '–' werden ignoriert, diese beiden Zeichen werden gleich behandelt.

3.4.3 ASCII Zeichensatz

Die ASCII Ordnung setzt also Ziffern vor Großbuchstaben, diese vor Kleinbuchstaben.

3.4.4 Operationen auf Texten

= <> <	Vergleiche
<= >= >	
+	Aneinanderhängen zweier Texte
	Vervielfachung eines Textes
CAT	Anfügen Text an Text
change	einmaliges Austauschen
changeall	mehrfaches Austauschen
code	Zeichen des Codes
code	Code des Zeichens
compress	Leerzeichen abschneiden
is digit	Code einer Ziffer
is lowercase letter	Code eines Kleinbuchstaben
is uppercase letter	Code eines Großbuchstaben
length	Liefert Textlänge
LENGTH	Liefert Textlänge
LEXEQUAL	Vergleiche
LEXGREATEREQUAL	— " —
LEXGREATER	— " —
lowercase	Text in Kleinschreibung wandeln
LOWERCASE	Liefert Text in Kleinschreibung
NORMEQUAL	Textgleichheit
maxtextlength	Konstante
pos	Position eines Teiltextes
replace	Ersetzen eines Teiltextes
SUB	Teiltext liefern
subtext	Teiltext liefern
text	Teiltext vorgegebener Länge liefern
uppercase	Text in Großschreibung wandeln
UPPERCASE	Liefert Text in Großschreibung
select container	Textcontainer anfordern
shorten	Teiltext aus Container

:=

TEXT OP := (TEXT VAR a, TEXT CONST b)

Wirkung: Zuweisung.

=

BOOL OP = (TEXT CONST left, right)

Wirkung: Vergleich von zwei Texten auf Gleichheit.

Bemerkung: Texte mit ungleichen Längen sind immer ungleich.

<>

BOOL OP <> (TEXT CONST left, right)

Wirkung: Vergleich zweier Texte auf Ungleichheit.

Bemerkung: Texte mit ungleichen Längen sind immer ungleich.

<

BOOL OP < (TEXT CONST left, right)

Wirkung: Liefert TRUE, falls der Text 'left' lexikographisch kleiner als 'right' ist.

<=

BOOL OP <= (TEXT CONST left, right)

Wirkung: Liefert TRUE, falls der Text 'left' lexikographisch kleiner oder gleich 'right' ist.

>

BOOL OP > (TEXT CONST left, right)

Wirkung: Liefert TRUE, falls der Text 'left' lexikographisch größer als 'right' ist.

>=

BOOL OP >= (TEXT CONST left, right)

Wirkung: Liefert TRUE, falls der Text 'left' lexikographisch größer oder gleich 'right' ist.

*****

TEXT OP * (INT CONST xtimes, TEXT CONST text)

Wirkung: 'text' wird 'xtimes' mal verkettet.

Bemerkung: Falls 'xtimes' kleiner oder gleich Null ist, wird niltext geliefert.

Beispiel: `put (3 * "schwarzer Kater ")`

+

TEXT OP + (TEXT CONST left, right)

Wirkung: Verkettung der Texte 'left' und 'right' in dieser Reihenfolge.

Bemerkung: Die Länge des Resultats ergibt sich aus der Addition der Länge der Operanden.

CAT

TEXT OP CAT (TEXT VAR left, TEXT CONST right)

Wirkung: Verknüpfung von 'left' und 'right'.

Bemerkung: Entspricht `left := left + right` CAT ist effizienter als die explizite Zuweisung.

change

PROC change (TEXT VAR text, TEXT CONST old, new)

Wirkung: Ersetzung des (Teil-) TEXTes 'old' in 'text' durch 'new' beim erstmaligen Auftreten.

Bemerkung: Ist 'old' nicht in 'text' vorhanden, so wird keine Meldung abgesetzt. Die Länge von 'text' kann sich bei dieser Operation verändern.

Beispiel:
```
TEXT VAR mein text :: "L3-Benutzerhandbuch";
change (mein text, "Ben", "N");
(* L3-Nutzerhandbuch *)
```

PROC change (TEXT VAR text, INT CONST from, to, TEXT CONST new)

Wirkung: Der TEXT 'new' wird in den TEXT 'text' anstatt des TEXTes, der zwischen 'from' und 'to' steht, eingesetzt. Die Länge von 'text' kann sich dabei verändern.

changeall

PROC changeall (TEXT VAR text, TEXT CONST old, new)

Wirkung: Ersetzt in 'text' jedes Auftreten von 'old' durch 'new'. Die Länge von 'text' kann sich dabei ändern.

code

TEXT PROC code (INT CONST code)

Wirkung: liefert das Zeichen, dessen ASCII Code angegeben ist.

Bemerkung: Es muß gelten:`0 <= code <= 255`.

INT PROC code (TEXT CONST text)

Wirkung: Wandelt ein Zeichen 'text' in einen INT-Wert um.

Bemerkung: Falls `length (text) <> 1` ist, wird der Wert -1 geliefert (also bei mehr als einem Zeichen oder niltext).

compress

TEXT PROC compress (TEXT CONST text)

Wirkung: Führende und abschließende Leerzeichen werden aus 'text' entfernt.

is digit

BOOL PROC is digit (INT CONST code)

Wirkung: Liefert TRUE, falls `(code (code))` eine Ziffer ist.

is lowercase letter

BOOL PROC is lowercase letter (TEXT CONST code)

Wirkung: Liefert TRUE, falls `(code (code))` ein Kleinbuchstabe ist.

is uppercase letter

BOOL PROC is uppercase letter (INT CONST code)

Wirkung: Liefert TRUE, falls `(code (code))` ein Großbuchstabe ist.

length

INT PROC length (TEXT CONST text)

Wirkung: Liefert die Länge (= Anzahl von Zeichen) von 'text' einschließlich Leerzeichen.

LENGTH

INT OP LENGTH (TEXT CONST text)

Wirkung: Liefert die Länge (= Anzahl von Zeichen) von 'text' einschließlich Leerzeichen.

LEXEQUAL

BOOL OP LEXEQUAL (TEXT CONST left, right)

Wirkung: Prüfung auf lexikalische Gleichheit.

Beispiel: `"Knödel" LEXEQUAL "Knoedel" (* => TRUE*)`
siehe **NORMEQUAL**.

LEXGREATER

BOOL OP LEXGREATER (TEXT CONST left, right)

Wirkung: Prüfung, ob der Text 'left' lexikalisch größer als 'right' ist.

LEXGREATEREQUAL

BOOL OP LEXGREATEREQUAL (TEXT CONST left, right)

Wirkung: Prüfung, ob der Text 'left' lexikalisch größer oder gleich dem Text 'right' ist.

lowercase

PROC lowercase (TEXT VAR text)

Wirkung: 'text' wird (ohne Kopie) in Kleinschreibung gewandelt. Jedes Zeichen aus 'text', dessen Code zwischen 65 und 90 liegt, also gerade die Großbuchstaben, wird in durch den entsprechenden Kleinbuchstaben ersetzt.

TEXT OP LOWERCASE (TEXT CONST text)

Wirkung: Liefert eine Textkopie von 'text', in der alle Großbuchstaben in Kleinbuchstaben gewandelt sind.

NORMEQUAL

BOOL OP NORMEQUAL (TEXT CONST left, right)

Wirkung: Liefert TRUE, falls `UPPERCASE (left) = UPPERCASE (right)`.

Beispiel: `"Knödel" NORMEQUAL "KNÖDEL" (* => TRUE*)`
`"Knödel" NORMEQUAL "Knoedel" (* => FALSE*)`
siehe **LEXEQUAL**.

max text length

INT PROC max text length

Wirkung: Liefert die maximale Anzahl von Zeichen in einem TEXT (32 000).

pos

INT PROC pos (TEXT CONST text, pattern)

Wirkung: Liefert die erste Position des ersten Zeichens von 'pattern' in 'text'.

Bemerkung: Wird 'pattern' nicht gefunden oder ist 'pattern' niltext, so wird der Wert '0' geliefert.

INT PROC pos (TEXT CONST text, pattern, INT CONST start)

Wirkung: Es wird erst ab der Position 'start' gesucht. Dabei gilt folgende Einschränkung: `length (pattern) < 255`

Beispiel:
```
TEXT VAR t1 :: "abcdefghijk...xyz",
t2 :: "cd";
... pos (t1, t2) ... (* liefert 3 *)
... pos (t2, t1) ... (* liefert 0 *)
```

INT PROC pos (TEXT CONST text, low char, high char, INT CONST start)

Wirkung: Liefert die Position des ersten Zeichens 'x' in 'text' ab der Position 'start'. 'x' soll kein eindeutig bestimmtes Zeichen sein, sondern ein Zeichen aus einer spezifizierten Menge von Zeichen. Diese Spezifikation geschieht durch 'low char' und 'high char'. Dadurch wird eine Zeichenmenge angegeben, deren (von der Codierung gesehen) kleinstes Zeichen 'low char' und deren größtes Zeichen 'high char' ist.

Bemerkung: 'low char' und 'high char' müssen TEXTe der Länge 1 sein. Wird kein Zeichen in 'text' in dem Bereich zwischen 'low char' und 'high char' gefunden, wird der Wert '0' geliefert.

Beispiel:

```
(* Suche nach Zeichen <> blank nach einer Leerspalte*)
TEXT VAR zeile :: "BlaBla       Hier gehts weiter";
INT VAR pos erstes blank :: pos (zeile, " "),
        ende leerspalte  ::
           pos (zeile,""33"",""254"", pos erstes blank);
```

replace

PROC replace (TEXT VAR text, INT CONST position, TEXT CONST new text)

Wirkung: Ersetzung eines (Teil-) Textes in 'text' durch 'new text' an der Position 'position' in 'text'.

Bemerkung: Es muß gelten `1 <= position <= length (text)` d.h. 'position' muß innerhalb von 'text' liegen und 'new text' muß von der Position 'position' ab in 'text' einsetzbar sein. Dabei bleibt die Länge von 'text' unverändert.

SUB

TEXT OP SUB (TEXT CONST text, INT CONST pos)

Wirkung: Liefert das Zeichen an der Stelle 'pos' aus 'text'. Entspricht `subtext (text, pos, pos)` , ist jedoch effizienter .

Bemerkung: Für pos <= 0 oder pos > length (text) wird niltext geliefert.

subtext

TEXT PROC subtext (TEXT CONST text, INT CONST from)

Wirkung: Liefert den Teiltext von 'text', der bei der Position 'from' anfängt.

Bemerkung: Die Länge des Resultats ergibt sich also aus `length (text) - from + 1` d.h. von der Position 'from' bis zum Ende von 'text'. 'from' muß innerhalb von 'text' liegen. Ist 'from' < 1, dann wird 'text' geliefert. Falls `from > length (text)` ist, wird **niltext** geliefert.

TEXT PROC subtext (TEXT CONST text, INT CONST from, to)

Wirkung: Der Teiltext von 'text' von der Position 'from' bis einschließlich der Position 'to' wird geliefert. Dabei muß gelten `1 <= from <= to <= length (text)` d.h. die Positionen 'from' und 'to' müssen in dieser Reihenfolge innerhalb von 'text' liegen. Ist `to >= length(text)` wird **subtext (text, from)** ausgeführt. Für die Bedingungen für 'from' siehe vorstehende Beschreibung von 'subtext'.

text

TEXT PROC text (TEXT CONST text, INT CONST length)

Wirkung: Liefert einen Teiltext aus 'text' mit der Länge 'length', beginnend bei der Position 1 von 'text'. Das Resultat hat auf jeden Fall die angegebene Länge.

TEXT PROC text (TEXT CONST text, INT CONST length, from)

Wirkung: Liefert einen Teiltext aus 'text' ab der Position 'from' mit der Länge 'length'.

Bemerkung: Es muß gelten `1 <= 'from' <= length (text)`. Für den Fall `from > length (text)` wird ein Text aus 'length' Leerzeichen geliefert. Für `from < 0` wird `text (text, length, 1)` ausgeführt.

uppercase

PROC uppercase (TEXT VAR text)

Wirkung: 'text' wird (ohne Kopie) in Großschreibung gewandelt. Jedes Zeichen aus 'text', dessen Code zwischen 97 und 122 liegt, also gerade die Kleinbuchstaben, wird in durch den entsprechenden Großbuchstaben ersetzt.

TEXT OP UPPERCASE (TEXT CONST text)

Wirkung: Liefert eine Textkopie von 'text', in der alle Kleinbuchstaben in Großbuchstaben gewandelt sind.

3.5 Interne Organisation von TEXTen

Jedes Datenobjekt vom Typ TEXT besteht aus einem festen Teil und möglicherweise aus einem flexiblen Teil auf dem **Textheap**. Im festen Teil werden Texte bis zur Länge von 11 Zeichen untergebracht. Wenn eine TEXT-Variable einen Wert mit mehr als 11 Zeichen Länge annimmt, werden alle Zeichen auf dem Heap untergebracht. Der Textheap ist ein ausschließlich der Speicherung von Texten gewidmeter Abschnitt eines Datenraums. Im Standarddatenraum ist der Textheap 14 MB groß. Er liegt im Adreßbereich ab `0x1200000`. Bei benutzerdefinierten Datenräumen sind Größe und Lage des Textheaps einstellbar (siehe **set min heapsize** Kap.5).

Der Textheap des Standarddatenraums besteht aus **Containern** verschiedener Größe zwischen 32 Bytes und 32 Kilobyte. Die 11 vorhandenen Containergrößen fassen 28 Bytes, 60 Bytes, 124 Bytes, 1*4 KBytes, 2*4 KBytes bis 8*4 KBytes. Der Textheap von anderen Datenräumen (`BOUND .... TEXT....`) einfacher organisiert (siehe unten).

Falls ein Textobjekt Platz auf dem Textheap benötigt, wird ihm dort der kleinstmögliche, passende Container zugewiesen. Der Platzbedarf von TEXTen richtet sich also nicht nach der reinen Textgröße, sondern nach der erforderlichen Containergröße.

3.5.1 Speicherung im Standarddatenraum

Textdenoter erhalten eine Sonderbehandlung. Ein Textdenoter 'd' wird im Standarddatenraum mit den Länge `len (d) + 1` im Datenbereich des ELAN Programms gespeichert.

```
len | 'len' text bytes..
 0     1              ..254
```

Im ersten Byte ist die Länge des Denoters vermerkt, gefolgt vom Denoter selbst in einer Länge zwischen 0 bis 254 Bytes.

Kurze Textvariablen bis zu einer Länge von 11 Byte werden Texte ohne Benutzung des Textheaps gespeichert.

```
len |  text bytes     | heaplink
 0     1               12      15
```

Nicht benutzte Textbytes und 'heaplink' sind undefiniert.

Lange Textvariablen Das 'heaplink' Feld enthält die 32 Bit Adresse des Containers in dem der Text gespeichert ist.

```
0xff | xx | len | xxxx | heaplink
 0     1    2    4       12      15
```

Textcontainer im Standarddatenraum Der Heap im Standarddatenraum ist in 11 Klassen von Textcontainern gegliedert. Die ersten drei Klassen werden durch einen zusammenhängenden Abschnitt gebildet. In diesem Abschnitt zur Verfügung:

Containerlänge	Max.Inhalt	Max. Anzahl Container
32	28	4096
64	60	2048
128	124	2048

Die Nutzlänge dieser Container ist um vier Byte geringer als ihre Bruttolänge, da die ersten vier Byte den »Backlink« (die Adresse der Textvariable) enthalten.

Längere Texte werden in Textcontainern gespeichert, die eine Länge von `n * 4KB` `(1<=n<=8)` haben. Sie tragen keine interne Verwaltungsinformation, bei ihnen ist also Brutto = Nettolänge. Die Container werden nach Bedarf nach einem »first fit« Algorithmus vergeben. Ein Text, der Platz in einer dieser Klassen benötigt, erhält den kürzestmöglichen, »ersten« freien Container. Da durch Wachsen von Texten Lücken entstehen und die Heapverwaltung stets bemüht ist, den Gesamtplatzbedarf des Textheap klein zu halten, finden unvorhersagbar Reorganisationen (»Text-Müllabfuhr«, nicht zu verwechseln mit der System-Müllabfuhr) des Heap statt.

Adressen auf dem Textheap (geliefert z.B. durch INT OP TADDR) sind also unsicher, weil nur unbestimmte Zeit gültig!

select container

PROC select container (TEXT VAR text, INT CONST length)

Wirkung: Unabhängig von der aktuellen Länge von 'text' wird ein Container, der 'length' Byte aufnehmen kann, auf dem Textheap reserviert.

shorten

PROC shorten (TEXT VAR text, INT CONST length)

Wirkung: 'text' wird auf die Länge 'length' verkürzt, indem die Längenangabe auf den neuen Wert gesetzt wird. Die Textcontainerzuordnung bleibt also unverändert.

3.6 Speicherung in anderen Datenräumen

In anderen Datenräumen als dem Standarddatenraum ist der Textheap ein zusammenhängender Bereich, in dem Texte in Containern gespeichert werden, deren Größe ein Vielfaches von 16 Byte ist.

```
Benutzerdef. Datenraum

0x0000   Zeiger auf ersten freien Container
0x0004   Zeiger auf ersten Container

           ...

         Statischer
             Teil des Datenraums

         ---------------
         Erster Textcontainer
         x.Textcontainer
           ...
         y.Textcontainer
         erster freier Container
```

Jeder Container hat in seinen ersten beiden Byte eine Kapazitätsangabe `c (1 <= c <= 2048)`. Die Kapazität des Containers errechnet sich durch `(16 * c) - 2`. Falls ein Text über die Größe seines Containers hinaus wächst, wird ein neuer Container an der Heapspitze zugewiesen. Eine »Müllabfuhr« auf dem Textheap findet nicht statt. Es kann also nützlich sein, Daten in einen neuen Datenraum umzukopieren, wenn sehr viel Speicherplatz durch »alte« Texte verbraucht wird.

Die Mindestgröße des Textheap in einem Datenraum kann mit der Prozedur **set min heapsize** festgesetzt werden.

4 Textdateien

4.1 Einführung

Textdateien sind Datenräume, deren innere Struktur als Folge von Sätzen aufgefaßt wird. Beispielsweise können das die einzelnen Zeilen eines Textdokuments, das am Bildschirm angezeigt werden kann, sein. Der Zugriff auf solche Objekte erfolgt durch Assoziierung eines solchen Datenobjekts mit dem Datentyp FILE.

Ein FILE ist ein Datenobjekt des Typs sequentielle Datei. Eine solche Datei besteht aus einer Folge von *records* (Sätzen oder Zeilen). Die Assoziierungsprozedur **sequential file** hat die Aufgabe, eine in einem Programm deklarierte FILE VAR mit einem bereits vorhandenen oder noch einzurichtenden Datenraum, der die geforderte innere Struktur »Reihe von Sätzen« hat, zu koppeln. Bei der Koppelung muß angegeben werden, in welcher Weise auf die Datei zugegriffen werden soll.

Der erste Parameter der Prozedur **sequential file** gibt stets die »Betriebsrichtung« (TRANSPUTDIRECTION) der folgenden Operationen an. Es gibt deren drei:

input Die Datei kann vom Programm nur sequentiell gelesen werden. Durch *input* wird bei der Assoziierung automatisch auf den ersten Satz der Datei positioniert. Ist die zu lesende Datei nicht vorhanden, wird ein Fehler gemeldet.

output Die Datei kann vom Programm nur sequentiell beschrieben werden. Durch *output* wird bei der Assoziierung automatisch hinter den letzten Satz der Datei positioniert (bei einer leeren Datei also auf den ersten Satz). Ist die Datei vor der Assoziierung nicht vorhanden, wird sie automatisch eingerichtet.

modify Die Datei kann vom Programm in beliebiger Weise gelesen und beschrieben werden. Im Gegensatz zu den Betriebsrichtungen *input* und *output*, bei denen ausschließlich ein rein sequentielles Lesen oder Schreiben erlaubt ist, kann bei *modify* beliebig positioniert, gelöscht, eingefügt und neu geschrieben werden. Bei Assoziierung mit *modify* ist bei einer existierenden Datei die Position zunächst undefiniert. Eine Positionierung muß explizit vorgenommen werden. Die Datei wird automatisch eingerichtet, wenn sie vor der Assoziierung nicht vorhanden war.

Der zweite Parameter der Assoziierungsprozedur **sequential file** gibt an, unter welchem Namen die Datei in der Task existiert oder eingerichtet werden soll.

Beispiel:

```
FILE VAR meine datei :: sequential file (output, "xyz");
```

Folgendes Beispiel zeigt ein Programm, welches eine Datei liest und auf dem Ausgabemedium ausgibt:

Beispiel:

```
FILE VAR f :: sequential file (input, "datei1");
TEXT VAR satz;
WHILE NOT eof (f) REP
   getline (f, satz);
   putline (satz);
END REP.
```

Das Festlegen einer Betriebsrichtung impliziert eine Kontrolle der Benutzung der betreffenden Datei und hilft somit Programmierfehler zu vermeiden.

ACHTUNG: *Alle* Prozeduren, die auf FILEs zugreifen, verlangen Objekte vom Typ FILE VAR, da auch die Leseoperationen als ändernd betrachtet werden.

4.1.1 Operationen auf Textdateien

sequential file	Assoziierung
input	Betriebsrichtung
modify	— " —
output	— " —
eof	Dateiende
line no	Aktuelle Zeilennummer
lines	Dateilänge
get	aus input lesen
getline — " —	
col	in modify positionieren
down	— " —
to line	— " —
to eof	— " —
up	— " —
delete record	Satz löschen
insert record	Satz einfügen
read record	Satz lesen
write record	Satz schreiben
line	Zeilenwechsel in output
put	Schreiben in output
putline	Zeile schreiben in output
write	Schreiben in output
lexsort	Lexikalische Sortierung
sort	ASCII Sortierung

4.1.2 Assoziierung

sequential file

FILE PROC sequential file(TRANSPUTDIRECTION CONST mode, DATASPACE CONST ds)

Wirkung: Assoziierung einer sequentiellen Datei mit dem Dataspace 'ds' und der Betriebsrichtung 'mode' (vergl. *modify*, *input* bzw. *output*). Diese Prozedur dient zur Assoziierung eines temporären Datenraums in der Benutzer-Task, auf den nach der Beendigung des Programmlaufs nicht mehr zugegriffen werden kann.

Bemerkung: Da der Name des Datenraums nach Beendigung des Programmlaufs nicht mehr ansprechbar ist, muß der Datenraum explizit vom Programm gelöscht werden, da sonst eine »Datenraumleiche« entsteht!

FILE PROC sequential file(TRANSPUTDIRECTION CONST mode, TEXT CONST name)

Wirkung: Assoziierung einer sequentiellen Datei mit dem Namen 'name' und der Betriebsrichtung 'mode'. Existiert der FILE bereits, dann wird mit *input* auf den Anfang des FILEs, bei *output* hinter den letzten Satz der Datei positioniert. Existiert dagagen der FILE noch nicht und ist die TRANSPUTDIRECTION *output* oder *modify*, wird ein neuer FILE eingerichtet.

Fehler: "name" gibt es nicht Es wurde versucht, einen nicht vorhandenen FILE mit *input* zu assoziieren.

input

PROC input (FILE VAR file)

Wirkung: Ändern der Verarbeitungsart von *modify* oder *output* in *input*. Dabei wird auf den ersten Satz der Datei positioniert.

TRANSPUTDIRECTION CONST input

Wirkung: Assoziierung - in Zusammenhang mit der Prozedur *sequential file* -einer sequentiellen Datei mit der *TRANSPUTDIRECTION input* (nur lesen).

output

PROC output (FILE VAR file)

Wirkung: Ändern der Verarbeitungsart von *input* oder *modify* in *output*. Dabei wird hinter den letzten Satz der Datei positioniert.

TRANSPUTDIRECTION CONST output

Wirkung: In Verbindung mit der Prozedur *sequential file* kann eine Datei assoziiert werden mit der Betriebsrichtung *output* (nur schreiben).

modify

PROC modify (FILE VAR file)

Wirkung: Ändern der Betriebsrichtung von *input* oder *output* in die Betriebsrichtung *modify*.

TRANSPUTDIRECTION CONST modify

Wirkung: Diese Betriebsrichtung erlaubt das Vorwärts- und Rückwärts- Positionieren und das beliebige Einfügen und Löschen von Sätzen.

4.1.3 Informationsprozeduren

eof

BOOL PROC eof (FILE CONST file)

Wirkung: Informationsprozedur auf das Ende eines FILEs. Liefert den Wert TRUE, sofern hinter den letzten Satz eines FILEs positioniert wurde, sonst FALSE.

line no

INT PROC line no (FILE CONST file)

Wirkung: Liefert die aktuelle Zeilennummer.

lines

PROC lines (FILE VAR file)

Wirkung: Liefert die Anzahl der Zeilen der Datei 'file'.

4.1.4 Betriebsrichtung INPUT

In der Betriebsrichtung *input* sind nur Leseoperationen auf der Datei zugelassen. Die Assoziierungsprozedur *sequential file* bewirkt folgendes:

- Falls die Eingabedatei noch nicht existiert, wird eine Fehlermeldung ausgegeben.
- Falls es eine Datei des angegebenen Namens gibt, wird auf das erste Zeichen des ersten Satzes positioniert.

Mit »Wort« ist im folgenden eine nicht leere Zeichenkette aus Zeichen, die ungleich dem Leerzeichen sind, gemeint.

get

PROC get (FILE VAR file, INT VAR number)

Wirkung: Lesen des nächsten Wortes aus der Datei 'file' und Konvertierung des Wortes zu einem Integer-Objekt.

PROC get (FILE VAR file, REAL VAR number)

Wirkung: Lesen des nächsten Wortes aus der Datei 'file' und Konvertierung des Wortes zu einem Real-Objekt.

PROC get (FILE VAR file, TEXT VAR text)

Wirkung: Lesen des nächsten Wortes aus der Datei 'file'.

PROC get (FILE VAR file, TEXT VAR text, TEXT CONST delimiter)

Wirkung: Lesen eines TEXT-Wertes 'text' von der Datei 'file', bis das Zeichen 'delimiter' angetroffen wird. Ein eventueller Zeilenwechsel in der Datei wird dabei übergangen.

PROC get (FILE VAR file, TEXT VAR text, INT CONST maxlength)

Wirkung: Lesen eines TEXT-Wertes 'text' von der Datei 'file' mit 'maxlength' Zeichen. Ein eventueller Zeilenwechsel in der Datei wird dabei *nicht* übergangen.

getline

PROC get line (FILE VAR file, TEXT VAR record)

Wirkung: Lesen der nächsten Zeile aus der sequentiellen Datei 'file'.

Fehler: "Datei zu!" Die Datei 'file' ist gegenwärtig nicht assoziiert.

"Leseversuch nach Dateiende" Es wurde versucht, über die letzte Zeile einer Datei hinaus zu lesen.

"Leseversuch auf output file" Es wurde versucht, von einem mit *output* assoziierten FILE zu lesen.

"Unzulässiger Zugriff auf modify-FILE" Es wurde versucht, aus einer mit *modify* assoziierten Datei zu lesen. Bei **getline** muß die Datei mit *input* assoziiert sein.

4.1.5 Betriebsrichtung MODIFY

In der Betriebsrichtung *modify* sind Lese- und Schreiboperationen auf der Datei zugelassen. Desweiteren ist beliebiges Positionieren in der Datei erlaubt. Neue Sätze können an beliebiger Stelle in die Datei eingefügt werden, die sequentielle Struktur der Datei bleibt erhalten. Die Assoziierungsprozedur *sequential file* bewirkt folgendes:

- Falls die Ausgabedatei noch nicht existiert, wird sie angelegt.

- Falls es bereits eine Datei des angegebenen Namens gibt, ist undefiniert, wo positioniert ist. Die erste Positionierung muß explizit vorgenommen werden!

col

PROC col (FILE VAR file, INT CONST position)

Wirkung: Positionierung auf die Spalte 'position' innerhalb der aktuellen Zeile.

INT PROC col (FILE CONST file)

Wirkung: Liefert die aktuelle Position innerhalb der aktuellen Zeile.

down

PROC down (FILE VAR file)

Wirkung: Positionieren um eine Zeile vorwärts.

PROC down (FILE VAR file, INT CONST lines)

Wirkung: Positionieren um 'lines' Zeilen vorwärts.

to line

PROC to line (FILE VAR file, INT CONST number)

Wirkung: Positionierung auf die Zeile 'number'.

to eof

PROC to eof (FILE VAR file)

Wirkung: Positionierung auf die letzte Zeile der Datei.

up

PROC up (FILE VAR file)

Wirkung: Positionieren um eine Zeile rückwärts.

PROC up (FILE VAR file, INT CONST number)

Wirkung: Positionieren um 'number' Zeilen rückwärts.

delete record

PROC delete record (FILE VAR file)

Wirkung: Der aktuelle Satz der Datei 'file' wird gelöscht. Der folgende Satz wird der aktuelle Satz.

insert record

PROC insert record (FILE VAR file)

Wirkung: Es wird ein leerer Satz in die Datei 'file' vor die aktuelle Position eingefügt. Dieser Satz kann anschließend mit *write record* beschrieben werden (d.h. der neue Satz ist jetzt der aktuelle Satz).

read record

PROC read record (FILE CONST file, TEXT VAR record)

Wirkung: Liest den aktuellen Satz der Datei 'file' in die TEXT-Variable 'record'. Die Position wird dabei nicht verändert.

write record

PROC write record (FILE VAR file, TEXT CONST record)

Wirkung: Schreibt einen Satz in die Datei 'file' an die aktuelle Position. Dieser Satz muß bereits vorhanden sein, d.h. mit **write record** kann keine leere Datei beschrieben werden, sondern es wird der Satz an der aktuellen Position überschrieben. Die Position in der Datei wird nicht verändert.

4.1.6 Betriebsrichtung OUTPUT

In der Betriebsrichtung *output* sind nur Schreiboperationen auf der Datei zugelassen. Die Assoziierungsprozedur *sequential file* bewirkt folgendes:

- Falls die Ausgabedatei noch nicht existiert, wird sie angelegt und auf den ersten Satz positioniert.
- Falls es bereits eine Datei des angegebenen Namens gibt, wird hinter den letzten Satz positioniert. Die Datei kann fortgeschrieben werden.

line

PROC line (FILE VAR file)

Wirkung: Positionierung auf die nächste Zeile der Datei 'file'.

PROC line (FILE VAR file, INT CONST lines)

Wirkung: Positionierung mit 'lines' Zeilen Vorschub in der Datei 'file'.

Fehler: "Datei zu!" Die Datei 'file' ist gegenwärtig nicht assoziiert.

"Schreibversuch auf input-File" Es wurde versucht, auf einen mit 'input' assoziierten FILE zu schreiben.

put

PROC put (FILE VAR file, INT CONST number)

Wirkung: Ausgabe eines INT-Wertes 'number' in die Datei 'file'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

PROC put (FILE VAR file, REAL CONST number)

Wirkung: Ausgabe eines REAL-Wertes 'number' in die Datei 'file'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

PROC put (FILE VAR file, TEXT CONST text)

Wirkung: Ausgabe eines TEXT-Wertes 'text' in die Datei 'file'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

Bemerkung: Bei Textdateien, die mit dem Editor weiterbearbeitet werden sollen, ist also zu beachten: eine Ausgabe mit 'put' setzt ein 'blank' hinter die Ausgabe. Falls dieses Leerzeichen das letzte Zeichen in der Zeile ist, wird eine Absatzmarke in der Zeile gesetzt. Wird mit *write* oder *putline* ausgegeben, steht kein Leerzeichen und somit keine Absatzmarke am Zeilenende.

putline

PROC putline (FILE VAR file, TEXT CONST record)

Wirkung: Ausgabe eines TEXTes 'record' in die Datei 'file'. Danach wird auf die nächste Zeile positioniert. 'file' muß mit 'output' assoziiert sein.

write

PROC write (FILE VAR file, TEXT CONST text)

Wirkung: Schreibt 'text' in die Datei 'file' (analog `put (f, text)`), aber ohne Trennblank.

4.1.7 Sortierung von Textdateien

Textdateien können zeilenweise sortiert werden. Die Sortierung kann nach ASCII oder lexikographisch erfolgen. Die ASCII Sortierung ordnet zeilenweise durch »`text (zeile[i]) <= text (zeile[j])`«. Insbesondere werden Kleinbuchstaben hinter Großbuchstaben geordnet.

Die lexikographischen Sortierung ordnet zeilenweise durch »`text (zeile[i]) LEXGREATEREQUAL text (zeile[j])`«, also nach Art des Telefonbuchs.

lexsort

PROC lexsort (TEXT CONST filename)

Wirkung: Die Datei 'filename' wird zeilenweise in lexikographischer Ordnung sortiert.

PROC lexsort (TEXT CONST filename, INT CONST firstpos)

Wirkung: Die Datei 'filename' wird zeilenweise in lexikographischer Ordnung, beginnend ab Spalte 'firstpos', sortiert.

sort

PROC sort (TEXT CONST filename)

Wirkung: Die Datei 'filename' wird zeilenweise in lexikographischer Ordnung sortiert.

PROC sort (TEXT CONST filename, INT CONST firstpos)

Wirkung: Die Datei 'filename' wird zeilenweise in lexikographischer Ordnung, beginnend ab Spalte 'firstpos', sortiert.

4.2 Editor

Der L3 Editor arbeitet auf Datenräumen des Typs FILE. Er bildet ein Eingabefenster auf einen Datenraum des Typs Textdatei. Der Editor stellt die üblichen Editorfunktionen bereit. Desweiteren verfügt der Editor über einen Kommandointerpreter. Dieser interpretiert Tastenfolgen der Art `ESC key` als Kommandos. Diese Tastenkombinationen und die Handhabung des Editors ist im L3 Benutzerhandbuch in Kapitel 14 beschrieben. Im weiteren sind »fortgeschrittene« Operatoren und Prozeduren erklärt. Sie können insbesonders dazu benutzt werden, Aktionen, die normalerweiser per Hand ausgeführt werden, auf Textdateien per Programm ablaufen zu lassen.

at	Treffer
CA, change all	Mehrmaliges Vertauschen
C, change to	Einmaliges Vertauschen
col	Aktuelle Spaltennummer
D, down, downety	Suche zum Dateiende
edit	Editoraufruf
editfile	Liefert FILE
G	Datei einkopieren
len	Aktuelle Satzlänge
limit	Schreibgrenze einstellen
line no	Aktuelle Zeilennummer
lines	Zeilenzahl
mark	Ausschnitt markieren
mark col	Spalte des Markierungsanfangs
mark line no	Zeile des Markierungsanfangs
margin	Linken Rand einstellen
P	In Datei schreiben
push	In Eingabe einfügen
quit	Abschließen
show	Editor ohne Schreibaktion
T, to line	Auf Zeile 'x'
U, up, uppety	Suche zum Dateianfang
wordwrap	Automatischer Zeilenumbruch
anything noted	Eintragung vorhanden
note	Ins Notizbuch schreiben
note edit	Doppeleditor
notefile	Assoziiert Notizdatei
noteline	Neue Zeile

at

BOOL PROC at (TEXT CONST pattern)

Wirkung: Liefert TRUE, wenn ab der aktuellen Dateiposition 'pattern' steht.

CA

OP CA (TEXT CONST old, new)

Wirkung: Zwischen aktueller Dateiposition und Dateiende wird jedes Auftreten von 'old' durch 'new' ersetzt.

PROC change all (TEXT CONST old, new)

Wirkung: s.o.

C

OP C (TEXT CONST old, new)

Wirkung: Zwischen aktueller Dateiposition und Dateiende wird das erste Auftreten von 'old' durch 'new' ersetzt.

PROC change to (TEXT CONST old, new)

Wirkung: s.o.

col

INT PROC col (FILE CONST file)

Wirkung: Liefert die aktuelle Spaltenposition in 'file'.

INT PROC col

Wirkung: Liefert die aktuelle Spaltenposition in **editfile**.

D

OP D (INT CONST lines)

Wirkung: Positioniert in der gerade edierten Datei (**editfile**) 'lines' Zeilen weiter in Richtung Dateiende , ggf. in die letzte Zeile.

OP D (TEXT CONST pattern)

Wirkung: siehe **down**.

PROC down (INT CONST lines)

Wirkung: siehe **D**.

PROC down (TEXT CONST pattern)
PROC down (TEXT CONST pattern, INT CONST lines)
PROC down (FILE VAR file, TEXT CONST pattern)
PROC down (FILE VAR file, TEXT CONST pattern, INT CONST lines)

Wirkung: Positioniert auf das erste Auftreten von 'pattern' zwischen aktueller Position und Dateiende. Die Suche beginnt ein Zeichen hinter der aktuellen Position. Falls 'pattern' nicht gefunden wird, ist die aktuelle Position das Dateiende.

Die Suche kann durch Angabe von 'lines' auf die nächsten 'lines' Zeilen beschränkt werden.

PROC downety (TEXT CONST pattern)
PROC downety (TEXT CONST pattern, INT CONST lines)
PROC downety (FILE VAR file, TEXT CONST pattern)
PROC downety (FILE VAR file, TEXT CONST pattern, INT CONST lines)

Wirkung: Wie **down**, die Suche beginnt auf der aktuellen Position. Falls mehrfach nach dem gleichen 'pattern' gesucht wird, bleibt die Position unverändert.

edit

PROC edit

PROC edit (TEXT CONST pathname)

PROC edit (THESAURUS CONST thes)

PROC edit (FILE VAR file)

PROC edit (TEXT CONST pathname, INT CONST x, y, xsize, ysize)

PROC edit (FILE VAR file, INT CONST x, y, xsize, ysize)

Wirkung: Editoraufruf.

Parameter: 'pathname': Dateiname, wird falls nötig expandiert. 'file': assoziierte Datei. 'thes': `do (PROC (TEXT CONST) edit, thes)` 'x', 'y': linke, obere Ecke des Editorfenster auf dem Bildschirm. 'xsize', 'ysize': Fenstergröße.

editfile

FILE PROC editfile

Wirkung: Liefert einen FILE Wert um die aktuell edierte Datei zu assoziieren.

Beispiel: `FILE VAR f := editfile; put (len(f));`

G

OP G (TEXT CONST filename)

Wirkung: Vor der aktuellen Position wird die Datei 'filename' einkopiert. Ist in dieser Datei ein Bereich markiert, so wird nur dieser Bereich kopiert.

len

INT PROC len (FILE CONST file)

Wirkung: Liefert die Länge des aktuellen Satzes der Datei.

INT PROC len

Wirkung: Liefert die Länge des aktuellen Satzes von **editfile**.

limit

PROC limit (INT CONST new limit)

Wirkung: Stellt die maximal zulässige Zeilenlänge der aktuelle Datei auf 'new limit' Zeichen ein.

INT PROC limit

Wirkung: Liefert die eingestellte zulässige Zeilenlänge.

line no

INT PROC line no

Wirkung: Liefert die aktuelle Zeilennummer der edierten Datei.

lines

INT PROC lines

Wirkung: Zeilenzahl der aktuell edierten Datei.

mark

PROC mark (FILE VAR file, INT CONST line no, col)

Wirkung: In der Datei 'file' wird ein Ausschnitt markiert. Der Ausschnitt wird beschrieben durch den Start 'line no', 'col' und das Ende »aktuelle Position« (**line no**, **col**). Falls die Startposition der Markierung hinter dem Ende liegt, passiert nichts.

BOOL PROC mark (FILE CONST file)

Wirkung: Liefert TRUE, wenn in 'file' ein Abschnitt markiert ist.

mark col

INT PROC mark col (FILE CONST file)

Wirkung: Liefert die Spaltennummer des Markierungsanfangs in 'file'.

mark line no

INT PROC mark line no (FILE CONST file)

Wirkung: Liefert die Zeilennummer des Markierungsanfangs in 'file'.

margin

PROC margin (INT CONST col)

Wirkung: Stellt den linken Rand für die Darstellung der aktuellen Datei im aktuellen Editorfenster ein. Die Position des Editorfenster auf dem Bildschirm bleibt unverändert.

P

OP P (TEXT CONST filename)

Wirkung: Kopie des markierten Textes aus dem Editorfenster. Die Datei 'filename' wird sofort erzeugt. Falls 'filename' schon existiert, wird erfragt ob gelöscht werden soll. Falls 'filename' nicht gelöscht wird, wird der markierte Text aus der aktuell edierten Datei an 'filename' angehängt, sonst wird gelöscht und direkt neu erzeugt.

push

PROC push (TEXT CONST string)

Wirkung: Fügt die Zeichen 'string' in Editoreingabe ein.

Beispiel: `push (""27"q")`

quit

PROC quit

Wirkung: Das aktuelle Editorfenster wird geschlossen.

show

PROC show (TEXT CONST filename)

PROC show (FILE VAR file)

PROC show (THESAURUS CONST thes)

PROC show

Wirkung: Öffnet ein Editorfenster auf der (den) Datei(en). In diesem Fenster sind keine Schreibaktionen zulässig. Es ist aber möglich Text zu markieren und den markierten Text zu duplizieren (*ESC d*).

T

OP T (INT CONST line no)

PROC to line (INT CONST line no)

Wirkung: Positioniert auf Zeile 'line no' der aktuellen Datei ohne die Spaltenposition zu ändern. Falls die Datei weniger als 'line no' Zeilen hat, ist die aktuelle Position hinter dem letzten Zeichen der Datei.

U up uppety

Bemerkung: Wirkung und Parametrisierung sind analog zu **D**, **down**, **downety**. Die Suchrichtung ist jedoch von der aktuellen Position in Richtung Dateianfang.

wordwrap

PROC wordwrap (BOOL CONST bool)

Wirkung: Bei 'TRUE' wird der automatische Zeilenumbruch des Editors eingeschaltet, bei FALSE ausgeschaltet. Wenn das Kommando während des Edirens gegeben wird, wird die Einstellung als Dateiattribut gespeichert, wird die Einstellung im Monitor vorgenommen, so gilt sie als Voreinstellung für neue Dateien.

4.2.1 Notizbuch

Das Notizbuch ist ein unbenannte Datei, auf die ein zweites Editorfenster gesetzt werden kann, um z.B. Zwischenergebnisse oder besondere Meldungen anzuzeigen. Meldungen des ELAN Compilers etwa werden ins Notizbuch geschrieben.

anything noted

BOOL PROC anything noted

Wirkung: Liefert TRUE nachdem die erste Schreiboperation im Notizbuch vorgenommen wurde und FALSE nachdem das Notizbuch mit **note edit** angezeigt wurde.

note

PROC note (TEXT CONST text)

Wirkung: Schreibt 'text' ins Notizbuch.

PROC note (INT CONST number)

Wirkung: Schreibt `text (number) + " "` ins Notizbuch.

note edit

PROC note edit

Wirkung: Zeigt das Notizbuch an.

notefile

FILE PROC notefile

Wirkung: Liefert einen FILE Wert, um das Notizbuch zu assoziieren.

noteline

PROC noteline

Wirkung: Zeilenvorschub im Notizbuch.

4.3 Die Tastenverwaltung

In den Eingabeprozeduren **edit** und **editget** ist es möglich, Tastenkombinationen aus »ESC« und einer weiteren Taste zu definieren, die dann als Kommando oder Macro (Lernsequenz) interpretiert werden. Die Tasten werden also mit beliebigen Zeichenfolgen »belegt«. Auf diese Weise ist es möglich, häufig auftretende Eingaben auf Tasten zu legen.

Einige Tastenkombinationen sind in L3 vorbelegt, sie können jedoch tasklokal geändert werden. Falls in einer Task die Tastenbelegungen geändert waren, kann durch **std keys** die Normaleinstellung jederzeit wiederhergestellt werden.

Der Unterschied zwischen Macros und Kommandos besteht darin, daß ein Text, der als Macro auf eine Taste gelegt wurde, ausgegeben wird und dann weiter ediert werden kann. Natürlich kann ein Macro nicht nur druckbare Zeichen sondern auch Cursorpositionierungen etc. enthalten. Ein Kommando hingegen wird direkt an den ELAN Compiler übergeben.

Ein Taste kann entweder mit einem Macro oder einem Kommando belegt werden.

load key command	Kommando auf Taste legen
load key macro	Macro auf Taste legen
key command	Liefert Kommandosequenz einer Taste
key macro	Liefert Macro auf einer Taste
key contains command	Taste belegt
std keys	Normaleinstellung

load key command

PROC load key command (TEXT CONST key, string)

Wirkung: Die Taste 'key' wird mit der Zeichenfolge 'string' belegt. Falls 'string' ein korrektes ELAN Programm darstellt, wird dieses Programm direkt übersetzt und ausgeführt, sobald `ESC` key hintereinander gedrückt werden. Falls das Programm nicht korrekt ist, erfolgt keine Reaktion. Es wird keine Fehlermeldung ausgegeben, um den Bildschirm nicht zu zerstören.

Beispiel: `load key command ("l", "list")`

load key macro

PROC load key macro (TEXT CONST key, string)

Wirkung: Die Taste 'key' wird mit der Zeichenfolge 'string' belegt. Sobald `ESC key` hintereinander gedrückt werden, wird die Zeichenfolge 'string' an der aktuellen Cursorposition ausgegeben.

Beispiel:

```
(* Editoraufruf auf Dateinamen aus 'list'    *)
(* Alle Dateien werden angezeigt, der Cursor *)
(* steht auf der Datei, die mit dem Makro    *)
(* ediert werden soll.                       *)

load key macro ("E", edit from list).
edit from list :
      esc + "p"          (* Zeile in Zwischenspeicher *)
    + esc + "q"          (* 'list' verlassen *)
    + esc + "g"          (* Zeile aus Zwischenspeicher holen *)
    + ")" + hop + left
    + "edit (" +  (13 * entf)+ return. (* Kommandozeile aufbauen *)
```

```
esc    : ""27"". (* Codes der benutzten Tasten *)
hop    : ""1"".
left   : ""8"".
entf   : ""12"".
return : ""13"".
```

key command

TEXT PROC key command (TEXT CONST key)

Wirkung: Liefert den Quelltext des auf die Taste 'key' gelegten ELAN Programms. Falls kein Kommando auf der Taste liegt, wird **niltext** geliefert.

key macro

TEXT PROC key macro (TEXT CONST key)

Wirkung: Liefert die Tastenfolge, die auf die Taste 'key' gelegt wurde. Falls keine Folge auf der Taste liegt, wird **niltext** geliefert.

key contains command

BOOL PROC key contains command (TEXT CONST key)

Wirkung: Liefert TRUE, wenn auf die Taste 'key' ein Kommando gelegt ist.

std keys

PROC std keys

Wirkung: Setzt die Belegung der systemseitig mit Macros voreingestellten Tasten auf die Standardbelegung zurück.

4.4 Scanner

Der Scanner kann benutzt werden, um festzustellen, welche Art von Symbolen in einem TEXT enthalten sind. Die Analyse des Textes folgt den Regeln von ELAN: die Repräsentation der Symbole muß der ELAN-Syntax entsprechen. Folgende **Symbole** werden vom Scanner erkannt:

`tags`: Namen. Zeichenketten, die ausschließlich aus Kleinbuchstaben oder Ziffern bestehen. Leerzeichen innerhalb von Namen sind zulässig, werden jedoch überlesen.

`bolds`: Schlüsselworte. Zeichenketten, die ausschließlich aus Großbuchstaben bestehen. Leerzeichen innerhalb von Schlüsselworten sind nicht zulässig.

`number`: INT oder REAL Zahlen. Zeichenketten, die aus den Ziffern 0 bis 9 bestehen und ggf. ein Vorzeichen, einen Dezimalpunkt oder ein `e` enthalten (siehe REALs). Leerzeichen innerhalb von Zahlen sind zulässig, werden jedoch überlesen.

`operator:` Alle Einzelzeichen, die nicht als Delimiter klassifiziert sind, sowie `::`, `:=`, `**`, `<=`, `>=`.

`delimiter:` Begrenzer. Gültige Delimiter sind `"`, `(`, `)`, `[`, `]`, `.`, `,` und `;`. und das Ende des Scan-Textes.

Der Scanner überliest Kommentare und Leerzeichen zwischen den Symbolen. Der (erste) zu verarbeitende Text muß mit der Prozedur

scan

an den Scanner übergeben werden. Mit der Prozedur

next symbol

wird das jeweils nächste Symbol des TEXTes geholt. Am Ende wird `end of scan` und als Symbol `niltext` geliefert. Falls innerhalb eines TEXT-Denoters oder eines Kommentars `end of scan` auftritt, wird `within text` bzw. `within comment` gemeldet. Der Scan-Prozeß kann dann mit dem nächsten zu scannenden TEXT (der nächsten Zeile) fortgesetzt werden. Dafür wird nicht die Prozedur **scan**, sondern

continue scan

verwendet. Sie setzt im letzten Scan-Zustand (z.B. Kommentar oder TEXT-Denoter) wieder auf, so daß auch Folgen von TEXTen (Zeilen) - wie z.B. Dateien - leicht gescannt werden können.

Mit den Prozeduren

scan `(* meldet eine Datei zum scannen an *)`
next symbol `(* holt die Symbole *)`

kann man auch eine Datei nach ELAN-Symbolen untersuchen.

4.4.1 Scanner-Prozeduren

continue scan

PROC continue scan (TEXT CONST string)

Wirkung: Das Scanning soll mit 'string' fortgesetzt werden. Falls der Scan- Vorgang beim vorigen 'string' innerhalb eines TEXT-Denoters oder eines Kommentars abgebrochen wurde, wird er jetzt entsprechend mit dem nächsten **next symbol** fortgesetzt. Der erste Teil-Scan einer Folge muß aber stets mit **scan** eingeleitet werden!

next symbol

PROC next symbol (TEXT VAR symbol, INT VAR type)

Wirkung: Holt das nächste Symbol. In 'symbol' steht der TEXT des Symbols, so z.B. die Ziffern eines INT-Denoters. Bei TEXT-Denotern werden die führenden und abschließenden Anführungsstriche

die führenden und abschließenden Anführungsstriche abgeschnitten. Leerzeichen oder Kommentare spielen in **tags** und **numbers** keine Rolle. Zwischen Symbolen spielen Leerzeichen und Kommentare keine Rolle. Der Type des Symbols wird in der Variablen 'type' - wie unter **Parameter** beschrieben - geliefert.

Bemerkung: Wird Scan-Ende innerhalb eines Kommentars gefunden, so wird `niltext` und `within comment` geliefert. Wird Scan-Ende innerhalb eines TEXT-Denoters gefunden, so wird der schon analysierte Teil des Denoters und `within text` geliefert.

Parameter: In 'type' steht eine Kennzeichung für den Typ des Symbols.

```
tag            = 1 ,
bold           = 2 ,
number         = 3 ,
text           = 4 ,
operator       = 5 ,
delimiter      = 6 ,
end of file    = 7 ,
within comment = 8 ,
within text    = 9 .
```

PROC next symbol (TEXT VAR symbol)

Wirkung: Wie oben. Es wird aber nur der Text des Symbols (ohne Typ) geliefert.

PROC next symbol (FILE VAR file, TEXT VAR symbol, INT VAR type)

Wirkung: Wie oben. Jedoch wird auf einem FILE gearbeitet.

PROC next symbol (FILE VAR file, TEXT VAR symbol)

Wirkung: Wie oben. Jedoch wird auf einem FILE gearbeitet.

scan

PROC scan (TEXT CONST string)

Wirkung: Meldet einen 'string' für den Scanner zur Verarbeitung an. Die Prozedur **scan** muß vor dem ersten Aufruf von **next symbol** gegeben werden. Im Gegensatz zu **continue scan** normiert **scan** den inneren Zustand des Scanners, d.h. vorherige Scan-Vorgänge haben keinen Einfluß mehr auf das Scanning.

PROC scan (FILE VAR file)

Wirkung: Wie oben. Jedoch wird auf einen FILE gearbeitet. Die zu untersuchende Zeile ist die nächste Zeile im FILE 'f' (**scan** macht zuerst ein **getline**).

Beispiel:

```
FILE VAR file := sequential file (input, filename);

TEXT VAR symbol := "",
         bedeutung := "";
INT VAR type;

scan (file);
WHILE NOT eof (file)
REP
   next symbol (file, symbol, type);
   analyse ;
   IF type = 8 COR type = 9
      THEN continue scan (symbol) FI;
PER

.analyse:
   put(symbol) ; line ;
   put("Typ: ");put(type);
SELECT type OF
   CASE 1 :  bedeutung := "tag"
   CASE 2 :  bedeutung := "bold"
   CASE 3 :  bedeutung := "number"
   CASE 4 :  bedeutung := "text"
   CASE 5 :  bedeutung := "operator"
   CASE 6 :  bedeutung := "delimiter"
   CASE 7 :  bedeutung := "End of File"
   CASE 8 :  bedeutung := "within comment"
   CASE 9 :  bedeutung := "within text"

OTHERWISE
errorstop("Unmöglicher Wert vom Scanner");
END SELECT;

put (bedeutung);
line;
pause(10).
```

5 Datenräume

5.1 Einführung

Datenräume bilden die oberste und allgemeinste Klasse für Objekte, die gespeichert werden. Der Typ DATASPACE bezeichnet Objekte vom Typ Datenraum, ein elementarer Datentyp des L3-Systemkerns. Datenräume können Daten beliebigen Typs aufnehmen und gewähren direkten Zugriff auf ihren Inhalt. Ein Datenraum ist als Container für Daten aufzufassen, es sind zunächst keine Annahmen über den Inhalt (Texte, ausführbarer Code oder vielleicht eine Datenbank?) nötig. Ein Datenraum ist einfach ein Stück linearer Speicher, das bis zu einem Gigabyte groß werden kann.

Datenräume sind lokale Objekte in den Tasks eines L3 Systems, oder einfacher gesagt, jeder Datenraum hat genau eine Besitzertask. Eine Task kann bis zu 16380 Datenräume besitzen. Da der dafür theoretisch erforderliche Adreßraum 16 Terabyte beträgt, der Prozessor jedoch *nur* 4 Gigabyte pro Task adressieren kann, muß noch eine Abbildung der aktuell angesprochenen Datenräume in den virtuellen Adreßraum der Task erfolgen. Diese Abbildung heißt **Mapping**. Das Mapping erfolgt in Segmenten zu 16 MB. Bis zu 234 dieser 16 MB Segmente können *gleichzeitig* »gemappt« sein, die restlichen 22 Segmente werden intern beansprucht. Da Datenobjekte meistens kleiner als 16 MB sind, bedeutet das, daß gleichzeitig 234 »Dateien geöffnet« sein können. Falls Datenräume größer als 16 MB sind, verringert sich die Anzahl der möglichen Mappings entsprechend, da ein solch großer Datenraum dann zwei oder mehr nebeneinanderliegende Einträge in der Mappingliste beansprucht. Falls sehr große Datenräume gemappt werden sollen, sollten sie sofort nach Erzeugung der Task gemappt werden, da andernfalls die Gefahr besteht, daß sich in der Mappingliste keine hinreichend große »Lücke« mehr findet. Mapping- Operationen sind implizit in den üblichen Datenraumoperationen und darauf aufbauenden enthalten. Beispielsweise ist nach einem **edit** auf eine neue Datei der Datenraum anschließend gemappt. Explizites Mapping per **open** und **close** ist nur dann erforderlich, wenn sehr viele (große) Datenräume gleichzeitig direkt adressiert werden müssen.

Die Datenräume einer Task sind in der Reihenfolge ihrer Erschaffung, bzw. ihres Empfangs, durchnumeriert. Dieser Datenraumindex kann zum Durchlaufen aller Datenräume einer Task benutzt werden. Datenraumindices, die nach Löschen eines Datenraums frei sind, können wiederbenutzt werden. Der Index kann also nicht zur Identifikation benutzt werden.

Bei der Beschreibung von Operationen auf Datenräumen sind drei Zugriffsebenen zu unterscheiden. Zunächst gibt es Operationen, die für den Datentyp DATASPACE Initialisierung, Zuweisungsprozeduren und dergleichen anbieten. Auf dieser Ebene ist ein Datenraum ein atomares Objekt. Er wird in seiner Ganzheit erzeugt und gehandhabt. Diese Ebene wird z.B. von der Intertask Kommunikation benutzt. Wie in Kapitel 8 beschrieben, ist es möglich einen oder mehrere Datenräume in einem atomaren Botschaftstransfer zwischen Sender- und Empfängertask zu übertragen.

Die nächste Zugriffsebene betrifft die Organisation eines Datenraums. Er ist ein Objekt, das aus einer Anzahl von »Pages« im virtuellen Speicher besteht. Diese technische Struktur eines Datenraums kann Angaben über benötigten Speicherplatz liefern und wird insbesondere für blockorientierte Schreib- und Leseoperationen, z.B. auf Diskette, benutzt.

Die dritte Zugriffsebene betrifft den korrekten Zugriff auf den Datenrauminhalt. Diese innere Struktur wird einem Datenraum von der konkreten Anwendung »aufgeprägt«.

Durch das Schlüsselwort **BOUND** bei der Deklaration einer Variablen wird in einem ELAN-Programm erreicht, daß diese Variable nicht im Standarddatenraum, sondern im jeweils angekoppelten Datenraum abgelegt wird.

Da die innere Struktur eines Datenraums anwendungsabhängig ist, können an dieser Stelle keine Aussagen über Möglichkeiten des Zugriffs auf diese Struktur gemacht werden. Ein Datenraum kann eine Typnummer erhalten. Durch Test dieser Typnummer kann dann festgestellt werden, ob die passenden Zugriffswerkzeuge zur Verfügung stehen.

Eine gewisse Sonderbehandlung erfahren TEXTE in Datenräumen. Texte werden als dynamische Strukturen in einem Textheap gespeichert. Die Mindestgröße dieses Textheaps kann mit der Prozedur **set min heap size** vorgegeben werden. Siehe dazu auch Kap 2.3: *Aufbau des Textheap in anderen Datenräumen*.

5.1.1 Koppelung von Datenräumen

Eine ständig innerhalb des L3 Systems verwendete Technik ist die Koppelung von Datenräumen durch *faules Kopieren*. Die Überlegung dabei ist, daß die virtuelle Speicherverwaltung ermöglicht, Kopieroperationen zunächst in Bezug auf das eigentliche Datenobjekt (den Datenraum) zu unterdrücken.

Dazu zunächst ein Blick auf die in dieser Erörterung aus Gründen der Verständlichkeit vereinfacht dargestellte, technische Organisation eines Datenraums. Sie läßt sich vorstellen als ein Zeiger auf eine Seitenblocktabelle, die Einträge dieser Tabelle verweisen auf die Blöcke auf dem Hintergrundspeicher (Festplatte). Durch eine von der MMU (Memory Management Unit) des 80386 unterstützte Adreßumsetzung erhält man so ein wahlfrei adressierbares Datenobjekt.

```
                          Seite p1    ----->   Block a (4096 Byte)
DATASPACEA Ptr    --->    Seite p2    ----->   Block b    "
                          Seite p3    ----->   Block c    "
                          ...
```

Um eine Kopie dieses Datenraums zu erhalten, reicht es nun zunächst aus, einen zweiten Verweis auf die Seitenblocktabelle zu erzeugen und zu vermerken, daß der Datenraum gekoppelt ist.

```
                          Seite p1    ----->   Block a (4096 Byte)
DATASPACEA Ptr   ---->    Seite p2    ----->   Block b    "
DATASPACEB Ptr   ---->    Seite p3    ----->   Block c    "
                          ...
```

Da die (internen) Datenraumnamen innerhalb des Gesamtsystems eindeutig sind, wird diese Kopiertechnik sowohl tasklokal, als auch systemweit benutzt.

Erst wenn ein Schreibzugriff stattfindet, müssen die betroffenen Datenraumseiten tatsächlich kopiert werden.

```
                          Seite p1    ----> |--> Block a (4096 Byte)
DATASPACEA Ptr    --->    Seite p2    ----> ||-> Block b    "
```

```
                              Seite p3    ----> ||    Block c    "
                                                ||
                              Seite p1    ------||
   DSCOPY Ptr       --->      Seite p2    -------|
                              Seite p3    ---------> Block d    "
                                        (neu)
```

Dieses Verfahren erlaubt das Kopieren großer Datenmengen, ohne daß nennenswerte Datenmengen bewegt werden. Insbesondere im L3 Systemmantel ist diese Technik gut erkennbar. Das Tasksystem besteht von Beginn an aus mehreren Tasks, die jede logisch über 3MB Speicherplatz einnehmen. Da die Standarddatenräume der Tasks, die diesen Platz beanspruchen, sich jedoch nicht oder nur geringfügig unterscheiden, beträgt der Gesamtplatzbedarf eines frischen Systems nur etwa das Doppelte der Standarddatenraumgröße, plus ca. 1,5 MB für den L3- Systemkern und Treiberprozesse.

5.1.2 Datenraumoperationen

:=	Zuweisung
nilspace	Initialisierungsobjekt
copy	Kopieren
close	Datenraum abkoppeln
dataspaces	Anzahl der Datenräume
delete	Datenraum löschen
exists	Vollständige Kopplung
move	Löschend kopieren
next	liefert nächsten Datenraum
index	Liefert Index
pages	Anzahl der belegten Seiten
page size	Liefert Seitengröße
set min heap size	Textheapgröße
storage	Speicherplatz
type	Typ aufprägen
	Aufgeprägten Typ liefern
BOUNDPTR	Mappingvariable
address	Mappingadresse
close	Mapping aufheben
dataspace	Liefert Datenraum
open	Mapping

:=

OP := (DATASPACE VAR dest, DATASPACE CONST source)

Wirkung: Der Datenraum 'dest' wird als Kopie von 'source' angelegt.

Bemerkung: Es handelt sich um eine logische Kopie, eine (teilweise) physische Kopie wird erst nach einem Schreibzugriff auf einen der beiden Datenräume nötig.

nilspace

DATASPACE PROC nilspace

Wirkung: Liefert den leeren Datenraum **nilspace**, der ausschließlich als Quelle zum Kopieren verwendet werden darf.

copy

PROC copy (DATASPACE CONST source, DATASPACE VAR dest)

Wirkung: Der Datenraum 'dest' wird als Kopie von 'source' angelegt.

PROC copy (DATASPACE CONST source, dest, INT CONST from page, to page, pages)

Wirkung: Aus dem Datenraum 'source' werden 'pages' Seiten ab Seite 'from page' in den Datenraum 'dest' beginnend ab Seite 'to page' kopiert.

Bemerkung: Es handelt sich hier um logische Kopien, eine (teilweise) physische Kopie wird erst nach einem Schreibzugriff auf einen der beiden Datenräume nötig.

PROC copy (DATASPACE CONST ds, INT CONST from page, to page, pages)

Wirkung: Innerhalb des Datenraums 'ds' werden 'pages' Seiten ab Seite 'from page' beginnend ab Seite 'to page' kopiert.

Bemerkung: Es handelt sich (natürlich) um »faule Kopien«. Die Seiten werden nicht entkoppelt.

close

PROC close (DATASPACE CONST ds)

Wirkung: Die Abbildung des Datenraums 'ds' in den Adreßraum der Task wird explizit beendet.

dataspace

DATASPACE PROC dataspace (TEXT CONST name)

Wirkung: Liefert den Datenraum, der mit dem Namen 'name' assoziiert ist.

Beispiel:

```
create ("matrix4D", ds) ;

TYPE DA   = STRUCT (ROW 25 INT a),
     DB   = STRUCT (ROW 25 DA b),
     DC   = STRUCT (ROW 25 DB c),
     DD   = ROW 25 DC;

BOUND DD VAR sp := dataspace ("matrix4D") ;

INT VAR i,j,k,l;

FOR i FROM 1 UPTO 05
REP
   FOR j FROM 1 UPTO 05
   REP
      FOR k FROM 1 UPTO 05
      REP
         FOR l FROM 1 UPTO 05
         REP
         sp[i].c[j].b[k].a[l] := i * j * k * l;
         PER;
      PER;
  PER;
PER;
```

dataspaces

INT PROC dataspaces

Wirkung: Liefert die Anzahl aller (benannten und unbenannten) Datenräume der Task.

INT PROC dataspaces(TASK CONST task)

Wirkung: Liefert die Anzahl aller (benannten und unbenannten) Datenräume der Task 'task'.

Bemerkung: Die größtmögliche Anzahl der Datenräume einer Task beträgt (zur Zeit) 16383.

delete

PROC delete (DATASPACE VAR ds)

Wirkung: Der Datenraum 'ds' wird gelöscht.

PROC delete (DATASPACE CONST ds, INT CONST from page, pages)

Wirkung: Im Datemraum 'ds' werden beginnend ab Seite 'from page' 'pages' Seiten gelöscht.

Bemerkung: Durch diese Prozedur steht nicht unmittelbar mehr freier Speicherplatz auf der Festplatte zur Verfügung. Die physische Räumung von Speicherplatz erfolgt erst durch die »Müllabfuhr«.

exists

BOOL PROC exists (DATASPACE CONST ds)

Wirkung: Liefert 'TRUE', falls der Datenraum 'ds' im Adressraum der Task existiert, sonst 'FALSE'.

move

PROC move (DATASPACE VAR source, dest)

Wirkung: Kopiert den Datenraum 'source' nach 'dest' und löscht 'source'.

next

DATASPACE PROC next (DATASPACE CONST ds)

Wirkung: Liefert den auf 'ds' folgenden Datenraum der Task. Die vorhandenen Datenräume können mit dieser Prozedur zyklisch abgerufen werden. Es gilt: **next (nilspace)** liefert den Standarddatenraum, **next ('last ds')** liefert **nilspace**.

INT PROC next (DATASPACE CONST ds, INT CONST pageno)

Wirkung: Liefert die nächste belegte Seite des Datenraums 'ds'. Falls 'pageno' = -1 ist, wird die erste belegte Seite geliefert, gibt es keine belegte Seite mehr, so wird -1 geliefert.

Bemerkung: In den Seitennummern können Lücken sein.

index

INT PROC index (DATASPACE CONST ds)

Wirkung: Liefert den Index des Datenraums 'ds'. Es gilt: `index (next (nilspace)) = 1` Der Standarddatenraum einer Task hat den Index 1.

pages

INT PROC pages (DATASPACE CONST ds)

Wirkung: Liefert die Anzahl der durch 'ds' belegten Seiten (je 4096 Byte).

page size

INT PROC page size

Wirkung: Liefert die Größe einer Seite eines Datenraums in Byte.

set min heap size

PROC set min heapsize (DATASPACE CONST ds, INT CONST new limit)

Wirkung: Die Größe des Textheapbereiches in dem Datenraum wird verändert. Standardmäßig werden vom Laufzeitsystem mindestens 10 MB eines Datenraums als Textheap reserviert, unabhängig davon, ob die zu speichernde Struktur TEXTe enthält, oder nicht. Der tatsächlich zur Verfügung stehende Platz errechnet sich aus der Differenz der Größe des statischen Teils eines Datenobjekts zum nächstgrößen Vielfachen von 16 MB. Falls der statische Teil so viel Platz einnimmt, daß weniger als 10 MB für den Textheap zur Verfügung stehen, wird ein weiteres Datenraumsegment allokiert. Die Angabe 'new limit' erfolgt in Bytes, es findet keine Überprüfung dieses Werts auf eine sinnvolle Größe statt. Generell braucht man natürlich nur dann Überlegungen zu diesem Thema anzustellen, wenn man mit großen Datenobjekten agiert!

Beispiel:

1. Der Datenraum soll `ROW 3 000 000 INT` aufnehmen. Da kein Text benötigt wird, kann erreicht werden, daß der Datenraum auf ein Segment von 16 MB beschränkt bleibt, indem der das Datenobjekt an eine Datenraumvariable gekoppelt wird, für die kein Textheap vereinbart wurde.

```
create ("hugo", ds);
set min heap size (dataspace ("hugo"), 0);
TYPE MI = ROW 3 000 000 INT;
BOUND MI VAR mio := dataspace("hugo") ;
INT VAR i;
FOR i FROM 1 UPTO 3 000 000
  REP
     mio [i] := i
  PER ;
```

2. Der Datenraum soll `ROW 40 000 TEXT` aufnehmen. Es ist abzusehen, daß die Texte durchaus bis 1000 Byte lang werden. Die Heapgröße muß in diesem Fall auf (sicherheitshalber) 45 000 000 hochgesetzt werden:

```
create ("texte satt", ds);
set min heap size (dataspace ("texte satt"), 45000000);
TYPE TH = ROW 40 000 000 TEXT;
BOUND TH VAR thd := dataspace("texte satt") ; ...
```

3. Der Datenraum enthält eine `ROW 1000 INT`. Offensichtlich würde genügend Platz für den Textheap innerhalb 16 MB zur Verfügung stehen. Man braucht sich keine Gedanken um eine Einschränkung zu machen.

storage

INT PROC storage (DATASPACE CONST ds)

Wirkung: Liefert den von 'ds' belegten Speicherplatz. Die Angabe erfolgt in Bytes. Das Ergebnis ist jedoch immer ohne Rest durch 4096 teilbar, da es aus der Anzahl der Seiten des Datenraums errechnet wird.

type

PROC type (DATASPACE CONST ds, INT CONST typ)

Wirkung: Der Datenraum 'ds' erhält die Typnummer 'typ'. Die Typnummer muß eine positive Zahl sein.

INT PROC type (DATASPACE CONST ds)

Wirkung: Liefert die Typnummer des Datenraums 'ds'.

Bemerkung: Falls dem Datenraum 'ds' eine Typnummer zugewiesen wurde, wird diese geliefert. Wurde 'ds' keine Typnummer zugewiesen gilt folgendes:

type ('ds') < 0 falls 'ds' nie an ein BOUND Objekt angekoppelt war,
type ('ds') = 0 falls 'ds' schon an ein BOUND Objekt angekoppelt war.

5.1.3 Datenräume in ELAN-Programmen

Durch das Schlüsselwort BOUND bei der Deklaration einer Variablen teilt man dem ELAN-Compiler mit, daß die Werte dieser Variablen in einem Datenraum gespeichert werden. Das Ablegen von Datenobjekten in Datenräumen kann aus verschiedenen Gründen notwendig bzw. sinnvoll sein:

- die Größe des Datenobjektes überschreitet dem vom ELAN-Compiler für Variablen vorgesehenen Platz,
- das Datenobjekt soll in andere Tasks des L3 Systems kopiert werden (z.B. zur Sicherung der Daten)

Beispiel für die Verwendung eines **unbenannten Datenraums**:

```
BOUND ROW 10000 REAL VAR liste;
(* 'liste' soll in einem Datenraum abgelegt werden          *)
DATASPACE VAR ds :: nilspace;
(* erzeugen des Datenraums 'ds' als Kopie des nilspace      *)
liste := ds;
(* 'liste' wird an den Datenraum 'ds' angekoppelt           *)
bearbeite liste;
(* Das Datenobjekt 'liste' wird bearbeitet                  *)
copy (ds, public, "Liste");
(* Der Datenraum 'ds' wird unter dem Namen "Liste" in der
   Task "PUBLIC" gesichert                                  *)
delete (ds);
(* nicht vergessen, sonst wird der belegte Platz nicht
   wieder freigegeben, bevor die Task gelöscht wird         *)
. bearbeite liste:
  liste [314] := pi;
```

Beispiel für die Verwendung eines benannten, von der Dateiverwaltung der Task verwalteten, Datenraums:

```
BOUND ROW 10000 STRUCT (REAL wert, INT position) VAR liste;
(* 'liste' soll in einem Datenraum abgelegt werden          *)
create ("liste2") ;
liste := dataspace ("liste 2");
(* 'liste' wird an einen neuen Datenraum angekoppelt. Dieser
   Datenraum wird unter dem Namen "liste 2" von der Dateiver-
   waltung der Task verwaltet                               *)
bearbeite liste;
(* Das Datenobjekt 'liste' wird irgendwie bearbeitet        *)
programmende;
```

5.2 Mapping

In den bislang erörterten Prozeduren wurde zumindestens implizit vorausgesetzt, daß ein Datenraum ein Datencontainer ist. Er wird als Speicherobjekt für eine Datenstruktur aufgefaßt. Der Zugriff auf den Inhalt des Datenraums setzt Kenntnis der inneren Struktur voraus. Es kann jedoch auch notwendig sein, ohne Beachtung der inneren Struktur auf den Inhalt eines Datenraums zuzugreifen. Beispielsweise wird diese Möglichkeit benutzt, um in der Task *hardware configurator* in eine (Binär-)datei, die zum Standarddatenraum eines Treibers wird, die vom Benutzer eingestellte Interruptnummer, Portadresse und dergleichen einzutragen. Der in dieser Datei enthaltene ausführbare Code kann schwerlich als Datenstruktur betrachtet werden, nichtdestotrotz müssen Schreib/Leseoperationen in solchen Datenräumen möglich sein.

Für derartige Zwecke gibt es den Datentyp **BOUNDPTR**. Variablen dieses Typs liefern für einen gemappten Datenraum eine Basisadresse, die, durch ein Offset ergänzt, Zugriff auf den Inhalt des Datenraums ermöglicht.

BOUNDPTR

TYPE BOUNDPTR

Wirkung: Typ für Variablen die Mapping eines Datenraums erlauben, der nicht über BOUND Variable angekoppelt werden soll oder kann.

address

REF INT PROC address (BOUNDPTR CONST boundptr)

Wirkung: Liefert die Mappingadresse.

close

PROC close (BOUNDPTR VAR bound ptr)

Wirkung: Das Mapping des mit 'bound ptr' assoziierten Datenraum wird beendet.

dataspace

DATASPACE PROC dataspace (BOUNDPTR CONST boundptr)

Wirkung: Liefert den durch 'boundptr' adressierten Datenraum.

open

PROC open (DATASPACE CONST space, INT CONST map size BOUNDPTR VAR bound ptr)

Wirkung: Mappt den Datenraum 'space' in der (in Byte) angegebenen Größe. Die Variable 'bound ptr' muß solange gültig sein, wie das Mapping Bestand haben soll.

Beispiel:

```
LET mb16 = 0x 0100 0000 ,
    config area = 0x80  ;
BOUNDPTR VAR std space ;
....
open (dataspace("stdds"), mb16, std space);
...
drivers ip := VAL (REF INT [address (std space)
                           + config area]) ;
```

Soll sagen: die Startadresse für die Aktivierung des Treibers, also der Anfangswert seines Befehlszählers (`drivers ip`) wird aus dem Konfigurationsbereich des Programms gelesen. Dieses Programmsegment ist aus dem "install" Programm zu einem Treiber abgeleitet. Es zeigt ganz nebenbei, daß auch die Benutzung von BOUNDPTRn Kenntnisse über die innere Struktur des Datenraums erfordert, da ohne nähere Kenntnisse über die `config area` auch ein BOUNDPTR nur zum Herumstochern taugte.

5.3 Der Datentyp TASK

5.3.1 Einführung

Prozesse und damit insbesondere auch Tasks sind die einzig globalen Objekte in einem L3 System. Es muß für jeden Prozeß eine eindeutige Benennung im System geben. Insbesondere müssen Prozesse auch über ihre Lebensdauer hinaus eindeutig identifizierbar sein. Wäre diese Sicherheit nicht gegeben, bestünde z.B. die Gefahr, daß nach Löschen einer Task durch ihren Besitzer ein anderer Benutzer im System eine gleichnamige Task einrichtet und zum Empfänger von Daten wird, die nicht für ihn bestimmt sind.

Die Sicherheit der Benennung wird bei L3 durch vom Systemkern vergebene *interne Taskbezeichner* erreicht. Ein solcher Bezeichner besteht aus einem Taskindex und einer Generationsnummer. Der Index ist der zeitinvariante Teil des internen Taskbezeichners. Durch Kombination des Index mit einer Generationsnummer ist sichergestellt, daß Tasks auch über die Zeit eindeutig identifizierbar sind. Bei erneuter Vergabe des Index wird die Generationsnummer hochgezählt, so daß auch bei »alten« Systemen die Eindeutigkeit der Taskbezeichner sicher ist.

Diese Taskbezeichner können mit Hilfe von Taskvariablen benutzt werden. Bei benannten Tasks kann der Bezeichner durch den Operator / bzw. die Prozedur ***task*** erhalten werden. Unbenannte Tasks, die im Taskbaum mit ihrem »Pseudonamen '-'« ausgewiesen sind, können über Verwandschaftsbeziehungen erreicht werden.

Für einige standardmäßig in einem L3 System vorhandene Tasks und einige der an sich unbenannten Realprozesse gibt es vereinfachende Schreibweisen bzw. Pseudonamen, die die Identifizierung vereinfachen.

Das gesamte Tasksystem wird von der Task SUPERVISOR verwaltet. Der SUPERVISOR kann als einzige Task im System spezielle Verwaltungsoperationen auf Objekten vom Typ Task ausführen. Erzeugen, Anhalten und Löschen einer Task geschehen also stets durch Aufträge an die Task SUPERVISOR. Informationen über den Aufbau des Systems, also Namen und Vater/Sohn Beziehungen der Tasks verwaltet der SUPERVISOR im *Systemkatalog*. Falls eine Task Informationen aus dem Systemkatalog benötigt, kann sie durch **access catalogue** eine Kopie dieses Katalogs vom SUPERVISOR anfordern.

Neben den Operationen, die allgemein den Typ TASK und die Verwandschaftsbeziehungen zwischen Objekten dieses Typs betreffen, gibt es weitere, die die Interna einzelner Tasks behandeln. Darunter fallen **storage** (belegter Speicherplatz), Tasknummer (**index**) oder Statusangaben. Alle diese Angaben stammen aus dem Prozeßkontrollblock (PCB) der jeweiligen Task im Systemkern. Die Prozeduren am Ende dieses Abschnitts stellen die ELAN Schnittstelle zum Systemkern dar.

5.3.2 Privilegien von Tasks

Die Tasks sind in drei Privilegklassen eingeteilt. Das höchste Privileg hat die Task SUPERVISOR. Nur diese Task darf Objekte des Typs TASK anlegen und löschen. Die mittlere Stufe, das *Systemprivileg*, haben die Tasks des Systemzweigs. Diese Tasks können insbesondere für jede Task, außer ihren eigenen Vorfahren, einen Ende Auftrag absetzen, der von SUPERVISOR akzeptiert wird. Man kann also z.B. von der Task OPERATOR aus jede Task außer SUPERVISOR und SYSUR löschen (VORSICHT!).

Das niedrigste Privileg (das *Benutzerprivileg*) haben die Task PUBLIC und ihre Nachkommen. In diesem Zweig des Systems dürfen bzw. können Tasks nur sich selbst oder ihre Nachkommen löschen.

Privilegien werden vererbt, Tasks unter SYSUR haben also stets das Systemprivileg, Tasks unter PUBLIC das Benutzerprivileg.

5.3.3 Restriktionen für Tasks

Bei einem Betriebssystem mit virtueller Speicherverwaltung und Datenraumsharing (»faules Kopieren«) kann für eine Task nicht angegeben werden, daß sie z.B. x Prozent des Plattenplatzes einnehmen darf, da das Erreichen eines solcher Wertes nicht effizient bestimmt werden kann. Um andererseits zu verhindern, daß Tasks das System unkontrolliert bis zur Aktionsunfähigkeit vollschreiben, kann für alle Tasks eine Schranke vorgegeben werden, bei deren Erreichen sie von der Arbeit zeitweise suspendiert werden. So wird erreicht, daß diese Tasks keine neuen Datenräume anlegen etc. Zumindestens einigen Tasks mit Systemprivileg muß zugestanden werden, bis 100% agieren zu dürfen, um gegebenfalls von hier aus andere Tasks zu löschen. Für Tasks mit Benutzerprivileg sollte dagegen eine Schranke angegeben werden, die mindestens 10 MB Freiraum läßt. Die Einstellung dieser Schranke kann im privilegierten Systemzweig mit der Prozedur **limit** vorgenommen werden.

5.3.4 Prioritäten von Tasks

L3 ist ein Timesharing Betriebssystem. Die Zuteilung von Rechenzeit an Tasks erfolgt durch den Scheduler im Systemkern, der allen Prozessen reihum eine Zeitscheibe von 10 Millisekunden anbietet. Dieses Verfahren wird trotz der großen Anzahl von Prozessen in einem L3 System durch einige Randbedingungen sehr effizient:

Grundsätzlich erhalten nur Systemprozesse (Treiber, Kern) jede ihrer mögliche Zeitscheiben. Einige wichtige Tasks (Supervisor, CONSOLE) werden nur einmal (prio = 1), normale Tasks werden generell zwei- (prio = 2) bis fünfmal (prio = 5) bei der Vergabe übergangen. Desweiteren warten die meisten Prozesse im System auf die Übertragung einer Botschaft. Falls die Botschaft noch nicht da ist, verbraucht der Prozeß nicht seine zustehenden 10 Millisekunden, sondern teilt dem Kern per Systemcall mit, daß der Scheduler den nächsten Prozeß bedienen kann. Durch dieses Verfahren bleiben im Endeffekt nur wenige Prozesse, die ganze Zeitscheiben verbrauchen. Die Tasks haben, wie oben erwähnt eine Priorität zwischen 2 und 5. Diese Priorität wird dynamisch vergeben: Tasks, die viel Rechenzeit verbrauchen, also selten oder garnicht warten, sondern »rechnen«, sinken in ihrer Priorität, Tasks, die viel von ihren Zeitscheiben »weitergeben«, weil sie z.B. im wesentlichen auf Eingaben warten, werden belohnt und behalten ihre Priorität 2.

5.3.5 Operatoren und Prozeduren

TASK	Datentyp
:=	Zuweisung
/	Taskbezeichner liefern
<, <=	Test auf Vater/Sohn Beziehung
=, <>	Test auf Gleichheit
brother	
father	
myself	
niltask	
public	
son	
supervisor	
exists	Gültige Taskvariable
exists task	Gültige Task zu*name*
isniltask	Ungültige Task
name	Externer Taskname
system task	Taskbezeichner eines Realprozesses
task	Interner Taskbezeichner
xbrother	
xexiststask	
xfather	
xname	
xson	
xtask	
access catalogue	Systemkatalog anfordern
begin	Task erzeugen
end	Task löschen
halt	Programm in Task unterbrechen
rename myself	Umbenennen
callee	Liefert receive Partner
cpu time	Verbrauchte Rechenzeit
dataspaces	Datenräume
index	Taskindex
limit	Speicherlimit
prio	Taskpriorität
privilege	Taskprivileg
user privilege	
system privilege	
supervisor privilege	
status	Taskstatus
storage	Belegter Speicherplatz
wakeup time	Wartezeitende

TASK

TYPE TASK

Wirkung: Datentyp »interner Taskbezeichner«.

:=

OP := (TASK VAR left, TASK CONST right)

Wirkung: Zuweisungsoperator für Taskbezeichner.

= <>

BOOL OP = (TASK CONST left, right)

BOOL OP <> (TASK CONST left, right)

Wirkung: Liefert TRUE, falls 'left' gleich (bzw. ungleich) 'right' ist.

< <=

BOOL OP < (TASK CONST left, right)

BOOL OP <= (TASK CONST left, right)

Wirkung: Prüfung auf die Relation. Liefert 'TRUE', falls 'left' Nachkomme (Sohn, Enkel,..) von 'right' ist.

/

TASK OP / (TEXT CONST task name)

Wirkung: Liefert den internen Taskbezeichner der Task des angegebenen Namens, falls sie existiert. Der eigene Katalog wird automatisch aktualisiert (identisch mit: `task (TEXT CONST task name)`.

Fehler: * ... gibt es nicht

brother

TASK PROC brother (TASK CONST task)

Wirkung: Liefert den nächsten Bruder von 'task'. Falls kein Bruder existiert, wird 'niltask' geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

father

TASK PROC father

Wirkung: Liefert den internen Taskbezeichner der eigenen Vatertask.

TASK PROC father (TASK CONST task)

Wirkung: Liefert den internen Taskbezeichner der Vatertask von 'task'. Existiert keine Vatertask, wird niltask geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

myself

TASK PROC myself

Wirkung: Liefert den eigenen internen Taskbezeichner.

niltask

TASK CONST niltask

Wirkung: Bezeichner für »keine Task«. So liefern die Prozeduren 'son', 'brother' und 'father' als Resultat 'niltask', wenn keine Sohn-, Bruder- oder Vatertask existiert.

public

TASK PROC public

Wirkung: Liefert den Taskbezeichner der Task PUBLIC.

son

TASK PROC son (TASK CONST task)

Wirkung: Liefert den Taskbezeichner der ersten Sohntask von 'task'. Falls keiner im Katalog vermerkt ist, wird 'niltask' geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

supervisor

TASK PROC supervisor

Wirkung: Liefert den Taskbezeichner der Task SUPERVISOR.

exists

BOOL PROC exists (TASK CONST task)

Wirkung: Liefert 'TRUE', falls der mit 'task' spezifizierte Prozeß existiert. Hiermit kann also auch die Existenz von Realprozessen überprüft werden.

exists task

BOOL PROC exists task (TEXT CONST taskname)

Wirkung: Liefert 'TRUE', falls eine Task mit Namen 'taskname' existiert, sonst 'FALSE'.

is niltask

BOOL PROC is niltask (TASK CONST task)

Wirkung: task = niltask

name

TEXT PROC name (TASK CONST task)

Wirkung: Liefert den Namen von 'task'. Die Task muß noch im System existieren. Falls die 'task' noch nicht im eigenen Katalog enthalten ist, wird dieser aktualisiert.

systemtask

TASK PROC systemtask (TEXT CONST systaskname)

Wirkung: Liefert den Taskbezeichner des durch 'systaskname' spezifizierten Realprozesses. Es lassen sich alle Treiberprozesse über diese Prozedur erreichen, wenn ihr Name bekannt ist.

task

TASK PROC task (TEXT CONST task name)

Wirkung: Liefert den Taskbezeichner der Task des angegebenen Namens, falls sie existiert. Der eigene Katalog wird automatisch aktualisiert.

Fehler: * ... gibt es nicht

Die folgenden Prozeduren entsprechen in ihrer Wirkung den ähnlich benamten

brother, *son* etc. Im Gegensatz zu jenen wird das Resultat jedoch nicht über den

Taskkatalog erreicht, sondern direkt beim SUPERVISOR angefragt. Die Prozeduren sind also direkte Kommunikationen vermittels des SUPERVISOR Protokolls (Kap.13.6).

xbrother

TASK PROC xbrother (TASK CONST task)

Wirkung: Fragt direkt bei SUPERVISOR nach **brother(task)**. Falls es keine derartige Task gibt, wird **niltask** als Resultat geliefert.

xexists task

BOOL PROC xexists task (TEXT CONST task)

Wirkung: Direkte Anfrage an die Task SUPERVISOR. Liefert 'TRUE', falls eine Task mit Namen 'task' existiert, sonst 'FALSE'.

xfather

TASK PROC xfather (TASK CONST task)

Wirkung: Fragt direkt bei SUPERVISOR nach **father(task)**. Falls es keine derartige Task gibt, wird **niltask** als Resultat geliefert.

xname

TEXT PROC xname (TASK CONST task)

Wirkung: Fragt direkt bei SUPERVISOR nach **name (task)**. Falls keine benannte Task existiert, wird **niltext** als Resultat geliefert.

xson

TASK PROC xson (TASK CONST task)

Wirkung: Fragt direkt bei SUPERVISOR nach **son (task)**. Falls es keine derartige Task gibt, wird **niltask** als Resultat geliefert.

access catalogue

PROC access catalogue

Wirkung: Die Task SUPERVISOR wird aufgefordert eine Kopie des Systemkatalogs zu senden, mit der die eigene Version überschrieben wird. Prozeduren, die systemglobale Informationen benötigen (**father**, **brother**, **son**) arbeiten dann auf dieser neuen Fassung des Katalogs.

Beispiel:

```
INT VAR i; TASK VAR t;
taskinfo;
(* Im Systemkatalog sind die Tasks noch nicht vorhanden *)
page;
FOR i FROM 5 DOWNTO 1
   REP begin("Task " + text(i),PROC unlink,t);
PER;
taskinfo;
(* Im Systemkatalog sind alle fünf Tasks vorhanden *)
FOR i FROM 1 UPTO 5
   REP end (son (myself), quiet); access catalogue
PER;
taskinfo;
(* Hiernach sind alle fünf Tasks im Systemkatalog wieder gelöscht *)
(* Wäre access catalogue nicht aufgerufen worden, *)
(* hätte es beim Löschen des zweiten Sohnes *)
(* die Fehlermeldung "'end' unzulaessig" gegeben, *)
(* da der taskeigene Katalog nicht aktuell war. *)
```

begin

PROC begin (TEXT CONST new taskname)

Wirkung: Eine neue Task mit dem angegebenen Namen wird als Sohntask der aufrufenden Task mit der Startprozedur **monitor** gestartet. Falls diese Prozedur im Monitor, also interaktiv, aufgerufen wird, wird die neue Task sofort an das Terminal gekoppelt.

Fehler: Eine Task mit dem Namen 'new taskname' existiert bereits.

PROC begin (TEXT CONST new taskname, father)

Wirkung: Eine neue Task mit dem angegebenen Namen wird als Sohntask der (Manager)Task 'father' mit der Startprozedur **monitor** gestartet.

Fehler: `falscher Auftrag für Task "father"(04000010,"sender")` => 'fathername' ist keine Managertask passenden Typs. 40000010 ist der (gesendete) Auftragscode des Dateimanagerprotokolls.

`wait for SYSIO` => 'fathertask' hat sich nicht selbst vom Terminal abgekoppelt

- zu viele Tasks
- Name unzulässig
- Eine Task mit dem Namen 'new taskname' existiert bereits

PROC begin (TEXT CONST new taskname, PROC startproc, TASK VAR t)

Wirkung: Es wird eine Task mit dem Namen 'new taskname' als Sohn der aufrufenden Task mit der Prozedur 'startproc' gestartet. 't' identifiziert die neue Task, falls kein Fehler aufgetreten ist.

PROC begin (PROC startproc, TASK VAR t)

Wirkung: Es wird eine unbenannte Task (Pseudoname "–") als Sohn der aufrufenden Task mit der Prozedur 'startproc' gestartet. 't' identifiziert die neue Task, falls kein Fehler aufgetreten ist. Bei Verwendung des Pseudonamens kann keine Namenskollision auftreten.

end

PROC end (TASK CONST victim)

Wirkung: Nach Kontrollfrage (mit Prozedur **yes**) wird an die Task SUPERVISOR der Auftrag geschickt, die Task 'victim' zu löschen. Der Auftrag wird nur akzeptiert, wenn die auftraggebende Task eine höhere Privilegstufe als 'victim' hat oder 'victim' ein Nachkomme des Auftraggebers ist. Falls der Auftrag akzeptiert wird, beendet SUPERVISOR 'victim' und damit implizit auch alle Nachkommen von 'victim'.

PROC end (TASK CONST victim, QUIET CONST quiet)

Wirkung: Löschen ohne Kontrollfrage.

PROC end

Wirkung: Wie: end (myself)

halt

PROC halt (TASK CONST victim)

Wirkung: An die Task SUPERVISOR wird der Auftrag geschickt, der Task 'victim' einen Fehler zu senden. Die Wirkung in 'victim' entspricht dem Auftreten eines Laufzeitfehler im gerade laufenden Programm. Näheres siehe Kapitel 11.

Der Auftrag wird nur akzeptiert, wenn die auftraggebende Task eine höhere Privilegstufe als 'victim' hat oder 'victim' ein Nachkomme des Auftraggebers ist.

rename myself

PROC rename myself (TEXT CONST new name)

Wirkung: An die Task SUPERVISOR wird der Auftrag geschickt, den Namen der Task in 'new name' zu ändern. Damit werden alle Taskvariablen, die sich auf diese Task beziehen, ungültig. Es ist auch möglich, eine Task mit dem Pseudonamen "–" zu benennen, bzw. eine benannte Task »zur Strichtask« zu machen (name "--").

5.3.6 Verwaltung von Tasks

callee

TASK PROC callee (TASK CONST task)

Wirkung: Liefert den Partner von 'task', auf den 'task' im geschlossenen Wartezustand (**receive**) wartet. Falls 'task' nicht im geschlossenen Warten verharrt, wird **niltask** geliefert.

cpu time

DINT PROC cpu time (TASK CONST task)

Wirkung: Liefert die verbrauchte Rechenzeit von 'task'.

dataspaces

INT PROC dataspaces (TASK CONST task)

Wirkung: Liefert die Anzahl der Datenräume von 'task'.

INT PROC dataspaces

Wirkung: Liefert die Anzahl der eigenen Datenräume. (`put (dataspaces(myself))`).

index

INT PROC index (TASK CONST task)

Wirkung: Liefert den Taskindex von 'task'. Der Index ist der zeitinvariante Teil des internen Taskbezeichners.

Bemerkung: Der Taskindex ermöglicht die eindeutige Identifikation von »Strichtasks« (Tasks mit Pseudonamen -).

Beispiel: `put (hex(index(son(printer))))`

limit

PROC limit (TASK CONST task, INT CONST limit)

Wirkung: 'task' wird von der Speicherverwaltung blockiert, wenn 'limit' Prozent des Speicherplatzes des Systems belegt sind. Diese Einstellung kann nur von Tasks mit Systemprivileg vorgenommen werden!

Fehler: `'limit'-Angabe muß zwischen 1 und 100 liegen`

INT PROC limit (TASK CONST task)

Wirkung: Liefert den eingestellten Wert für 'task'.

prio

PROC prio (TASK CONST task, INT CONST prio)

Wirkung: Die Priorität von 'task' wird auf 'prio' gesetzt. Diese Einstellung kann nur von Tasks mit Systemprivileg vorgenommen werden!

INT PROC prio (TASK CONST task)

Wirkung: Liefert die Priorität von 'task'.

privilege

INT PROC privilege (TASK CONST task)

Wirkung: Liefert das Privileg von 'task'. Die Privilegstufen sind:

2: Supervisorprivileg; darf Tasks erzeugen. Nur Task SUPERVISOR.

`1`: Systemprivileg; erlaubt »Systemverwalteroperationen«. Task SYSUR und Nachkommen.

`0`: Benutzerprivileg; Task PUBLIC und Nachkommen.

Bemerkung: Für Abfragen der vorhandenen Berechtigungsstufe gibt es wertliefernde Prozeduren sinntragenden Namens:

INT PROC user privilege

INT PROC system privilege

INT PROC supervisor privilege

status

INT PROC status (TASK CONST task)

Wirkung: Liefert den Taskstatus von 'task'. Möglich sind:

`wait`: Task wartet auf Botschaft.

`busy`: Task ist aktiv.

`blkd`: Task ist blockiert

`dead`: Exitus. Die Task kann keine Aktion mehr ausführen. Sie kann nur noch gelöscht werden.

storage

INT PROC storage (TASK CONST task)

Wirkung: Liefert den belegten Speicherplatz von 'task'. Die Angabe besagt, wieviel Platz die Datenräume von 'task' beanspruchen. Die Angabe erfolgt in Byte und ist (wegen Paging) stets durch 4096 ohne Rest teilbar.

Bemerkung: Irgendwelche Rückschlüsse aus Angaben für mehrere Tasks sind nicht möglich.

wakeup time

DINT PROC wakeup time (TASK CONST task)

Wirkung: Wenn 'task' ein timeout gesetzt hat, um auf eine Botschaft zu warten oder durch **pause**, so liefert diese Prozedur den Zeitpunkt, zu dem diese Wartezeit abläuft.

Bemerkung: Das Ergebnis ist eine **systemtime** Angabe, die noch anstehende Wartezeit einer Task kann also z.B. durch: `put ( time (wakeup time (task) - systemtime, 8,3)` angezeigt werden.

6 Ein-/Ausgabe auf dem Bildschirm

6.1 Einführung

Die für das ELAN System zur Verfügung stehenden Ein-/Ausgaberoutinen lassen sich in diverse Kategorien einteilen. Die erste Gruppe sind die elementaren Ein/Ausgaben, die Cursorpositionierung und die Einstellung von Farbattributen. In der folgenden Übersicht sind die Prozeduren, die in Hinsicht auf Steuerzeichen »EUMEL- kompatibel« sind, durch `(E)` markiert.

In der zweiten Gruppe sind die umleitbaren Ein-/Ausgaben zusammengestellt. Diese Eingaben können auch aus einer Datei bezogen, die Ausgaben in eine Datei gelenkt werden.

Darauf aufbauend gibt es Operationen zur Abwicklung des Kommandodialogs, also der Abfrage bei Löschen einer Daten (`"xyz" löschen (j/n) ?`) und dergleichen.

Die nächste Kategorie umfaßt die Prozeduren, die die Kommunikation mit einem Ein-/Ausgabe Partner vorbereiten und dessen Konfiguration abfragen.

In 6.6 finden sich Operationen, die in angelsächsischen Handbüchern unter der Überschrift »Miscellaneaous« zusammengefaßt werden: Operationen, die zwar auch unter in einer der ersten drei Gruppen unterzubringen wären, auf Grund ihrer Subtilität und selteneren Anwendung jedoch aus den allgemeinen Operationen herausgezogen sind. Hierunter fallen z.B. das Auslesen von Scancode und Asciicode einer Taste oder das explizite Auffrischen der Bildschirmausgaben auf einem Terminal.

Abschließend ist das Thema ELAN Grafik behandelt. Dieser Abschnitt nimmt durch seine Hardwarenähe eine Sonderstellung ein. Die beschriebenen Mittel dienen der Ausnutzung der an der Console verfügbaren Hardware. Die in den Rechner eingebaute Grafikkarte läßt sich in einen Grafik- oder Zeichenmodus schalten und es sind Ausgaben in diesem Modus möglich. Es handelt bei diesen Prozeduren also nicht um eine erlesene Grafik-Programmierschnittstelle sondern um einen Hardwarezugriff.

Elementare Ein-/Ausgabe

editget	Zeileneditor
in (E)	Eingabe erwarten
get secret	Textzeile ohne Echo lesen
get secret line (E)	— " —
inchar (E)	Auf Zeichen warten
incharety (E)	Zeichen einlesen, falls vorhanden
pause (E)	Warten
cout	Bedingte Ausgabe von Integer
out (E)	Textausgabe
outsubtext (E)	Teiltext ausgeben
outtext (E)	Teiltext ausgeben
TIMESOUT (E)	Text mehrfach ausgeben
page	Bildschirm löschen
show cursor	Cursor an geforderte Position setzen
get cursor pos	Cursorposition abfragen
set cursor pos	Cursor setzen
cursor on / off	Cursor an / ausschalten
get attribute	Farbattribut abfragen
set attribute	Farbattribut setzen
display scroller	Fenster scrollen und füllen
display string	Text ausgeben

Umleitbare Ein-/Ausgabe

get	Eingabe einlesen
getline	Textzeile einlesen
line	Zeilenwechsel
put	Ausgabe
putline	Ausgabe mit Zeilenvorschub
write	Ausgabe

Kommandodialog

command dialogue
lastparam
no
paramposition
quiet
say
std
yes

Einstellungen

online	Terminalzuordnung
sysin partner	Tastaturmanager
sysout partner	Bildschirmmanager
x size	Bildschirmspalten
y size	Bildschirmzeilen

Sonstige

copy to screen	In Bildschirmspeicher schreiben
in key	Tastaturcodes
in scan ascii	— " —
screen write access	Ausgabe erzwingen
terminal necessary	Virtuelles Terminal

Grafik

SCREEN	Datentyp Bildschirm
SCREENMODE	Datentyp Bildschirm mit Einstellungen
:=	Zuweisungsoperator
=	Vergleichsoperator
screen mode	Screenmode Einstellung
alpha 80	Vordefinierte Einstellung
alpha 40	— " —
alpha 80 hr	— " —
cga mono	— " —
cga colour	— " —
ega	— " —
vga	— " —
mcga	— " —
char height	Liefert eingestellten Wert
char width	— " —
memory base	— " —
memory size	— " —
mode	— " —
page size	— " —
screen height	— " —
screen width	— " —
screen page	Bildschirmseite einstellen

6.2 Elementare Ein-/Ausgabe

editget

PROC editget (TEXT VAR line)

Wirkung: Zeileneditor. An der aktuellen Cursorposition wird der Text 'line'ausgegeben. 'line' kann beliebig überschrieben, ergänzt etc. werden. Die Eingabe wird durch »ENTER« abgeschlossen. Im Gegensatz zur Prozedur **get** ist auch eine leere Eingabe möglich.

PROC editget (TEXT VAR line, INT CONST width, scroll, TEXT CONST sep, res, TEXT VAR exit)

Wirkung: Wie oben, jedoch ist die Länge des eingegebenen Textes auf 'width' Zeichen begrenzt. Die Eingabe wird durch 'ENTER' oder durch eine Cursorbewegung über die Position 'width' hinaus abgeschlossen. Die Angabe 'scroll' setzt die Breite des Zeilenfensters fest. Wird diese Breite überschritten, so wird 'line' gescrollt. In 'sep' (Separator) können Zeichen festgesetzt werden, mit denen die Eingabe beendet wird (zusätzlich zu »ENTER«). In 'res' (reservierte Tasten) können Tasten festgelegt werden, die in Verbindung mit »ESC« die Eingabe beenden. Wurde der Zeileneditor durch einen Separator verlassen, so steht in 'exit' der Separator. Wurde der Zeileneditor über eine reservierte Taste verlassen, so enthält 'exit' 'ESC' und die Taste.

PROC editget (TEXT VAR line, INT CONST width, scroll)

Wirkung: Wie oben, jedoch ohne die Möglichkeit, die Eingabe mit einem Separator oder einer reservierten Taste zu beenden.

PROC editget (TEXT VAR line, TEXT CONST sep, res, TEXT VAR exit)

Wirkung: Wie oben, jedoch ohne die Möglichkeit, die Länge des Textes (Anzahl der Zeichen) zu begrenzen und ohne Scrollmöglichkeit.

PROC editget (TEXT VAR line, INT CONST width, TEXT VAR exit)

Wirkung: Wie oben, jedoch ohne die Möglichkeit, einen Separator oder eine reservierte Taste zu benutzen. Die Eingabe kann mit »ENTER« vor Erreichen von 'width' beendet werden.

in

TEXT PROC in (INT CONST milli)

Wirkung: Wartet 'milli' Millisekunden auf die Eingabe eines Zeichens.

get secret

PROC get secret (TEXT VAR secretline, INT CONST timeout)

Wirkung: Liest einen Text ein. Statt der eingegebenen Zeichen werden nur Punkte ausgegeben. Mit Bachspace kann zurückpositioniert werden, Return schließt die Eingabe ab. Falls 'timeout' Millisekunden keine Eingabe erfolgte oder die Task vom Terminal abgekoppelt wurde, wird die Eingabe abgebrochen und 'secretline' wird niltext zugewiesen.

get secret line

PROC get secret line(TEXT VAR secretline)

Wirkung: Wie **editget (secretline)**, jedoch werden statt der eingegebenen Zeichen auf dem Bildschirm nur Punkte ausgegeben.

inchar

PROC inchar (TEXT VAR char)

Wirkung: Wartet bis ein Zeichen eingegeben wird und schreibt dieses Zeichen in die Variable 'char'.

incharety

TEXT PROC incharety (INT CONST timelimit)

Wirkung: Wartet 'timelimit' Zehntelsekunden auf die Eingabe eines Zeichens. Wenn eine Taste gedrückt wird, also ein Zeichen in den Eingabepuffer geschrieben wird, wird dieses Zeichen geliefert. Falls die Wartezeit abgelaufen ist, ohne daß ein Zeichen im Eingabepuffer steht, wird **niltext** geliefert.

TEXT PROC incharety

Wirkung: Wirkt wie **incharety (0)**. Es wird versucht, ein Zeichen aus dem Eingabepuffer zu holen.

Beispiel:
```
(* Eingabepuffer leeren ohne die Zeichen zu verwerten *)
REP UNTIL incharety = "" PER
```

pause

PROC pause (INT CONST limit)

Wirkung: Wartet 'limit' Zehntelsekunden oder bis eine Taste gedrückt wird. Das eingegebene Zeichen wird 'verschluckt'. Ein negativer Wert für 'limit' wirkt wie **pause(0)**.

PROC pause

Wirkung: Wartet auf die Eingabe eines Zeichens.

cout

PROC cout (INT CONST number)

Wirkung: 'number' wird beginnend an der aktuellen Cursorposition auf dem Bildschirm ausgegeben. Danach wird wieder an die ursprüngliche Cursorposition zurückpositioniert. Falls die Ausgabe außerhalb der Grenzen **x size**, **y size** liegt, wird sie abgeschnitten. **cout** benötigt kein Terminal. Falls die Task nicht **online** ist, bleibt ein Programm also nicht stehen sondern läuft weiter, wenn die Ausgabe über **cout** erfolgt.

out

PROC out (TEXT CONST text)

Wirkung: Der Text 'text' wird beginnend an der aktuellen Cursorposition auf dem Bildschirm ausgegeben. Falls die Ausgabe außerhalb der Grenzen **x size**, **y size** liegt, wird sie abgeschnitten. Falls die Task nicht **online** ist, wird mit der Ausgabe gewartet bis die Task wieder einen **sysout partner** (Bildschirm) hat. Steuerzeichen in 'text' werden nach EUMEL Konvention interpretiert.

outsubtext

PROC outsubtext (TEXT CONST text, INT CONST from, to)

Wirkung: Ein Teiltext von 'text' wird beginnend an der aktuellen Cursorposition auf dem Bildschirm ausgegeben. Der Teiltext beginnt am 'from'ten Zeichen und endet mit dem 'to'ten Zeichen. Falls 'text' diesen Rahmen nicht füllt, wird nur bis zum Ende des Textes ausgegeben. Falls **LENGTH (text) < from** wird nichts ausgegeben. Falls die Ausgabe außerhalb der Grenzen **x size**, **y size** liegt, wird sie abgeschnitten. Falls die Task nicht **online** ist, wird mit der Ausgabe gewartet bis die Task wieder einen **sysout partner** (Bildschirm) hat.

PROC outsubtext (TEXT CONST text, INT CONST from)

Wirkung: Wie oben, aber ab dem 'from'ten Zeichen bis zum letzten Zeichen von 'text'.

outtext

PROC outtext (TEXT CONST text, INT CONST from, to)

Wirkung: Wie **outsubtext (text, from, to)**, jedoch werden für fehlende Zeichen werden Leerzeichen ausgegeben, so daß auf alle Fälle `to + 1 - from` Zeichen ausgegeben werden.

TIMESOUT

OP TIMESOUT (INT CONST times, TEXT CONST text)

Wirkung: Der Text 'text' wird beginnend an der aktuellen Cursorposition 'times' mal auf dem Bildschirm ausgegeben. Falls die Ausgabe außerhalb der Grenzen **x size**, **y size** liegt, wird sie abgeschnitten. Falls die Task nicht **online** ist, wird mit der Ausgabe gewartet bis die Task wieder einen **sysout partner** (Bildschirm) hat.

page

PROC page

Wirkung: Die aktuelle Bildschirmseite wird gelöscht und der Cursor auf die Position (0,0) gesetzt.

get cursor pos

PROC get cursor pos(INT VAR x, y)

Wirkung: Liefert in 'x'und 'y' die aktuellen Cursorkoordinaten. (0,0) ist die linke obere Ecke.

set cursor pos

PROC set cursor pos(INT CONST x, y)

Wirkung: Setzt den Cursor logisch auf die Koordinaten 'x', 'y'. Der Cursor wird nur dann wirklich umpositioniert, wenn auch eine Ausgabe an der neuen Position erfolgt.

Zulässige Werte:

```
0 <= x <= xsize - 1
0 <= y <= ysize - 1
x,y < 0 ==>  x,y := 0
x > xsize ;y > ysize  ==>  x := xsize; y := ysize -1
```

show cursor

PROC show cursor (INT CONST x,y)

Wirkung: Setzt den Cursor an die Position 'x', 'y' und zeigt den Cursor dort an, auch wenn keine Eingabe erfolgt. Grenzen: `0 <= x <= xsize - 1 0 <= y <= ysize - 1`

cursor on

PROC cursor on

Wirkung: Der Cursor wird angeschaltet. Falls z.B. irgendein DOS Programm den Cursor ausgeschaltet hat, kann er mit dieser Prozedur wieder sichtbar gemacht werden.

Bemerkung: Funktioniert natürlich nur in Textmodi (siehe 6.7).

cursor off

PROC cursor on

Wirkung: Der Cursor wird abgeschaltet.

get attribute

PROC get attribute (INT VAR color)

Wirkung: Liefert in 'color' den eingestellten Attributcode.

set attribute

PROC set attribute (INT CONST color)

Wirkung: Das Farbattribut für die Bildschirmdarstellung wird auf 'color' gesetzt. Alle folgenden Ausgaben erscheinen in dieser Farbdarstellung.

Bemerkung: Bei der Zeichenausgabe auf CGA/EGA/VGA Bildschirmen wird für jedes Zeichen im darauffolgenden Byte das Farbattribute festgehalten. Die verfügbaren Farben sind Rot, Grün, Blau und daraus kombinierte.Das Codierungsverfahren ist Folgendes: Das höherwertige Halbbyte legt die Hintergrundfarbe fest, das höchstwertige Bit gibt an, ob die Darstellung blinkend oder ruhig ist. Das niederwertige Halbbyte gibt die Schriftfarbe an, das Bit Nummer 3, also das höchstwertige dieses Halbbytes gibt an, ob die Farbe hell oder matt sein soll. Das Standardattribut ist `0x07` (Hintergrund ruhig schwarz, Vordergrund matt weiß). `Attributbit: 0 Blau 1 Grün Zeichen 2 Rot 3 hell 4 Blau 5 Grün Hintergrund 6 Rot 7 blinkend`

Beispiel: `set attibute (0b0111 1100)` liefert hellrote Schrift auf ruhigem, weißem Hintergrund.

display cursor

PROC dsiplay cursor (INT CONST x,y)

Wirkung: Setzt den Cursor an die Position 'x', 'y' und zeigt den Cursor dort an, auch wenn keine Eingabe erfolgt. Grenzen: `0 <= x <= xsize - 1 0 <= y <= ysize - 1`

display scroller

PROC display scroller (INT CONST x, y, width, height, lines, code, attr)

Wirkung: In dem durch die linke obere Ecke ('x','y'), seine Breite 'width' und seine Höhe 'height' beschriebenen Bildschirmfenster wird der Fensterinhalt um 'lines' Zeile gescrollt (nach oben oder unten verschoben). Der durch die Verschiebung freiwerdende Fensterausschnitt wird mit Zeichen 'code' in der Farbdarstellung 'attr' gefüllt. Eine negativer Wert für 'lines' bewirkt Verschiebung nach unten, eine positiver Wert Verschiebung nach oben.

Beispiel: `display scroller (20, 8, 30, 7, -2, code ("?"),0x7c)`
Ab der Position 20,8 wird ein 30 Zeichen breites und 7 Zeilen hohes Fenster um 2 Zeilen nach unten Verschoben, der Freiraum wird mit hellroten Fragezeichen aufgefüllt.

display string

PROC display string (TEXT CONST string)

Wirkung: Ab der aktuellen Cursorposition wird 'string' in der eingestellten Farbe ausgegeben.

Bemerkung: Diese Prozeduren namens »display *« erzeugen eine Ausgabe, wenn die Task an einem Terminal hängt. Falls dies nicht der Fall ist, geht die Ausgabe verloren.

6.3 Umleitbare Prozeduren

Die folgenden Eingabeprozeduren lesen von der Tastatur. Die Eingabequelle kann jedoch durch die Prozedur **sysin** umgestellt werden. Falls mit **sysin** eine Datei vereinbart wird, so werden die folgenden Eingaben statt von der Tastatur aus dieser Datei gelesen.

Für die Ausgaben kann mit **sysout** eine Umleitung vereinbart werden. Die folgenden Ausgaben werden dann - statt auf das Terminal - in die angegebene Datei geschrieben.

sysin

PROC sysin (TEXT CONST filename)

Wirkung: Falls 'filename' eine existierende Datei bezeichnet, lesen alle umlenkbaren Eingaberoutinen aus dieser Datei. Durch **sysin("")** wird die Eingabe zurückgestellt.

TEXT PROC sysin

Wirkung: Liefert den Namen einer eingestellten Sysin-Datei oder niltext, falls keine Datei eingestellt ist.

get

PROC get (TEXT VAR word, TEXT CONST sep)

Wirkung: Liest in die Variable 'word' einen Text ein. Die Eingabe wird durch den Standardseparator 'Zeilenwechsel' (ENTER, `""13""`) oder eines der in 'sep' festgelegten Zeichen beendet. Abschließende Leerzeichen werden abgeschnitten.

PROC get (TEXT VAR word, INT CONST length)

Wirkung: Liest in die Variable 'word' einen Text ein. Die Eingabe wird nach dem 'length'ten Zeichen oder nach der Eingabe von »ENTER« abgebrochen.

PROC get (TEXT VAR word)

Wirkung: Liest in die Variable 'word' einen Text ein. Die Eingabe wird durch »ENTER« oder falls mindestens ein Zeichen eingegeben wurde, dessen Code > 32 ist - auch durch ein Leerzeichen - beendet. Führende Leerzeichen werden anschließend abgeschnitten.

PROC get (INT VAR number)

Wirkung: Wie :
```
TEXT VAR number word;
get (number word);
number := int (number word)
```

PROC get (DINT VAR dnumber)

Wirkung: Wie : `TEXT VAR number word;`
```
get (number word);
dnumber := dint (number word)
```

PROC get (INT VAR rnumber)

Wirkung: Wie : `TEXT VAR number word;`
```
get (number word);
rnumber := real (number word)
```

Bemerkung: Wenn die Eingabe für obige Prozeduren ungültige Zeichen enthält, wird die Eingabe bis zur Fehlerstelle ausgewertet und **last conversion ok** wird auf **FALSE** gesetzt. Falls die Eingabe korrekt abgeschlossen wurde, liefert last conversion 'TRUE'.

getline

PROC getline (TEXT VAR textline)

Wirkung: Liest in 'textline' einen Text ein. Das Einlesen wird mit der Eingabe von »ENTER« beendet.

sysout

PROC sysout (TEXT CONST filename)

Wirkung: Falls 'filename' eine existierende Datei bezeichnet, schreiben alle umlenkbaren Ausgaberoutinen in diese Datei. Falls 'filename' <> **niltext** ist, aber noch nicht existiert, wird eine Datei diese Namens angelegt und die Ausgaben in diese Datei geschrieben. Durch **sysout("")** wird die Ausgabe zurückgestellt.

TEXT PROC sysout

Wirkung: Liefert den Namen einer eingestellten Sysout-Datei oder niltext, falls keine Datei eingestellt ist.

line

PROC line (INT CONST lines)

Wirkung: Es wird um 'lines' Zeilen nach unten auf den Zeilenanfang positioniert. Dabei wird ggf. gescrollt, der Cursor steht dann auf der ersten Position der letzten Bildschirmzeile.

PROC line

Wirkung: Es wird auf den Anfang der nächsten Ausgabezeile positioniert.

put

PROC put (TEXT CONST text)

Wirkung: Ausgabe von 'text', gefolgt von einem Leerzeichen. Die einzelnen Ausgaben auf einer Zeile sind also voneinander getrennt. Falls eine Ausgabe über die Zeilengrenze hinausreichen würde, wird vorher in die nächste Zeile positioniert.

PROC put (INT CONST number)

Wirkung: Wie : `put (text (number))`

PROC put (DINT CONST dnumber)

Wirkung: Wie : `put (text (dnumber))`

PROC put (REAL CONST rnumber)

Wirkung: Wie : `put (text (rnumber))`

putline

PROC putline (TEXT CONST text)

Wirkung: Der Inhalt der Variablen 'text' wird ausgegeben. Anschließend erfolgt ein Zeilenwechsel. Die Prozedur entspricht also: `write (text); line`

write

PROC write (TEXT CONST text)

Wirkung: Der Inhalt der Variablen 'text'wird ausgegeben. Die einzelnen Ausgaben auf einer Zeile sind nicht voneinander getrennt. Falls eine Ausgabe über die Zeilengrenze hinausreichen würde, wird vorher in die nächste Zeile positioniert.

6.4 Kommandodialog

Der Kommandodialog dient zur zentralen Steuerung und einfachen Durchführung von Dialogen wie

`"datei" loeschen (j/n)?`

Er wird von allen Systemteilen verwandt, die einen Kommandodialog mit dem Benutzer aufnehmen. Anwenderprozeduren mit ähnlichen Problemen sollten genauso damit arbeiten.

Der Kommandodialog kann in der Task aus- und eingeschaltet werden.

command dialogue

BOOL PROC command dialogue

Wirkung: Liefert den aktuellen Zustand des Kommandodialogs: TRUE - Dialog soll geführt werden! FALSE - Dialog soll nicht geführt werden!

PROC command dialogue (BOOL CONST status)

Wirkung: Schaltet den Kommandodialog ein ('status' = TRUE) oder aus ('status' = FALSE). Der alte Zustand wird überschrieben. Soll später wieder in den alten Zustand zurückgeschaltet werden, muß er vorher erfragt und gesichert werden.

last param

TEXT PROC last param

Wirkung: Liefert den zuletzt gesetzten Parameter-Text (siehe folgende Prozedur). Falls 'command dialogue' = TRUE und die 'param position' > 0 ist, wird der Parametertext als Standardparameter an der angegebenen x-Position eine Zeile höher in der Form ("...") ausgegeben. Diese Prozedur wird von den parameterlosen Kommandos bzw. Prozeduren wie **edit**, **run** usw. verwandt, um mit dem Standardparameter weiterzuarbeiten.

PROC last param (TEXT CONST new)

Wirkung: Setzt 'last param' auf 'new'. (Das Setzen muß explizit durchgeführt werden). **last param** wird beispielsweise von den einparametrigen Prozeduren **edit** und **run** gesetzt.

no

BOOL PROC no (TEXT CONST question)

Wirkung: a) command dialogue -> TRUE Frage und Antwort wie bei 'yes'. Das Resultat ist TRUE bei verneinender Antwort (n,N) FALSE bei bejahender Antwort (j,J,y,Y) b) command dialogue -> FALSE Keine Aktion, das Resultat ist FALSE.

param position

PROC param position (INT CONST x)

Wirkung: Setzt die Echoposition für 'last param'. Bei x=0 wird ein Echo unterdrückt.

quiet

QUIET PROC quiet

Wirkung: Unterdrückt die **yes/no** Abfrage und liefert 'TRUE' für die betreffende Aktion.

Beispiel: `delete (std, quiet)`
die zuletzt bearbeitete Datei wird ohne Dialog gelöscht.

say

PROC say (TEXT CONST message)

Wirkung: `IF command dialogue THEN out (text) FI`

std

TEXT PROC std

Wirkung: Liefert wie 'last param' den zuletzt gesetzten Parameter. Im Gegensatz dazu wird der Parameter aber nicht ausgegeben.

yes

BOOL PROC yes (TEXT CONST question)

Wirkung: a) command dialogue –> TRUE Der übergebene Fragetext wird durch " (j/n)?" ergänzt auf dem Terminal ausgegeben. Als Antwort wird eine der Tasten <j>, <J>, <y>, <Y>, <n>, <N> akzeptiert; jede andere Eingabe führt zur Fragewiederholung. Das Resultat der Prozedur ist TRUE bei bejahender Antwort (j,J,y,Y) FALSE bei verneinender Antwort (n,N) b) command dialogue –> FALSE Keine Aktion, das Resultat ist TRUE.

6.5 Einstellungen

online

BOOL PROC online

Wirkung: Liefert 'TRUE', falls die Task an ein Terminal gekoppelt ist.

Bemerkung: Die Task muß oberstes Element eines Terminalstacks sein, siehe dazu (9.2) **get terminal stack**.

BOOL PROC online (TASK CONST taskname)

Wirkung: Liefert 'TRUE', falls die Task 'taskname' an ein Terminal gekoppelt ist.

sysin partner

TASK PROC sysin partner

Wirkung: Liefert den internen Taskbezeichner des aktuell zugeordneten Tastaturmanagers.

sysout partner

PROC sysout partner

Wirkung: Liefert den internen Taskbezeichner des aktuell zugeordneten Bildschirmmanagers.

x size

INT PROC x size

Wirkung: Liefert die Anzahl der Spalten des zugeordneten Bildschirms.

y size

INT PROC y size

Wirkung: Liefert die Anzahl der Zeilen des zugeordneten Bildschirms.

6.6 Sonstige

copy to screen

PROC copy to screen (INT CONST x, y, attr, TEXT CONST string)

Wirkung: Der Text 'string' wird an der Bildschirmposition 'x', 'y' in der Farbe 'attr' ausgegeben. Die Prozedur schreibt direkt in den Bildschirmspeicher der Task. Es findet keine explizite Überprüfung auf Einhaltung der Bildschirmgrenzen statt. Die Prozedur fuktioniert nur im Alpha-Modus.

in key

INT PROC in key (INT CONST limit)

Wirkung: Es wird 'limit' 1/1000 Sekunden gewartet, ob eine Eingabe anliegt. Falls ja wird diese Eingabe geliefert, falls innerhalb 'limit' keine Eingabe kam, wird `-1` geliefert. Das unterste Byte enthält den Ascii Code der Taste, das zweitunterste den Scan Code. Falls die Task nicht angekoppelt ist, (`sysin partner = niltask`) wird gewartet, bis die Task wieder angekoppelt ist.

in scan ascii

INT PROC in scan ascii(INT CONST limit)

Wirkung: Es wird 'limit' 1/1000 Sekunden gewartet, ob eine Eingabe anliegt. Falls ja wird diese Eingabe geliefert, falls innerhalb 'limit' keine Eingabe kam, wird `-1` geliefert. Das unterste Byte enthält den Ascii Code der Taste, das zweitunterste den Scan Code.

Bemerkung: Falls die Task nicht angekoppelt ist, (`sysin partner = niltask`) wird `-1` geliefert. Falls die Task abgekoppelt wird, während sie auf Eingabe wartet, wird sofort `-1` geliefert.

screen write access

PROC screen write access

Wirkung: Der Bildschirmmanager erhält die Mitteilung »es wurde geändert«, der Bildschirminhalt der Task wird ohne die sonst übliche Analyse von Änderungen ausgegeben.

terminal necessary

PROC terminal necessary (BOOL CONST flag)

Wirkung: Dem I/O System wird mitgeteilt, ob Ausgaben auf ein Terminal warten sollen, oder ob ohne Rücksicht auf ein tatsächlich zugeordnetes Terminal in den Bildschirmspeicher geschrieben werden soll.

Bemerkung: Viele »DOS Programme« laufen in der Voreinstellung **terminal necessary (TRUE)** *nicht* ohne Terminal (»im Hintergrund«) weiter, obwohl nur »unwichtige« Ausgaben, etwa bei einem Suchlauf in einer Datenbank, ausgegeben werden. Durch Einstellung von **terminal necessary (FALSE)** kann erreicht werden, daß ein solches Programm auch nach Abkoppeln weiterläuft.

ACHTUNG: Bei dieser Einstellung droht **busy wait**. Viele DOS Programme warten z.B. in Endlosschleifen auf Eingaben. Wenn mehrere solcher Programme aktiv sind ist »der Rechner zu«.

BOOL PROC terminal necessary

Wirkung: Liefert die aktuelle Einstellung. Die Standardeinstellung ist **FALSE**.

6.7 Grafik

Die im weiteren beschriebene Behandlung des Bildschirms setzt auf eine »naturgetreue« Nachbildung der PC Hardware auf. Der Bildschirmspeicher ist ein besonderer Abschnitt des Arbeitsspeichers. Informationen, die in diesen Bereich geschrieben werden, werden ohne weiteres Zutun auf dem Bildschirm ausgegeben. Die Ausgabe kann in zwei Modi erfolgen. Im Textmodus wird der Inhalt des Bildschirmspeichers als Folge von Zeichen und Farbattributen behandelt. Im Graphikmodus wird der Bildschirmspeicher als zweidimensionales Feld von Bildpunkten (Pixel) angesehen. Die Darstellung dieser Punkte wird ggf. durch Schichten von Bildschirmspeicher-Ebenen (Planes) gesteuert. Weitere Details möge der geneigte Leser der gut und reichlich vorhandenen Literatur entnehmen.

Der Bildschirmspeicher hat also eine Anfangsadresse und eine bestimmte Größe. Beide Werte sind abhängig von der eingesetzten Hardware, bzw. vom eingestellten Bildschirmmodus. Desweiteren gehört zu einem Bildschirm noch die Form und Position des Cursors etc.

SCREEN	Datentyp Bildschirm
:=	Zuweisungsoperator
screen	Liefert Bildschirm
get screen	aktuellen Bildschirm in Datenraum
set screen	Bildschirminhalt aus Datenraum
print screen	SCREEN drucken
SCREENMODE	Datentyp Bildschirm-Einstellungen
:=	Zuweisungsoperator
=	Vergleichsoperator
screen mode	Liefert Einstellung
alpha 80	Vordefinierte Einstellung
alpha 40	— " —
alpha 80 hr	— " —
cga mono	— " —
cga colour	— " —
ega	— " —
vga	— " —
mcga	— " —
char height	Liefert Wert aus SCREENMODE
char width	— " —
memory base	— " —
memory size	— " —
mode	— " —
page size	— " —
screen height	— " —
screen width	— " —
screen page	Bildschirmseite einstellen

SCREEN

TYP SCREEN

Wirkung: Datentyp Bildschirm. Der Datentyp ist ein BOUND-Typ und muß mit einem Datenraum verknüpft werden. Ein Objekt vom Type Bildschirm ist beschrieben durch:

- Attribute
- Cursorposition x-Wert, y Wert
- Cursorform
- Cursormode
- Bildschirmmodus
- Anfangsadresse des Bildschirmspeichers
- Größe einer Seite im Bildschirmspeicher
- Größe des Bildschirmspeichers

Bemerkung: Wenn im folgenden Text von »Bildschirm« oder »Bildschirmspeicher« die Rede ist, so sind damit alle diese Informationen gemeint.

:=

OP := (SCREEN VAR l, SCREEN CONST r)

Wirkung: Zuweisungsoperator für SCREEN.

screen

SCREEN PROC screen (DATASPACE CONST ds)

Wirkung: Assoziiert den Datenraum 'ds' mit einer SCREEN. Falls es sich um einen neuen Datenraum handelt (type (ds) < 0), wird SCREEN auf **alpha 80** initialisiert.

get screen

PROC get screen (SCREEN VAR scr)

Wirkung: Der aktuelle Inhalt des Bildschirmspeichers wird in 'scr' gespeichert.

set screen

PROC setscreen (SCREEN CONST scr ds)

Wirkung: Der in 'scr ds' gespeicherte Bildschirm wird ausgegeben.

print screen

PROC print screen (TEXT CONST sname)

Wirkung: Der Datenraum 'sname', der den Bildschirmspeicherinhalt enthält, wird gedruckt.

Beispiel:

```
PROC print sc :

  DATASPACE VAR ds := nilspace;;
  SCREEN VAR s := screen (ds);
  TEXT VAR sname :=  name (myself) + text( (cputime)) ;

  get screen (s);
  move (ds, sname) ;
  print screen (sname);
  delete (sname , quiet);
  delete (sname + ".screen", quiet);

END PROC print sc;
```

screen mode

SCREENMODE PROC screen mode (SCREEN CONST screen)

Wirkung: Liefert die Einstellungen von 'screen'.

get screen page

PROC get screen page (SCREEN CONST scr, INT VAR page base, size)

Wirkung: Liefert Basisadresse und Seitengröße.

get cursor pos

PROC get cursor pos (SCREEN CONST scr, INT VAR xpos, ypos)

Wirkung: Liefert Cursorkoordinaten.

get attribute

PROC get attribute (SCREEN CONST scr, INT VAR attr)

Wirkung: Liefert zuletzt eingestelltes Attribut.

SCREENMODE

TYP SCREENMODE

Wirkung: Datentyp Bildschirmmodus. Konstanten dieses Typs können die verschiedenen Bildschirmmodi aufnehmen.

:=

OP := (SCREENMODE VAR l, SCREENMODE CONST r)

Wirkung: Zuweisungsoperator für SCREENMODE.

=

OP := (SCREENMODE CONST l, r)

Wirkung: Vergleichsoperator für SCREENMODE.

screenmode

PROC screenmode (SCREENMODE CONST sc mode)

Wirkung: Versucht den angegebenen Bildschirmmodus zu setzen und schaltet auf die erste Bildschirmseite. Wird die Prozedur am Terminal aufgerufen, bleibt der alphanumerische Modus eingestellt.

Beispiel: `screenmode(vga)`

SCREENMODE PROC screenmode (INT CONST screen width, screen height, char width, char height, mode, memory base, memory size, page size)

Wirkung: Liefert SCREENMODEs.

Folgende SCREENMODE Prozeduren sind definiert:

alpha 80
alpha 40
alpha 80 hr
cga mono
cga colour
ega
vga
mcga

```
alpha 80   := screenmode : ( 80,  25, 1, 1,  0, 0xb8000,  0x8000, 0x1000)
alpha 40   := screenmode : ( 40,  25, 1, 1,  0, 0xb8000,  0x8000, 0x0800)
cga mono   := screenmode : (640, 200, 8, 8,  1, 0xb8000,  0x8000, 0x4000)
cga colour := screenmode : (320, 200, 8, 8,  2, 0xb8000,  0x8000, 0x4000)
ega        := screenmode : (640, 350, 8, 14, 4, 0xa0000, 0x10000,0x10000)
vga        := screenmode : (640, 480, 8, 16, 4, 0xa0000, 0x10000,0x10000)
mcga       := screenmode : (320, 200, 8, 8,  5, 0xa0000, 0x10000,0x10000)

SCREENMODE PROC alpha 80 hr :

  INT CONST type := screen hardware;
  IF on (type, vga bit)
     THEN screenmode : ( 80, 50, 1, 1, 0, 0xb8000, 0x8000, 0x2000)
     ELIF on (type, ega bit)
          THEN screenmode : ( 80, 43, 1, 1, 0, 0xb8000, 0x8000, 0x2000)
          ELSE alpha 80
  FI

END PROC alpha 80 hr;
```

Die einzelnen Werte einer SCREENMODE Konstanten können durch folgende Prozeduren ausgelesen werden:

char height

INT PROC char height (SCREENMODE CONST sm)

Wirkung: Liefert die Höhe eines Zeichens im Bildschirmmodus 'sm'. In den Zeichenmodi ist die Höhe stets 1.

char width

Wirkung: Liefert die Breite eines Zeichens im Bildschirmmodus 'sm'. In den Zeichenmodi ist die Höhe stets 1.

memory base

INT PROC memory base (SCREENMODE CONST sm)

Wirkung: Liefert die Startadresse des Bildschirmspeichers im Modus 'sm'.

memory size

INT PROC memory size (SCREENMODE CONST sm)

Wirkung: Liefert die Größe des Bildschirmspeichers im Modus 'sm'.

mode

INT PROC mode (SCREENMODE CONST sm)

Wirkung: Liefert den Typ des Bildschirmspeichers. (siehe auch 13.5).

screen height

INT PROC screen height (SCREENMODE CONST sm)

Wirkung: Liefert die Höhe des Bildschirmspeichers (= Zeilen im Textmodus, sonst Pixel).

screen width

INT PROC screen width (SCREENMODE CONST sm)

Wirkung: Liefert die Breite des Bildschirmspeichers (= Spalten im Textmodus, sonst Pixel).

page size

INT PROC page size (SCREENMODE CONST sm)

Wirkung: Liefert die Größe einer Bildschirmseite.

screen page

PROC screen page (INT CONST page no)

Wirkung: Wechselt auf die angegebene Bildschirmseite. Die erste Seite hat die Nummer 0. Falls ein nicht einstellbarer Wert angefordert wird, wird der nächstmögliche eingestellt.

7 Intertask Kommunikation

7.1 Einführung

Jeglicher Informationsaustausch in einem L3 System erfolgt durch Versand einer Botschaft von einer Task an eine andere. Eine Task ist der Sender, die andere der Empfänger einer Botschaft. Prinzipiell kann jede Task irgendwelche Botschaften an irgendeine andere senden. Eine Botschaft wird jedoch vom avisierten Empfänger nur angenommen, wenn der Empfänger seine Empfangsbereitschaft hergestellt hat. Die Kommunikation ist synchron. Eine Task kann empfangsbereit für Sendungen einer bestimmten Task sein (**geschlossenes Warten**) oder auch empfangsbereit für Sendungen jeder beliebigen Task (**offenes Warten**). Der Sende-/Empfangsvorgang ist atomar. Es gibt nur die Zeit vor der Übermittlung einer Botschaft und die Zeit nach der Übermittlung. Der Fall, daß eine Botschaft »teilweise empfangen« wurde, ist nicht möglich. Die eigentliche Übermittlung einer Botschaft ist logisch zeitlos.

Tatsächlich kann die Übermittlung sehr wohl einige Zeit benötigen, z.B. wenn mehrere lange Bytestrings übertragen werden. Es ist sogar der Fall denkbar, daß der Rechner ordnungsgemäß abgeschaltet wird und eine Botschaftübermittlung faktisch vor dem Abschalten und dem damit verbundenen Fixpunkt begann und nach dem Wiedereinschalten beendet wird. Trotzdem garantiert der Systemkern für die beiden beteiligten Tasks die Atomizität des Botschaftstransfers.

Die Objekte der Kommunikation, die Botschaften, sind strukturierte Datenobjekte. Die Struktur und der Inhalt einer Botschaft sind im Botschaftsvektor der Task beschrieben. Der Botschaftsvektor, der ein Teil des Prozeßkontrollblocks einer jeden Task ist, ist ein String von 32 Doppelworten. In diesen String werden in beliebiger Mischung und Anordnung Einträge für Botschaftsobjekte der Typen INT, Bytesequenz (z.B. TEXT) oder DATASPACE gemacht. Ein Eintrag für INT und DATASPACE ist ein Doppelwort, ein Bytesequenz-Eintrag zwei Doppelworte lang. Eine Botschaft kann einen bis sechzehn Integer, einen bis sechzehn Datenräume, einen bis sieben Texte oder beliebige Mischungen daraus bis zur maximalen Eintragslänge von sechzehn Doppelworten aufnehmen.

Der Aufbau einer Botschaft wird vom Absender durch Initialisierung des Botschaftsvektors mit der Prozedur **new msg** eingeleitet. **put msg (param)** »füllt« den Botschaftsvektor mit Objekten des definierten Typs. Die fertige Botschaft wird dann durch **send** oder **call** an den Empfänger geschickt.

Der Empfänger initialisiert ebenfalls seinen Botschaftsvektor (**new msg**) und spezifiziert durch **invite msg (param)** den Aufbau der erwarteten Botschaft. Anschließend geht die empfangswillige Task in geschlossenes Warten (**receive**) oder offenes Warten (**wait**) über. Nach Empfang der Botschaft wird der Botschaftsvektor durch **get msg (param)** ausgewertet.

Beispiel:

```
(* TASK sender *)

(*1*)   INT VAR result, integer := 1234 ;
(*2*)   TEXT VAR string := "Hallo Empfänger!" ;
(*3*)   DATASPACE VAR space := dataspace (std);
```

```
(*4*)     new msg;
(*5*)     put msg (integer);
(*6*)     put msg (string);
(*7*)     put msg (space);
(*8*)         send (/"empfänger",never,result);

(*9*)     IF result = communication ok
             THEN putline ("O.K.")
             ELSE putline ("K.O.")
          FI;



(* TASK empfänger *)

(*10*)   INT VAR my int, maxlength := 40,
                 result;

         TEXT VAR textvar := "";
         DATASPACE VAR rec ds;

(*11*)   new msg;
(*12*)   invite msg (int msg);
(*13*)   invite msg (string msg, textvar, maxlength);
(*14*)   invite msg (int msg);
(*15*)   invite msg (ds msg );
(*16*)      receive (/"sender",never,result);
(*17*)   get msg (my int);
(*18*)   get msg (textvar);
(*19*)   get msg (rec ds);

(*20*)   FILE VAR f := sequential file (output, logfile);
         putline (f, (text (my int)));
         putline (f, (textvar));
         putline (f, (text (index (rec ds))));

  (* So kann der Inhalt des empfangenen Datenraums *)
  (* in eine Textdatei kopiert werden: *)

         copy (rec ds, "neue datei um " + time) ;
```

Dieses Beispiel skizziert den Ablauf einer einmaligen Botschaftsübermittlung zwischen einem Sender und einem Empfänger. In der sendenden Task wird zunächst der Inhalt der Botschaft festgelegt. Es sollen die Zahl '1234', der Text 'Hallo Empfänger!' und ein Datenraum, der die zuletzt benutzte Datei enthält (wird durch die PROC **std** geliefert) an die Task 'empfänger' geschickt werden.

Diese in Zeile 1 bis 3 deklarierten Objekte werden nach Initialisierung des Botschaftsvektors

(Zeile 4) durch **put msg** in den Botschaftsvektor eingetragen (Zeile 5 bis 7). Die eigentliche Übertragung (Zeile 8) erfolgt mit der Option **never** (ggf. beliebig lange auf Empfangsbereitschaft des Empfängers warten) für das Sende- Timeout.

Anschließend wird das vom Systemkern in 'result' gelieferte Ergebnis der Operation ausgewertet.

Der Empfänger legt ebenfalls Objekte für den Empfang der erwarteten Botschaft fest (10), initialisiert seinen Botschaftsvektor (11) und bestimmt die Interpretation der erwarteten Objekte (**invite msg (..)**).

Hinweis: da Texte als Bytestrings verschickt werden (Zeile 14), muß die empfangende Task zwei Abschnitte des Botschaftsvektors freihalten. Siehe dazu den folgenden Abschnitt über den Aufbau des Botschaftsvektors.

Anschließend geht die empfangende Task in den Zustand »geschlossenes Warten« (Botschaften werden nur von 'sender' akzeptiert (Zeile 16)). Nach Empfang einer Botschaft wird der Inhalt des Botschaftsvektors in die entsprechenden Variablen übernommen (Zeile 17 bis 19) und in einer Datei protokolliert (Zeile 20).

Die wesentliche Erkenntnis an dieser Stelle soll sein, daß zwischen Sender und Empfänger eine Absprache über den Aufbau der gesendeten und erwarteten Botschaft existieren muß, da durch nachträgliche Inspektion der Inhalt einer empfangenen Botschaft nicht sicher entschlüsselt werden kann.

Solche Absprachen sind für wichtige Bereiche der Intertask Kommunikation bereits getroffen. Ihre Beschreibung ist Inhalt von Kapitel 13 ff.: Protokolle.

Achtung: Über die Intertask Kommunikation wird nicht nur die von Ihnen programmierte Kommunikation abgewickelt, sondern zum Beispiel auch die normale Ein/Ausgabe.

JEDE EINGABE/AUSGABE OPERATION BENUTZT DEN BOTSCHAFTSVEKTOR.

Der Botschaftsvektor wird also von Testausgaben gegebenenfalls zerstört !

Würde also im vorangegangenen Beispiel zwischen Zeile 16 und Zeile 17 das Kommando `putline ("Die Nachricht ist angekommen")` eingefügt, wäre der Botschaftsvektor der empfangenden Task zerstört und sie könnte die für sie bestimmte Nachricht nicht mehr korrekt auswerten.

Das Programm

```
new msg;
put msg (int msg);
put msg (ds msg)
    put ("HALLO");
...
```

ist folgendermaßen zu lesen:

```
initialisiere den Botschaftsvektor;
erster Abschnitt wird Integer;
zweiter Abschnitt wird Datenraum;
   initialisiere den Botschaftsvektor; (* !!!!! *)
```

```
      erster Abschnitt wird Integer ; (*'Gib-aus Code'*)
      zweiter Abschnitt wird Text   ; (* 'HALLO' *)
          call (/"my display", never, never, result);
   ....
```

7.2 Der Aufbau des Botschaftsvektors

Der Botschaftsvektor ist ein 128 Byte langer Abschnitt im Prozeßkontrollblock einer Task.

```
MDV  RDV  B1  B2 .... B15  B16  E1  E2 .... E13  E14
```

Die Doppelworte 'MDV' und 'RDV' beschreiben die Typen der folgenden Einträge, B1 bis B16 sind Botschaftsabschnitte, E1 bis E14 Eigenadressen.

MDV: Message Dope Vector. Der Message Dope Vektor beschreibt die Typen der zu sendenden Datenobjekte in den Botschaftsabschnitten B1 bis B16. Der Inhalt wird implizit durch die `put msg` Operationen beim Aufbau der Sendung erzeugt. Für jeden Abschnitt B1 bis B16 wird in zwei Bit der Typ festgehalten. MDV wird durch `new msg` gelöscht.

RDV: Receive Dope Vector. Der Receive Dope Vektor beschreibt die Typen der zu empfangenden Datenobjekte in den Botschaftsabschnitten B1 bis B16. Der Inhalt wird implizit durch die **invite msg** Operationen beim Empfangsaufbau erzeugt. Für jeden Abschnitt B1 bis B16 wird in zwei Bit der Typ festgehalten. RDV wird durch `new msg` gelöscht.

B1 bis B16: Botschaftsabschnitte. Die Botschaftsabschnitte enthalten die eigentlichen Botschaften in Form von Integer Einträgen. Die Interpretation dieser Einträge unterliegt folgender Systematik:

Sender vor der Sendung: int msg: der Abschnitt `Bx` enthält die Zahl.

ds msg: der Abschnitt `Bx` enthält den internen Datenraumbezeichner.

string msg: der Abschnitt `Bx+1` enthält die Startadresse des Bytestring in der Task. Der Abschnitt `Bx` enthält die Stringlänge.

Nach erfolgreicher Sendung ist ein gesendeter Datenraum gelöscht. INTs oder Strings werden als Kopien versendet.

Empfänger nach der Sendung: int msg: der Abschnitt `Bx` enthält die Zahl.

ds msg: der Abschnitt `Bx` enthält den internen Datenraumbezeichner des empfangenen Datenraums.

string msg: der Abschnitt `Bx` enthält die Stringlänge des Bytestring. Der Abschnitt `Ex+1` enthält die Adresse.

Da der Empfänger im Falle der `string msg` eine Fremdadresse erhält, muß noch die Angabe eines geeigneten Platzes im eigenen Adreßraum folgen. Nur zu diesem Zweck werden die Eigenadressen E1 bis E14 benötigt.

E1 bis E14: Eigenadressen In diesem Abschnitt werden für `string msg` die Adresse und die akzeptierte Länge für den jeweiligen Bytestring im eigenen Adreßraum festgehalten. Die Einträge werden z.B. durch `PROC invite msg (INT CONST typ, TEXT VAR string, INT CONST max length)` implizit vorgenommen.

Dieser Abschnitt des Botschaftsvektors wird also nur benutzt, wenn Bytestrings empfangen werden sollen.

7.3 Prozeduren und Konstanten

Die ELAN Schnittstelle zur Intertask-Kommunikation umfaßt neben den Prozeduren zum Aufbau von Botschaften und den Sende-/Empfangsoperationen auch noch eine Reihe vom Konstanten zur Auswertung von Botschaften und Antworten.

Prozeduren

new msg	Initialisierung des Botschaftsvektors
put msg	Botschaftsabschnitt füllen
invite msg	Empfangstyp auf Abschnitt aufprägen
get msg	Botschaftsabschnitt entnehmen
msg type	Abschnittstyp liefern
pop msg	Botschaftsvektor sichern
push msg	Botschaftsvektor entsichern
delete msg	Botschaftsvektor löschen
call	Botschaft senden; Warten auf Antwort
receive	(geschlossenes) Warten auf Botschaft
send	Botschaft senden
wait	(offenes) Warten auf Botschaft

Konstanten

communication ok	0
task not existing	1
send timeout	2
receive timeout	3
int msg	
string msg	
ds msg	
nil msg	
never	

7.3.1 Der Aufbau von Botschaften

new msg

PROC new msg

Wirkung: Initialisierung des Botschaftsvektors der Task, um den Kommunikationsaufbau zu beginnen.

Bemerkung: Der (interne) Zähler für die Einhaltung der maximal zulässigen Botschaftsvektorlänge, der durch *put, get, invite* hochgezählt wird, wird auf 0 zurückgesetzt. Ein Botschaftsvektor kann nicht teilweise neuaufgebaut werden, wenn ein Fehler aufgetreten ist.

Botschaftsabschnitte und Eigenadressen werden *nicht* gelöscht. Bei Analyse des Botschaftsvektors können also durchaus noch alte Inhalte in diesen Abschnitten enthalten sein.

put msg

PROC put msg (INT CONST int)

Wirkung: In den nächsten Abschnitt des Botschaftsvektors wird der INTEGER Wert 'int' eingetragen.

PROC put msg (DINT CONST int)

Wirkung: In die nächsten beiden Abschnitte des Botschaftsvektors wird der DINT Wert 'dint' als *int msg, int msg* eingetragen.

PROC put msg (INT CONST bool)

Wirkung: In den nächsten Abschnitt des Botschaftsvektors wird der BOOLsche Wert 'bool' als int msg eingetragen.

PROC put msg (DATASPACE VAR ds)

Wirkung: In den nächsten Abschnitt des Botschaftsvektors wird der Wert des Datenraumbezeichners 'ds' eingetragen. 'ds' selbst wird ungültig. Nach Ausführung der **send** bzw. **call** Operation ist der Datenraum gelöscht. Falls eine Kopie des Datenraumobjekts 'ds' an die empfangende Task geschickt werden soll, muß vor **send** oder **call** eine Kopie, die in der sendenden Task verbleibt, angelegt werden. Da der Datenraumbezeichner schon beim Eintrag in den Botschaftsvektor gelöscht wird, kann der Datenraum über den Bezeichner anschließend nicht mehr erreicht werden. Falls die send/call Operation nicht durchgeführt werden kann, sollte der Bezeichner wieder aus dem Botschaftsvektor mit **get msg** in den Bezeichner zurückgelesen werden, damit der Datenraum nicht als »Leiche« in der Task verbleibt.

PROC put msg (TEXT CONST string)

Wirkung: In die nächsten beiden Abschnitte des Botschaftsvektors wird der Text 'string' als Typ **string msg** eingetragen. Im Botschaftsvektor stehen aktuelle Länge und Adresse von 'string'.

PROC put msg (INT CONST stringaddr, stringlength)

Wirkung: In die nächsten beiden Abschnitte des Botschaftsvektors wird der Wert 'stringlength' und die Adresse 'string addr' als **string msg** eingetragen. Damit lassen sich beliebige Bytesequenzen übertragen.

Bemerkung: Eine Bytefolge, die übertragen werden soll, muß nicht unbedingt im Standarddatenraum stehen, sondern kann aus einem beliebigen Datenraum der Task sein.

invite msg

PROC invite msg (INT CONST typ)

Wirkung: Der nächste Botschaftsabschnitt, der erwartet wird, soll vom Typ 'typ' sein. Zulässige Typen sind **int msg**, **ds msg** oder **int msg OR ds msg**.

Bemerkung: Die leere Botschaft **nil msg** darf nicht als Botschaft erwartet werden. Der Empfang eines Botschaftsabschnitts von Typ **string msg** bedarf weiterer Parameter, da nicht feststeht, wohin die Bytesequenz geschrieben werden soll und wieviel Platz sie benötigen wird. Die grundsätzliche Bereitstellung von **maxtextlength** Bytes würde zu starken Ineffizienzen führen.

PROC invite msg (INT CONST typ, TEXT VAR string, INT CONST max length)

Wirkung: Der nächste Botschaftsabschnitt, der erwartet wird, soll vom Typ 'typ' sein. Zulässig sind alle Typen und Kombinatioenen.

Bemerkung: Da der interne Zähler für den Botschaftsvektor durch **invite msg** nur um 1 erhöht wird, muß der »Einladung für einen Text« ein **invite msg (int msg)** folgen, um im Botschaftsvektor die benötigten zwei Doppelworte Platz zu schaffen. Die maximale Länge einer Bytefolge, die in diesem Botschaftsabschnitt empfangen werden kann, ist durch die angegebene 'max length' bestimmt.

PROC invite msg (INT CONST typ, stringaddr, max length)

Wirkung: Der nächste Botschaftsabschnitt, der erwartet wird, soll vom Typ 'typ' sein. Zulässige Typen sind alle Typen und Kombinationen. Bei Empfang eines Botschaftsabschnitts vom Typ **string msg** wird dieser an der explizit angegebenen 'stringaddr' mit der Länge 'max length' geschrieben.

Bemerkung: Da der interne Zähler für den Botschaftsvektor durch **invite msg** nur um 1 erhöht wird, muß der »Einladung für einen Text« ein **invite msg (int msg)** folgen, um im Botschaftsvektor die benötigten zwei Doppelworte Platz zu schaffen. Die maximale Länge einer Bytefolge, die in diesem Botschaftsabschnitt empfangen werden kann, ist durch die vorgegebene 'max length' bestimmt.

ACHTUNG: 'stringaddr' muß im **Standarddatenraum** der Task liegen.

get msg

PROC get msg (INT VAR int)

Wirkung: Der nächste Botschaftsabschnitt wird in die INTEGER Variable 'int' kopiert.

PROC get msg (DINT VAR dint)

Wirkung: Die nächsten beiden Abschnitte werden in die DINT Variable 'dint' kopiert. Die beiden Botschaftsabschnitte müssen als **int msg** erwartet worden sein.

PROC get msg (BOOL VAR bool)

Wirkung: Der nächste Botschaftsabschnitt wird in die BOOLsche Variable 'bool' kopiert. Er muß als **int msg** erwartet worden sein.

PROC get msg (DATASPACE VAR ds)

Wirkung: Der nächste Botschaftsabschnitt wird in die DATASPACE Variable 'ds' kopiert. Er muß als **ds msg** erwartet worden sein.

Bemerkung: Falls das Senden eines Datenraums nicht erfolgreich war, muß mit dieser Prozedur wieder in die Datenraumvariable zurückgeschrieben werden.

PROC get msg (TEXT VAR string)

Wirkung: Die nächsten beiden Botschaftsabschnitte werden benutzt, um die Bytefolge, deren Adresse und Länge dort enthalten ist, in die Textvariable 'string' zu kopieren.

Bemerkung: Falls diesem Aufruf von **get msg** ein Aufruf von **invite msg** mit derselben Textvariablen 'string' voranging, muß der Text nicht noch einmal kopiert werden.

Fehler: Falls sehr lange Strings empfangen werden sollen, muß als Empfangspuffer die Zielvariable selbst verwendet werden:

```
TEXT VAR buffer ;
new msg;
invite msg (buffer, 32000);
wait(...);
get msg (buffer);
```

Falls der empfangene String in einer anderen Variablen benötigt wird, muß anschließend kopiert werden.

Folgendes Vorgehen bietet nämlich eine wunderliche Fehlerquelle:

```
TEXT VAR buffer, tvar ;
new msg;
invite msg (buffer , 32000);
wait (...);
get msg (tvar)   (* sehr fehleranfällig !! *)
```

Es *kann* bei diesem Vorgehen vorkommen, daß der Empfang in der Textvariablen 'tvar' eine Müllabfuhr auf dem Textheap des Standarddatenraums auslöst, die die Quelle 'buffer' verschiebt. In diesem Falle wird der String von einer falschen Anfangsadresse an kopiert.

Eine andere Möglichkeit, diese Fehlerquelle auszuschließen, besteht darin, als Empfangspuffer eine `ROW 32000 BYTE` zu definieren.

PROC get msg (INT VAR stringaddr, stringlength)

Wirkung: Die nächsten beiden Botschaftsabschnitte werden als Bytefolge interpretiert. Die Bytefolge wird dann ab der Adresse 'stringaddr' in der Länge 'stringlength' in den Standarddatenraum kopiert.

Beispiel:

```
(************************************************)
(* Dieses Programm in Task "empfänger" starten  *)
(************************************************)

TASK VAR sender ;
LET RBUFFER = ROW 100 BYTE ;
RBUFFER VAR bufa, bufb, bufc;

INT VAR res, intvar1, intvar2 ,
        in addr := ADDR bufa(1) ,
        in length := 100 ;
           (* Hier kann man sich selbst umzingeln *)
           (* Falls der Wert > Puffer ist, wird *)
           (* hinter dem Puffer überschrieben *)
new msg;
invite msg (  string msg , in addr      , in length);
           (* bytestring , Adr im Stdds , soviel darfs sein *)

wait (50000 , sender , res);

get msg (intvar1, intvar2) ;

(***********************************************)
(* Anschließend dieses in der sendenden Task   *)
(***********************************************)

TASK VAR partner := task ("empfänger") ;
INT VAR res , i ;
LET SBUFFER = ROW 100 BYTE ;
SBUFFER VAR sbuf1, sbuf2, sbuf3;

FOR i FROM 1 UPTO  100
```

```
REP
   sbuf1 [i] := byte (i )
PER;

new msg;
put msg (ADDR sbuf1 [1] , 300) ;
send (partner, 1000 , res);
IF res = communication ok THEN put ("OK") FI ;
```

Allgemein gilt: Falls bei einer **get msg** Operation keine Übereinstimmung des Typs des erwarteten und des gelieferten Abschnitts besteht, ist die Wirkung dieser und aller folgenden **get msg** Operationen undefiniert. Dieser Zustand wird erst durch Initialisierung des Botschaftsvektors mit **new msg** beendet.

msg type

INT PROC msg type

Wirkung: Liefert den Typ des nächsten zu empfangenden Botschaftsabschnitts.

7.3.2 Konstanten

Um bei Definition und Auswertung der erwarteten Botschaftsabschnitte mit sprechenden Namen arbeiten zu können, sind folgende Konstanten vordefiniert.

INT PROC ds msg

Wirkung: Liefert den Abschnittstyp Datenraum.

INT PROC int msg

Wirkung: Liefert den Abschnittstyp Integer.

INT PROC nil msg

Wirkung: Liefert den Abschnittstyp leere Botschaft.

INT PROC string msg

Wirkung: Liefert den Abschnittstyp Textstring.

INT PROC never

Wirkung: Liefert den Wert »kein Timeout« für Timeout Angaben bei Sende- und Empfangsoperationen.

Die Resultatkonstanten **communication ok, task not existing, send timeout** und **receive timeout** werden im Abschnitt über Sende- und Empfangsoperationen erläutert.

7.3.3 Zwischenspeichern

Um initialisierte Botschaftsvektoren vor Überschreiben zu schützen, können sie in einen Stack gebracht werden, der maximal 100 Vektoren aufnehmen kann. Dies kann insbesondere nötig werden, wenn vor dem Absenden oder Auswerten des Botschaftsvektors irgendwelche Ausgaben gemacht werden sollen.

pop msg

PROC pop msg

Wirkung: Das oberste Element des Stacks wird zum aktuellen Botschaftsvektor.

Fehler: "Unterlauf des Botschafts-Stack" Es ist kein Botschaftsvektor auf dem Stack.

push msg

PROC push msg

Wirkung: Der aktuelle Botschaftsvektor wird auf dem Stack abgelegt.

Fehler: "Überlauf des Botschafts-Stack" Der Stack ist bereits gefüllt. Er enthält 100 Botschaftsvektoren.

delete msg

PROC delete msg

Wirkung: Löscht das oberste Element im Stack.

Fehler: "Unterlauf des Botschafts-Stack" Es ist kein Botschaftsvektor auf dem Stack.

7.3.4 Senden und Empfangen

Der Botschaftstransfer, die Übermittlung des Inhalts des Botschaftsvektors der sendenden Task zum Botschaftsvektor der empfangenden Task kann stattfinden, wenn der Empfänger seine Empfangsbereitschaft hergestellt hat. Der Empfänger stellt durch **wait** oder **receive** seine Empfangsbereitschaft her und eine vom Sender durch **send** oder **call** abgeschickte Botschaft wird empfangen. Da die Kommunikation Unwägbarkeiten unterworfen ist (z.B. kann eine Task bereits von einem anderen Kommunikationspartner vereinnahmt sein), enthält jede der Prozeduren eine Variable, in der vom System das erzielte Ergebnis geliefert wird.

Alle Operationen können durch ein **Timeout** limitiert werden. Wenn nach Ablauf dieser in Millisekunden angegebenen Zeitspanne der avisierte Partner nicht entsprechend reagiert hat, so wird die Operation mit einer entsprechenden Meldung beendet.

Soll kein Abbruch durch Zeitüberschreitung möglich sein, muß **never** als Timeout gewählt werden.

Falls bei einer **send** oder **call** Operation, in der ein oder mehrere Datenräume verschickt werden sollten, ein **send timeout** auftrat, sollten die Datenraumbezeichner mit **get msg** aus dem

Botschaftsvektor geholt und ggf. gelöscht werden. Andernfalls bliebe eine *Datenraumleiche* in der Task zurück.

call

PROC call (TASK CONST partner, INT CONST send timeout, receive timeout, INT VAR result)

Wirkung: Die vorher zusammengestellte Botschaft wird an die Task 'partner' geschickt und eine Botschaft von 'partner' erwartet. Die Prozedur hat somit zwei Phasen, die jedoch untrennbar miteinander verbunden sind. Die erfolgreiche Durchführung von **call** umfaßt die Sendung der Botschaft und den Empfang der (Quittungs) Botschaft von der Zieltask. Die maximale Wartezeit auf die Empfangsbereitschaft von 'partner' wird durch 'send timeout' bestimmt. Wenn diese Zeit abgelaufen ist, ohne daß die Zieltask empfangsbereit war, wird keine Botschaft geschickt und keine Botschaft empfangen. Die maximale Wartezeit auf eine Antwort wird durch 'receive timeout' vorgegeben. Wenn diese Zeit abgelaufen ist, wurde die Botschaft geschickt, jedoch keine Botschaft empfangen.

Bemerkung: Bei erfolgreicher Durchführung liefert **send (..) ; receive (..)** ein identisches Resultat. Es ist jedoch der Fall vorstellbar, daß die Zieltask die Antwort schickt, bevor die eigene Empfangsbereitschaft hergestellt werden kann. In diesem Fall würde eine im Grunde korrekt verlaufene Kommunikation nicht erkannt.

Parameter: 'result' liefert das Ergebnis der **call** Operation.

communication ok: eine Botschaft wurde geschickt und von 'partner' empfangen. Die Antwort von 'partner' wurde empfangen.

task not existing: die Task 'partner' gibt es nicht. Es wurde keine Botschaft geschickt und keine Antwort empfangen.

send timeout: bis zum Ablauf von **send timeout** war die Task 'partner' nicht empfangsbereit. Es wurde keine Botschaft geschickt und keine Antwort empfangen.

receive timeout: bis zum Ablauf von **receive timeout** wurde keine Antwort von 'partner' empfangen. Dieses Ergebnis wird auch geliefert, wenn 'partner' gelöscht wird, bevor sie die Antwort zurückschicken kann.

Beispiel:

```
PROC send one receive one (TASK CONST partner,
               INT CONST code, DATASPACE VAR ds,
               INT VAR reply, result):
     disable stop;
     new msg;
     put msg (code);   (* Sendung aufbauen *)
     put msg (ds);
     invite msg (int msg); (* Antwort vorbereiten *)
     invite msg (ds msg);
        call (partner, never, never, result);
        IF result = communication ok
           THEN get msg (reply);
```

```
                get msg (ds)
            FI
        END PROC send one receive one;
```

receive

PROC receive (TASK CONST sender, INT CONST receive timeout, INT VAR result)

Wirkung: Geschlossenes Warten. Nach Definition der erwarteten Botschaft wartet die Task auf eine Botschaft der Task 'sender'. Für andere Tasks ist sie nicht empfangsbereit.

Parameter: 'result' liefert das Ergebnis der **receive** Operation:

communication ok: eine Botschaft wurde von 'sender' empfangen.

task not existing: die Task 'sender' gibt es nicht.

receive timeout: bis zum Ablauf von **receive timeout** wurde keine Botschaft von 'sender' empfangen oder die Task wurde gelöscht.

Bemerkung: `receive (niltask, timeout, result)` ist erlaubt.

send

PROC send (TASK CONST partner, INT CONST send timeout, INT VAR result)

Wirkung: Die fertige Botschaft wird an die Task 'partner' geschickt, sofern 'partner' existiert und empfangsbereit ist.

Parameter: 'result' liefert das Ergebnis der **send** Operation:

communication ok: eine Botschaft wurde geschickt und von 'partner' empfangen.

task not existing: die Task 'partner' gibt es nicht. Es wurde keine Botschaft geschickt.

send timeout: bis zum Ablauf von 'send timeout' war die Task 'partner' nicht empfangsbereit. Es wurde keine Botschaft geschickt.

wait

PROC wait (INT CONST receive timeout, TASK VAR sender, INT VAR result)

Wirkung: Offenes Warten. Nach Definition der erwarteten Botschaft wartet die Task auf eine Botschaft von irgendeiner Task.

Parameter: 'result' liefert das Ergebnis der **wait** Operation:

communication ok: eine Botschaft wurde korrekt empfangen.

receive timeout: bis zum Ablauf von 'receive timeout' wurde keine Botschaft von 'sender' empfangen.

8 Managertasks

8.1 Einführung

Ein Manager ist eine Task, die irgendwelche Objekte verwaltet und Aufträge bearbeitet, die sich auf diese Objekte beziehen. Eine Form des Managing findet in jeder gewöhnlichen Task statt. Datenräume, häufig von der Art »Textdatei«, werden angelegt und, gegebenenfalls in Verzeichnissen geordnet, verwaltet. Diese innere Ordnung von Datenräumen in einer Task mit der Möglichkeit Pfade über Verzeichnisse zu spezifizieren und Mengen von Namen (Thesauri) zusammenzustellen, ist der lokale Aspekt des Managing.

Eine Task kann aber nicht nur intern, sondern auch für andere Tasks Ressourcen verwalten. In diesem Falle werden passende Aufträge via Intertask Kommunikation von anderen Tasks erwartet.

Die Aufträge beziehen sich auf irgendwelche Ressourcen, die im Besitz dieses Managers sind. Standardbeispiel ist ein Dateimanager, eine Task, die Dateien besitzt und nach Auftrag Dateien an andere Tasks sendet oder auch Dateien, die von anderen Tasks geschickt werden, in ihren Besitzstand aufnimmt. Ein anderes Beispiel ist ein Spoolmanager, der nur Dateien empfängt, möglicherweise in eine Warteschlange reiht und schlußendlich an den Drucker leitet.

In beiden Fällen ist eine Managertask programmiertechnisch als Task zu verstehen, die in einer Endlosschleife auf eine bestimmte Klasse von Aufträgen wartet. Die Realisierung als ELAN Programm besteht in jedem Fall aus einer Rahmenprozedur **manager**, die die Endlosschleife realisiert und nach Eingang eines Auftrags (und Analyse dieses Auftrags) eine Dienstprozedur aufruft, die den Auftrag bearbeitet. Die Rahmenprozedur hat also die Gestalt:

```
PROC manager (PROC (INT CONST, TASK CONST) execute operation)
```

Die Dienstprozedur 'execute operation' kann, falls sie den vorliegenden Auftrag nicht bearbeiten kann, eine andere Dienstprozedur aufrufen. Die Hierarchie der Dienstprozeduren ist in Kap.8.5 beschrieben. Neben dem eigentlichen Auftragscode, der `INT CONST` in der obigen Prozedur, gehören ggf. weitere Parameter zum Auftrag an die Managertask. Die grundsätzlich für Dateimanager zulässigen Aufträge und ihre Parametrisierung sind zusammenfassend in Kap.13 »Das Dateimanagerprotokoll« beschrieben.

Der Aufbau der folgenden Abschnitte ist folgender: zunächst sind die Basistypen für die Klassifikation und Zusammenfassung von Objekten beschrieben. Die Klassifikation erfolgt über den Datentyp **SHAPE**, der die Aufteilung von Objekten in die Klassen »Datenraum« und »Verzeichnis« zuläßt. Die Zusammenfassung von Objekten ist über Verzeichnisse, die eine baumartige Ordnung auf Objekten definieren oder über den Typ **THESAURUS**, der eine lineare Namensliste liefert, möglich.

Mit diesen Mitteln lassen sich dann die möglichen Aktionen auf Objekten definieren, zunächst sind die Aktionen mit Wirkung in einer Task beschrieben (**local manager**), dann von außen induzierte und nach außen wirkende Aktionen (**free manager**). Abschließend sind die beiden speziellen Managertypen **archive manager** und **streamer manager** beschrieben.

8.2 Thesaurus

Ein Thesaurus ist eine lineare Namensliste mit einer maximalen Länge von 29995 Zeichen. Jedem Namen in einem Thesaurus wird ein Index zugeordnet. Somit können Thesauri zur Verwaltung benannter Objekte verwandt werden. Der Zugriff auf ein Objekt ist dann über den Index seines Namens möglich.

Interessant ist die Möglichkeit Prozeduren mit TEXT Parameter durch das am Ende dieses Abschnitts beschrieben **do** auf ganze Namenslisten anzuwenden. So ist es möglich, eine Prozedur `myproc (filename)` auf alle Dateien einer gegebenen Namensliste anzuwenden, indem auch noch die Prozedur für den Parametertyp THESAURUS definiert wird.

Der Typ THESAURUS in seiner vorliegenden Form ist nicht vollständig an die Möglichkeiten von L3 angepaßt. Eine Task kann 16000 benannte Datenräume besitzen. Die Länge der Namensliste kann mit höchstens 29995 Zeichen zu gering sein, um solche Namenslisten vollständig zu verwalten.

8.2.1 Operationen auf Thesauri

Basisoperationen

THESAURUS	Datentyp
:=	Zuweisung
CONTAINS	Stellt fest, ob enthalten
delete	Löscht ein Element falls vorhanden
empty thesaurus	Initialisierungsobjekt
get	Liefert nächsten Eintrag
highest entry	Liefert letzten Eintrag
insert	Fügt ein Element ein
link	Index im thesaurus
name	Name zu Index
rename	Benennt ein Element um

Verknüpfungen

+	Vereinigungsmenge
-	Differenzmenge
/	Schnittmenge
ALL	Vollständiger Thesaurus
FILLBY	Thesaurus auffüllen
LIKE	Filter
SOME	Teilthesaurus bestimmen

Spezial

do	
remainder	Liefert Restthesaurus

THESAURUS

TYPE THESAURUS

Wirkung: Datentyp

:=

OP := (THESAURUS VAR dest, THESAURUS CONST source)

Wirkung: Zuweisung

CONTAINS

BOOL OP CONTAINS (THESAURUS CONST thes, TEXT CONST name)

Wirkung: Liefert TRUE, wenn das Objekt 'name' im Thesaurus 'thes' enthalten ist.

delete

PROC delete (THESAURUS VAR thes, TEXT CONST name, INT VAR index)

Wirkung: Falls der Name 'name' im Thesaurus 'thes' enthalten ist, wird er aus dem Thesaurus gelöscht. 'index' enthält dann den Index den dieser Eintrag im Thesaurus hatte. Falls der Name nicht in 'thes' enthalten ist, wird als Index `0` geliefert.

PROC delete (THESAURUS VAR thes, TEXT CONST name)

Wirkung: Falls der Name 'name' im Thesaurus 'thes' enthalten ist, wird er aus dem Thesaurus gelöscht.

PROC delete (THESAURUS VAR thes, INT CONST ix)

Wirkung: Löscht den 'ix'-ten Eintrag im Thesaurus 'thes'.

empty thesaurus

THESAURUS PROC empty thesaurus

Wirkung: Liefert zum Zwecke der Initialisierung einen leeren Thesaurus.

get

PROC get (THESAURUS CONST t, TEXT VAR name, INT VAR index)

Wirkung: Liefert den »nächsten« Eintrag aus dem Thesaurus 't'. »Nächster« heißt hier, der kleinste vorhandene mit einem Index größer als 'index'. Dabei wird in 'name' der Name und in 'index' der Index des Eintrags geliefert. D.h. 'index' wird automatisch weitergeschaltet. Den ersten Eintrag erhält man entsprechend durch Aufruf mit `index=0`. Nach dem letzten Eintrag wird `name=""` und `index=0` geliefert.

Beispiel:

```
TEXT VAR name;
INT VAR index := 0 ;
get (thesaurus, name, index) ;
WHILE index > 0 REP
    putline (name) ;
    get (thesaurus, name, index)
PER
```

highest entry

INT PROC highest entry (THESAURUS CONST t)

Wirkung: Liefert den höchsten belegten Index des Thesaurus 't'.

Bemerkung: Liefert nicht zwingend die Anzahl der vorhandenen Namen, da durch Löschungen Lücken entstanden sein können.

insert

PROC insert (THESAURUS VAR t, TEXT CONST name, INT VAR index)

Wirkung: Der Name 'name' wird als zusätzlicher Eintrag in den Thesaurus 't' eingetragen und der dafür vergebene Index geliefert. Falls der Thesaurus schon voll ist und der Name nicht mehr eingetragen werden kann, wird 0 als Index geliefert.

Bemerkung: Achtung: Mehrfacheintragungen sind möglich. Wenn man diese verhindern will, muß man entsprechend vermittels

```
IF NOT t CONTAINS name
    THEN insert (t, name, index)
FI
```

eintragen.

Fehler: Name unzulaessig

link

INT PROC link (THESAURUS CONST t, TEXT CONST name)

Wirkung: Liefert den Index des Namens 'name' im Thesaurus 't'. Falls der Name nicht enthalten ist, wird 0 geliefert. Ist der Name mehrfach im Thesaurus enthalten, ist nicht definiert, welcher der möglichen Indizes geliefert wird.

name

TEXT PROC name (THESAURUS CONST t, INT CONST index)

Wirkung: Liefert den Namen des Eintrags mit dem Index 'index' aus dem Thesaurus 't'. Falls kein solcher Eintrag im Thesaurus enthalten ist, wird niltext geliefert.

rename

PROC rename (THESAURUS VAR t, TEXT CONST old, new)

Wirkung: Ändert im Thesaurus 't' einen Eintrag mit dem alten Namen 'old' in 'new' um. Falls 'old' nicht im Thesaurus enthalten ist, wird keine Leistung erbracht. Falls 'old' mehrfach in 't' enthalten ist, ist nicht definiert, welcher der möglichen Einträge geändert wird.

Fehler: Name unzulaessig

PROC rename (THESAURUS VAR t, INT CONST index, TEXT CONST new)

Wirkung: Ändert in Thesaurus 't' den Namen des durch 'index' identifizierten Eintrags in 'new'.

Fehler: Name unzulaessig

8.2.2 Verknüpfungen

+

THESAURUS OP + (THESAURUS CONST left, right)

Wirkung: Liefert die Vereinigungsmenge von 'left' und 'right'. Achtung: Die Vereinigungsmenge enthält keine Namen mehrfach.

THESAURUS OP + (THESAURUS CONST left, TEXT CONST right)

Wirkung: Fügt dem Thesaurus 'right' zu, wenn 'right' noch nicht im Thesaurus enthalten ist.

-

THESAURUS OP - (THESAURUS CONST left, right)

Wirkung: Liefert die Differenzmenge. Achtung: Die Differenzmenge enthält keine Namen mehrfach.

THESAURUS OP - (THESAURUS CONST left, TEXT CONST right)

Wirkung: Nimmt den Namen 'right' aus dem Thesaurus.

/

THESAURUS OP / (THESAURUS CONST left, right)

Wirkung: Liefert die Schnittmenge Achtung: Die Schnittmenge enthält keine Namen mehrfach.

ALL

THESAURUS OP ALL (TASK CONST task)

Wirkung: Liefert einen Thesaurus, der alle Dateinamen der angegebenen Task enthält.

THESAURUS OP ALL (TEXT CONST object name)

Wirkung: Liefert einen Thesaurus, der die in der angegebenen Datei vorhandenen Namen (jede Zeile ein Name) enthält. `IF shape (object name) = dir THEN all (object name) ELIF shape (object name) = ds THEN result FILLBY objectname FI`

all

THESAURUS PROC all

Wirkung: Liefert einen Thesaurus, der alle Dateinamen der eigenen Task enthält. Entspricht 'ALL myself'.

THESAURUS PROC all (TEXT CONST dirname)

Wirkung: Liefert einen Thesaurus, der alle Namen von Datenräumen des Verzeichnisses 'dirname' enthält.

THESAURUS PROC all (TEXT CONST dirname, SHAPE CONST shape)

Wirkung: Liefert einen Thesaurus, der alle Namen von Objekten der Gestalt 'shape' des Verzeichnisses 'dirname' enthält.

THESAURUS PROC all (TASK CONST task, TEXT CONST dirname)

Wirkung: Liefert einen Thesaurus, der alle Namen von Objekten des Verzeichnisses 'dirname' aus der Task 'task' enthält.

THESAURUS PROC all (TASK CONST task, TEXT CONST dirname, SHAPE CONST shape)

Wirkung: Liefert einen Thesaurus, der alle Namen von Objekten der Gestalt 'shape' des Verzeichnisses 'dirname' aus der Task 'task' enthält.

Beispiel: `copy (all (myself, "TP/TVISION", ds) LIKE "*.PAS",/"DOK", ".")`
Lies: »kopiere alle Datenräume aus dem Verzeichnis "TP/TVISION" dieser Task, deren Name mit ".PAS" endet, in das Wurzelverzeichnis der Task "DOK".

FILLBY

OP FILLBY (THESAURUS VAR thesaurus, FILE VAR file)

Wirkung: Schreibt 'file' in den Thesaurus. Dabei werden Zeilen, die schon im Thesaurus sind, nicht mehr in den Thesaurus geschrieben. Jede Zeile kommt im Thesaurus also nur einmal vor.

OP FILLBY (FILE VAR file, THESAURUS CONST thesaurus)

Wirkung: Schreibt den Thesaurus in die Datei 'file'.

OP FILLBY (TEXT CONST filename, THESAURUS CONST thesaurus)

Wirkung: Richtet eine Datei mit dem Namen 'filename' ein und schreibt den Thesaurus in die Datei.

LIKE

THESAURUS OP LIKE (THESAURUS CONST thesaurus, TEXT CONST muster)

Wirkung: Alle im Thesaurus enthaltenen Dateien, die dem 'muster' entsprechen sind im Ergebnisthesaurus enthalten. Das 'muster' kann insbesondere durch Verwendung des "*"-Operators gebildet werden.

Beispiel: `print (all LIKE "*.p")`

SOME

THESAURUS OP SOME (THESAURUS CONST thesaurus)

Wirkung: Bietet den angegebenen Thesaurus zum editieren an. Dort können nicht erwünschte Namen gestrichen werden.

THESAURUS OP SOME (TASK CONST task)

Wirkung: Aufruf von: `SOME ALL task`.

THESAURUS OP SOME (TEXT CONST file name)

Wirkung: Aufruf von: `SOME ALL filename`.

do

PROC do (PROC (TEXT CONST) operate, THESAURUS CONST thes)

Wirkung: Die Prozedur 'operate' wird mit allen in 'thes' enthaltenen Namen als Parameter ausgeführt. Wenn in 'thes' n Namen enthalten sind, wird 'operate' n-mal ausgeführt. Die Reihenfolge ergibt sich durch die Indizes in 'thes'.

Bemerkung: Häufig haben Prozeduren genau einen Textparameter, der den Dateinamen der Datei, auf die diese Prozedur angewendet werden soll, übergibt. Da ELAN generische Prozeduren kennt, läßt sich mit Hilfe dieses **do** leicht die Anwendung dieser Prozedur auf Mengen von Namen, also mehrere Dateien, konstruieren.

Beispiel:

```
(* 1 *) do (PROC (TEXT CONST) lexsort, SOME all);

(* 2 *) PACKET new packet DEFINES myproc:

        PROC myproc (TEXT CONST filename):
        ......
        END PROC myproc;

        PROC myproc (THESAURUS CONST thes):
          do (PROC(TEXT CONST) myproc, thes)
        END PROC myproc
```

remainder

THESAURUS PROC remainder

Wirkung: Liefert nach einem **errorstop** die noch nicht abgearbeiteten Dateien.

Beispiel: Während der Ausführung von `saveall (archivea)` erfolgt ein Abbruch vom Archivmanager mit der Meldung »Archiv voll«. Nach Einlegen einer neuen Diskette können die noch nicht auf Diskette geschriebenen Dateien mit `save (remainder, archive)` gesichert werden.

8.2.3 Verwaltung von Objekten

Aus Gründen der Übersichtlichkeit und vor allem der Kompatibilität zu DOS muß neben der linearen Namensordnung **THESAURUS** noch die hierarchische Namensordnung **DIRECTORY** (Verzeichnis) verfügbar sein. Bei L3 sind Verzeichnisse nichts weiter als Namenspräfixe, die »Sichtbarkeitsregeln« für Dateien ergeben. Beispielsweise sind die Dateien des Verzeichnisses `MYDIR` genau alle die Dateien in der Task, deren voller Name mit `\MYDIR\` beginnt. Tiefere Verschachtelungen von Verzeichnissen sind also Präfixverkettungen auf Dateinamen. Wie üblich kann man »in Verzeichnisse wechseln«. Dateien können dann mit ihrem verkürzten Namen (ohne Verzeichnispräfix) bezeichnet werden.

Zur Realisierung dieser Namenshierarchie gibt es in einer Task zwei »Gestalten« (**SHAPES**) von lokalen Objekten: **dir** (Verzeichnis) und **ds** (Dataspace). Ein Verzeichnis ist also etwas gänzlich anderes als ein Datenraum (eine Datei). Eine Datei (meistens synonym für Textdatei) ist ein besonderer Typ Datenraum. Es ist durchaus zulässig einem Datenraum die Gestalt eines Eimers zuzugestehen, gleichgültig ob Texteimer oder anderer Eimer. Ein Verzeichnis hingegen ist von körperloser Gestalt, insbesondere hat das zur Folge, daß man es für sich nicht kopieren oder verschicken kann, sondern nur die mit seinem Namenspräfix existierenden Datenräume.

Allg. Prozeduren

SHAPE	Objekttyp
:=	
=	Vergleich
<>	— " —
anyshape	Menge aller Gestalten
dir	Typ Verzeichnis
ds	Typ Datenraum
LIKE	Schnittmenge
nilshape	ungültig
OR	Vereinigungsmenge

Pfadnamen

\	Konkatenation
extend	Namensentfaltung
head	erster Präfix
header	alle Präfixe
is relative path	Start ohne\ oder /
is simple name	Name ohne Trenner
PREFIXES	echter Teilname
tail	letzter Suffix
trailer	alle Suffixe

Besondere Symbole

/	Trenner für Teilnamen
\	— " —
.	aktuelles Verzeichnis
..	Vorgängerverzeichnis

SHAPE

TYPE SHAPE

Wirkung: Datentyp. Bezeichnet die Gestalt eines benannten Objekts.

:=

OP := (SHAPE VAR dest, SHAPE CONST source)

Wirkung: Zuweisung für SHAPE.

=

BOOL OP = (SHAPE CONST left, right)

Wirkung: Liefert TRUE, falls 'left' gleich 'right' ist.

<>

BOOL OP <> (SHAPE CONST left, right)

Wirkung: Liefert TRUE, falls 'left' ungleich 'right' ist.

any shape

SHAPE CONST any shape

Wirkung: Bezeichnet die Menge aller Gestalten.

dir

SHAPE CONST dir

Wirkung: Bezeichnet die Gestalt »Verzeichnis«.

ds

SHAPE CONST ds

Wirkung: Bezeichnet die Gestalt »Datenraum«.

LIKE

BOOL OP LIKE (SHAPE CONST left, right)

Wirkung: Liefert TRUE, falls 'left' und 'right' eine nicht leere Schnittmenge haben.

nil shape

SHAPE CONST nil shape

Wirkung: Bezeichnet eine ungültige Gestalt.

OR

SHAPE OP OR (SHAPE CONST left, right)

Wirkung: Bildet eine Menge aus 'left' und 'right'.

8.2.4 Pfadnamen

Pfadnamen sind Texte, die die Zeichen '/' bzw. '\' als Trenner enthalten können.

\

TEXT OP \ (TEXT CONST header, tail)

Wirkung: Liefert Pfadnamen aus 'header' und 'tail'.

extend

PROC extend (TEXT CONST path name, prefix, TEXT VAR extended path name)

Wirkung: Erzeugt einen vollqualifizierten Dateinamen. Falls etwa interaktiv in einem Unterverzeichnis eine Datei gelöscht werden soll, wird zunächst aus dem eingegebenen relativen Namen 'file name' und dem 'prefix' **current directory** der absolute Objektnamen extendiert. Das Resultat wird in 'extended path name' geliefert. Dieser 'extended path name' wird dann in der Kontrollabfrage ausgegeben und intern zum Löschen benutzt.

Beispiel:

```
TEXT VAR abs path name := "",
extend (std, current directory, abs path name);
putline (abs path name)
```

head

TEXT PROC head (TEXT CONST pathname)

Wirkung: Liefert den Teilpfadnamen bis zum ersten Trenner.

Beispiel: `head ("ab\cd\ef") => "ab"`

header

TEXT PROC header (TEXT CONST pathname)

Wirkung: Liefert den Teilpfadnamen bis zum letzten Trenner.

Beispiel: `header ("ab\cd\ef") => "ab\cd"`

is relative path

BOOL PROC is relative path (TEXT CONST path name)

Wirkung: Liefert 'TRUE', falls der Pfadname nicht mit einem Trenner beginnt.

is simple name

BOOL PROC is simple name (TEXT CONST path name)

Wirkung: Liefert 'TRUE', falls der Pfadname keine Trenner enthält.

PREFIXES

BOOL OP PREFIXES (TEXT CONST left path, right path)

Wirkung: Liefert TRUE, falls 'left path' **header** von 'right path' ist.

tail

TEXT PROC tail (TEXT CONST pathname)

Wirkung: Liefert den Teilpfadnamen hinter dem letzten Trenner.

Beispiel: `tail ("ab\cd\ef") => "ef"`

trailer

TEXT PROC trailer (TEXT CONST pathname)

Wirkung: Liefert den Teilpfadnamen hinter dem ersten Trenner.

Beispiel: `trailer ("ab\cd\ef") => "cd\ef"`

8.3 Manager Operationen

Mit den Datentypen **SHAPE** und **THESAURUS** stehen Hilfsmittel zur Verfügung, um die Verwaltung der lokalen Objekte einer Task zu bewerkstelligen. Die Verwaltung erledigt der »Catalogue Handler« der Task.

Die Objektverwaltung in einer Task wird durch eine maximal 16382 Einträge enthaltende Liste realisiert. Objekte können über ihren Namen oder ihren Listenindex (**catalogue index**) identifiziert werden. Die lokale Verwaltung umfaßt nicht nur die Namensverwaltung für die Datenräume und Verzeichnisse in der Task, sondern auch die Verwaltung der (Datei)attribute.

Objektnamen, in der Beschreibung der Prozeduren stets als `TEXT CONST pathname` auftauchend, können vollqualifizierte Name mit Pfadangabe sein. Falls man auf die Verwendung von Verzeichnissen verzichtet, kann dieser Pfadname einfach als Dateiname `TEXT CONST filename` verstanden werden.

Prozeduren

all	Liefert Thesaurus
catalogue index	Index des Objekts
catalogue name	Name zu Index
copy	Duplizierende Kopie
create	Erzeugen
current directory	eingestelltes Verzeichnis
dataspace	Typ
delete	Löschen
exists	Gibt es?
is empty directory	Verzeichnis ohne Objekte
list	Liste ohne Unterverzeichnisliste
move	Verschieben
name	Datenraumname
remove	Löschen
rename	Umbenennen
shape	Gestalt
xlist	Liste mit Unterverzeichnisliste
ATTRIBUTE	Objektattribute
nilattribute	Initialisierung
:=	Zuweisung
attributes	Liefert Attribute
attribute mask	
last access time	Zeit des letzten Zugriffs
last update time	Zeit des letzten Schreibens
last backup time	Zeit der letzten Sicherung

all

THESAURUS PROC all

Wirkung: Liefert den THESAURUS, der alle Objektnamen des aktuellen Verzeichnisses enthält.

THESAURUS PROC all (TEXT CONST path name)

Wirkung: Liefert den THESAURUS, der alle Objektnamen des angegebenen Verzeichnisses 'path name' enthält.

THESAURUS PROC all (TEXT CONST path name, SHAPE CONST mask)

Wirkung: Liefert den THESAURUS, der die Namen aller Objekte der Gestalt 'mask' des des aktuellen Verzeichnisses enthält.

catalogue index

INT PROC catalogue index (TEXT CONST name)

Wirkung: Liefert den Katalogindex des Objekt 'name'. Der Index liegt zwischen 1 und 16382. Falls der Index 0 geliefert wird, bedeutet das: »'name' gibt es nicht«.

catalogue name

TEXT PROC catalogue name (INT CONST index)

Wirkung: Liefert den Namen des Objekts mit dem Katalogindex 'index'. Falls kein Objekt existiert, wird niltext geliefert.

copy

PROC copy (TEXT CONST source path, dest path)

Wirkung: Kopiert die Quelle 'source path' nach 'dest path'. Attribute werden mitkopiert. Die Quelle wird über einen Zieldatenraum oder in ein Zielverzeichnis kopiert. Falls 'source path' ein Verzeichnis ist, werden alle seine Objekte kopiert. Datenräume werden nach Zustimmung bei der Kontrollabfrage überschrieben.

PROC copy (TEXT CONST source path, dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

PROC copy (THESAURUS CONST names, TEXT CONST dest path)

Wirkung: Kopiert alle Objekte aus 'names' nach 'dest path'. Attribute werden mitkopiert. Existiert 'dest path' nicht, so wird ein Verzeichnis angelegt, in das 'names' hineinkopiert wird.

PROC copy (THESAURUS CONST names, TEXT CONST dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

PROC copy (DATASPACE CONST source ds, TEXT CONST dest path)

Wirkung: Erzeugt ein Objekt 'dest path' und kopiert den Datenraum 'source ds'.

PROC copy (DATASPACE CONST source ds, TEXT CONST dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

create

PROC create (TEXT CONST path name, SHAPE CONST shape)

Wirkung: Ein Objekt der angegebenen Gestalt wird erzeugt. Falls erforderlich, werden Verzeichnisse wie in 'pathname' angegeben, erzeugt.

Beispiel: `create ("ab\cd\ef", ds)`
```
(* erzeugt Datenraum "ef" im Unterverzeichnis "ab\cd" *)
(* das Verzeichnis wird ggf. neu erzeugt *)
create ("NEW DIR", dir)
(* erzeugt ein Verzeichnis *)
```

PROC create (TEXT CONST path name)

Wirkung: wie `create (pathname, ds)`

current directory

TEXT PROC current directory

Wirkung: Liefert den Namen des aktuellen Verzeichnisses.

PROC current directory (TEXT CONST path name)

Wirkung: Stellt 'path name' als aktuelles Verzeichnis ein.

Fehler: "path name" gibt es nicht

dataspace

DATASPACE PROC dataspace (TEXT CONST pathname)

Wirkung: Liefert den Datenraum 'pathname'.

Fehler: "path name" gibt es nicht

delete

PROC delete (TEXT CONST path name)

Wirkung: Das Objekt 'pathname' wird gelöscht. Falls **command dialogue (TRUE)** gilt (Standardeinstellung), erfolgt zunächst eine Kontrollfrage. Falls ein Verzeichnis angegeben wird, müssen vor Löschen des Verzeichnisses alle darin enthaltenen Datenräume gelöscht werden.

PROC delete

Wirkung: Löscht das durch **std** benannte (zuletzt benutzte) Objekt.

PROC delete (TEXT CONST path name, QUIET CONST quiet)

Wirkung: Löscht wie oben beschrieben, aber ohne Kontrollabfrage.

PROC delete (THESAURUS CONST path names)

Wirkung: Löscht alle in der Liste bezeichneten.

PROC delete (THESAURUS CONST path name, QUIET CONST quiet)

Wirkung: Löscht alle in der Liste bezeichneten, aber ohne Kontrollabfrage.

Beispiel: `delete ("datei")`
```
delete ("mydir\subdir\file1.text", quiet)
delete ( SOME all)
```

exists

BOOL PROC exists (TEXT CONST path name)

Wirkung: `shape (path name) <> nil shape` Liefert TRUE, falls es das Objekt 'pathname' gibt.

is empty directory

BOOL PROC is empty directory (TEXT CONST path name)

Wirkung: Liefert TRUE, falls 'pathname' ein Verzeichnis spezifiziert, das keine Datenräume oder Unterverzeichnisse enthält.

list

PROC list

Wirkung: Listet alle Objekte des aktuellen Verzeichnisses in einer temporären Datei auf. Von Verzeichnissen wird nur der Name, nicht der Inhalt gelistet.

PROC list (TEXT CONST path name)

Wirkung: Listet alle Objekte des Verzeichnisses 'path name' auf.

PROC list (TEXT CONST path name, SHAPE CONST shapemask)

Wirkung: Listet alle Objekte des Verzeichnisses 'path name' auf, deren Gestalt 'shapemask' entspricht.

Beispiel: `list ("\", dir)` listet alle Verzeichnisnamen der Task.

PROC list (FILE VAR file)

Wirkung: Wie `list`, die Ausgabe wird jedoch in die eine mit **sequential file** assoziierte Datei 'file' gelenkt.

Bemerkung: Auch die anderen **list** Prozeduren können mit einer `FILE VAR` parametrisiert werden.

move

PROC move (TEXT CONST source path, dest path)

Wirkung: **move** entspricht **copy** mit Löschung der Quelle. Intern ist also nur eine Umbenennung erforderlich. Falls 'source path' ein Verzeichnis ist, werden alle seine Objekte verschoben. Datenräume werden nach Zustimmung bei der Kontrollabfrage überschrieben.

PROC move (TEXT CONST source path, dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

PROC move (THESAURUS CONST names, TEXT CONST dest path)

Wirkung: Versetzt alle Objekte aus 'names' nach 'dest path'.

PROC move (THESAURUS CONST names, TEXT CONST dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

PROC move (DATASPACE CONST source ds, TEXT CONST dest path)

Wirkung: Erzeugt ein Objekt 'dest path' und versetzt den Datenraum 'source ds'.

PROC move (DATASPACE CONST source ds, TEXT CONST dest path, QUIET CONST quiet)

Wirkung: s.o. aber ohne Kontrollabfrage.

name

TEXT PROC name (DATASPACE CONST ds)

Wirkung: Falls der Datenraum 'ds' im Katalog enthalten ist, wird sein Name geliefert, sonst niltext.

remove

PROC remove (TEXT CONST path name)

Wirkung: Entspricht **delete**, soweit Datenräume betroffen sind. Das Löschen von nicht leeren Verzeichnissen wird jedoch grundsätzlich abgewiesen.

PROC remove (TEXT CONST path name, QUIET CONST quiet)

Wirkung: s.o. jedoch ohne Kontrollabfrage.

rename

PROC rename (TEXT CONST source path, new simple name)

Wirkung: Schreiberleichterung. Entspricht: `move (source path, header (source path\ new simple name)`

Bemerkung: **last param** wird gesetzt.

shape

SHAPE PROC shape (TEXT CONST path name)

Wirkung: Liefert die Gestalt des Objekts 'path name'.

xlist

PROC xlist

Wirkung: Listet alle Objekte des aktuellen Verzeichnisses in einer temporären Datei auf. Von Verzeichnissen wird auch ihr Inhalt gelistet.

PROC xlist (TEXT CONST path name)

Wirkung: Listet alle Objekte des Verzeichnisses 'path name' auf.

PROC xlist (TEXT CONST path name, SHAPE CONST shapemask)

Wirkung: Listet alle Objekte des Verzeichnisses 'path name' auf, deren Gestalt 'shapemask' entspricht.

Beispiel: `xlist ("\", dir)` listet alle Verzeichnisnamen der Task.

Bemerkung: Alle **xlist** Prozeduren können mit einer `FILE VAR` parametrisiert werden. Die Ausgabe wird dann in eine Textdatei gelenkt.

ATTRIBUTES

TYPE ATTRIBUTES

Wirkung: Datentyp »Objekt (Datei)attribute«. Die Attribute sind:

- Datum/Zeit des letzten Zugriffs,
- Datum/Zeit des letzten Schreibzugriffs,
- Datum/Zeit des letzten Backups,

- SHAPE
- DOS Attributemaske

nilattributes

ATTRIBUTES PROC nilattributes

Wirkung: Liefert das Initialisierungsobjekt für ATTRIBUTES.

:=

OP := (ATTRIBUTES VAR left, ATTRIBUTES CONST right)

Wirkung: Zuweisungsoperator für ATTRIBUTES.

attributes

ATTRIBUTES PROC attributes (TEXT CONST path name)

Wirkung: Liefert die Attribute des Objekts 'path name'.

PROC attributes (TEXT CONST path name, ATTRIBUTES CONST attr)

Wirkung: Das durch 'pathname' identifizierte Objekt erhält das Attribute 'attr'.

attribute mask

INT PROC attribute mask (TEXT CONST path name)

Wirkung: Liefert die DOS Attributmaske des Objekts 'path name'.

last access time

DATETIME PROC last access time (ATTRIBUTES CONST attr)

Wirkung: Liefert aus 'attr' Datum/Uhrzeit der letzten Sicherung mit einem Sicherungssystem.

last update time

DATETIME PROC last update time (ATTRIBUTES CONST attr)

Wirkung: Liefert aus 'attr' Datum/Uhrzeit des letzten Schreibzugriffs.

last backup time

DATETIME PROC last backup time (ATTRIBUTES CONST attr)

Wirkung: Liefert aus 'attr' Datum/Uhrzeit des letzten Zugriffs.

8.3.1 Managerprozeduren

Wie bereits in 8.1 erwähnt wird die Managerfunktion einer Task durch eine Rahmenprozedur der Art:

```
PROC manager (PROC (INT CONST, TASK CONST) execute operation)
```

realisiert. Die Dienstprozedur **execute operation** wird von der Prozedur **manager** aufgerufen, wenn ein Auftrag empfangen wurde. Der Auftrag und ggf. Parameter werden dem Botschaftsvektor entnommen.

Falls die von der Rahmenprozedur aufgerufene Prozedur *execute operation* den vorliegenden Auftrag nicht bearbeiten kann, so ist möglich, daß diese Prozedur eine andere Dienstprozedur *execute operation* aufruft. Auf diese Weise läßt sich eine Hierarchie von Prozeduren aufbauen, die eine flexible Reaktion auf möglich Aufträge ermöglichen.

Die Zugriffsrechte auf Managerleistungen werden durch eine »access mask« beschrieben. Darin wird definiert, welche Art des Zugriffs (lesend, schreibend, zerstörend) von welcher Gruppe von Tasks zugelassen ist.

Beispiel:

```
PACKET read manager
       DEFINES read manager :

LET delete   ops  =  0x 00 00 00 01,
    write    ops  =  0x 00 00 01 00,
    read     ops  =  0x 00 01 00 00,
    info     ops  =  0x 01 00 00 00,
    free     bit  =  1,
    family   bit  =  2,
    system   bit  =  4,
    terminal bit  =  8;

INT CONST read access  :=     delete ops *  0
                           OR write  ops *  0
                           OR read   ops *  free bit
                           OR info   ops *  free bit;

PROC read manager :

  manager type ("read only");
  manager (PROC (INT CONST, TASK CONST) read manager);

END PROC read manager;

PROC read manager (INT CONST order code, TASK CONST order task) :
```

```
    object manager (order code, order task, read access)

  END PROC read manager;

  END PACKET read manager;
```

8.3.2 Zugriffsrechte

Rechte für den Zugriff auf Objekte in der Task können in der oben skizzierten Weise definiert werden. Für jede der vier Zugriffsarten »Löschen«, »Schreiben«, »Lesen« und »Info« wird in einem Byte der (Integer) Maske festgelegt, von wo aus dem System auf diese Weise zugegriffen werden darf. Folgende Rechte sind vordefiniert:

monitor access

Wirkung: Lösch-, Schreib-, Lese-, Infooperationen nur von Tasks aus, die im selben Terminalstack stehen.

family access

Wirkung: Lösch-, Schreib-, Lese-, Infooperationen nur von Tasks aus, die im selben Terminalstack stehen und von Tasks, die direkter Nachkomme der Task sind.

free access

Wirkung: Lösch-, Schreib-, Lese-, Infooperationen von allen Tasks aus.

system access

Wirkung: Lösch-, Schreiboperationen nur von Systemtasks (**system privilege**) aus, Lese-, Infooperationen von jeder Task aus.

8.3.3 Prozeduren

Manager	*Dienstprozedur*
void manager	keine Aktion
command manager	Kommandodialog
object manager	Dateimanager
begin manager	Dateimanager
monitor manager	
family manager	
free manager	
manager type	Typkennzeichnung einstellen

void manager

PROC void manager (INT CONST order code, TASK CONST order task, INT CONST access mask))

Wirkung: Weist jeden Auftrag von jeder Task ab. Es wird ein **errorstop** mit der Meldung `falscher Auftrag für Task ...` ausgelöst.

command manager

PROC command manager (INT CONST order code, TASK CONST order task, INT CONST access mask))

Wirkung: Akzeptiert nur den Auftrag sich an ein Terminal zu koppeln. Wenn die Task an ein Terminal gekoppelt ist, beginnt ein Kommandodialog. Insbesondere erlaubt ein solcher Manager also die in Kap. 8.4 beschriebenen Aktionen zur lokalen Dateiverwaltung.

object manager

PROC object manager (INT CONST order code, TASK CONST order task, INT CONST access mask))

Wirkung: Stellt seine lokalen Objekten (Datenräume, Verzeichnisse), anderen Tasks zur Verfügung. Wenn ein solcher Manager nicht zum Dialog an ein Terminal gekoppelt ist, erwartet er Aufträge zur Dateiverwaltung. Solche Aufträge können z.B derart sein, daß ein zugesandter Datenraum unter einem ebenfalls zugestellten Namen in das lokale Dateisystem eingefügt werden soll.

Bemerkung: Die Codierung und Parametrisierung der zulässigen Aufträge findet sich in Kap.13.

begin manager

PROC begin manager (INT CONST order code, TASK CONST order task, INT CONST access mask))

Wirkung: Akzeptiert Aufträge wie ein **object manager** und zusätzlich den Auftrag Sohntasks einzurichten.

Bemerkung: Auch »normale Tasks« können Sohntasks anlegen, jedoch nur, wenn der Auftrag in der Task selbst erzeugt wird (interaktiv oder durch Programm). Dem **begin manager** kann der Auftrag jedoch von außen, z.B. von der Terminaltask aus, zugestellt werden.

monitor manager

PROC monitor manager (INT CONST order code, TASK CONST order task)

Wirkung: Die Task wird ein **command manager** mit Zugriffsrecht **monitor access**.

family manager

PROC family manager (INT CONST order code, TASK CONST order task)

Wirkung: Die Task wird ein **object manager** mit Zugriffsrecht **family access**.

free manager

PROC free manager (INT CONST order code, TASK CONST order task)

Wirkung: Die Task wird ein **begin manager** mit Zugriffsrecht **free access**.

manager type

PROC manager type (TEXT CONST string)

Wirkung: Der Text 'string' wird hinter dem Tasknamen ausgegeben, immer wenn die Task an einen Bildschirm gekoppelt ist. So ist leicht erkennbar, welche Rechte die Task anderen einräumt. Beispielsweise wird ein **free manager** als (FREI) gekennzeichnet usw.

8.3.4 Aufträge an Objektmanager

Die folgenden Prozeduren sind die standardmäßig verfügbaren Aufträge an Objektmanager.

Prozeduren

all	Thesaurus
attributes	Liefert Attribute
copy	Schicken / Holen
create	Erzeugen
delete	Löschen
erase	Löschen
exists	Gibt es?
fetch	Holen
fetchall	Ganzen Thes. holen
is empty directory	Verzeichnis leer
list	Liste ohne Verzeichnisse
manager params	Zur bes. Verwendung
manager receive timeout	Wartezeit auf Lieferung
manager send timeout	Wartezeit auf Annahme
move	Verschieben
print	Drucken
printer	Druckertask
remove	Löschen
save	Schicken
saveall	Eigenen Thesaurus schicken
shape	Gestalt eines Objekts
storage	Größe eines Datenraums
type	Typ eines Datenraums
xlist	Liste mit Verzeichnissen

all

THESAURUS PROC all (TASK CONST manager, TEXT CONST dir, SHAPE CONST shape)

Wirkung: Der Manager 'task' liefert einen THESAURUS, der die Namen aller seiner Objekte der Gestalt 'shape' des Verzeichnisses 'dir' enthält.

THESAURUS PROC all (TASK CONST manager, TEXT CONST dir)

Wirkung: Der Manager liefert einen THESAURUS mit den Namen aller seiner Objekte im Verzeichnis 'dir'.

THESAURUS PROC all (TASK CONST manager)

Wirkung: Der Manager liefert einen Thesaurus mit den Namen aller seiner Objekte.

attributes

ATTRIBUTES PROC attributes(TASK CONST manager, TEXT CONSTpathname)

Wirkung: Der Manager liefert die Attribute zu 'pathname'.

copy

PROC copy (TASK CONST sourcetask, TEXT CONST sourcename, destname)

Wirkung: Aus der Task 'sourcetask' wird 'sourcename' unter dem Namen 'destname' kopiert.

PROC copy (TASK CONST sourcetask, THESAURUS CONST sourcenames, TEXT CONST destname)

Wirkung: Aus der Task 'sourcetask' werden alle in 'sourcenames' beschriebenen Datenräume in das eigene Verzeichnis 'destname' kopiert.

PROC copy (TASK CONST sourcetask, TEXT CONST sourcename)

Wirkung: Aus 'sourcetask' wird 'sourcename' kopiert.

PROC copy (TASK CONST sourcetask, THESAURUS CONST sourcenames)

Wirkung: Aus 'sourcetask' werden alle in 'sourcenames' beschriebenen Datenräume kopiert. Die Verzeichnisstruktur von 'sourcetask' wird übernommen. Leere Verzeichnisse werden jedoch nicht kopiert.

PROC copy (TEXT CONST sourcename, TASK CONST dest task, TEXT CONST destpath)

Wirkung: Der Datenraum 'sourcename' wird unter dem Namen 'destpath' in die Task 'desttask' kopiert. Falls es 'destpath' schon in 'desttask' gibt, kommt eine Kontrollfrage `"destpath" überschrieben? (j/n)`.

PROC copy (THESAURUS CONST sourcenames, TASK CONST desttask, TEXT CONST destname)

Wirkung: Die in 'sourcenames' angegebenen Datenräume werden in das Verzeichnis 'destname' von 'desttask' kopiert.

PROC copy (TEXT CONST sourcename, TASK CONST desttask)

Wirkung: Kopiert 'sourcename' nach 'desttask'.

PROC copy (THESAURUS CONST sourcenames, TASK CONST desttask)

Wirkung: Kopiert die in 'sourcenames' angegebenen Datenräume unter ihrem vollen Namen nach 'desttask'.

Bemerkung: Alle Prozeduren können auch als zusätzlichen Parameter eine QUIET CONST `quiet` erhalten. In diesem Falle unterbleiben Kontrollabfragen vor dem Überschreiben.

create

PROC create (TASK CONST manager, TEXT CONST pathname, SHAPE CONST shape)

Wirkung: Erzeugt in 'manager' ein Objekt 'pathname' der Gestalt 'shape'.

Beispiel: `create (/"PUBLIC", "A new dir", dir)`

delete

PROC delete (TASK CONST manager, TEXT CONST pathname)

Wirkung: Löscht in 'manager' das Objekt pathname mit Kontrollfrage. Falls 'pathname' ein Verzeichnis ist, werden zunächst alle Unterverzeichnisse und Datenräume nach Abfrage gelöscht. Falls 'pathname' danach »leer« ist wird 'pathname' selbst gelöscht.

PROC delete (TASK CONST manager, THESAURUS CONST thes)

Wirkung: Löscht in 'manager' die in 'thes' angegebenen Datenräume.

erase

PROC erase (TEXT CONST name)

Wirkung: Der Datenraum- (meistens Datei-)name 'name' wird mit **current directory** zum vollständigen 'pathname' extendiert. Der derart spezifizierte Datenraum wird in der Vatertask (die in diesem Falle natürlich Manager sein muß) nach Kontrollfrage gelöscht.

Bemerkung: Im wesentlichen stellt das eine Schreiberleichterung dar, da man in einem Unterverzeichnis befindlich, einfach den verkürzten Namen verwenden kann.

Beispiel:

```
REF HB\subdir1\subsubdir       (PRIVAT)
        erase ("text")
```

wirkt wie: `delete (father, "\subdir1\subsubdir\text")`

PROC erase (THESAURUS CONST thes)

Wirkung: Wie oben, aber alle in 'thes' angegebenen Datenräume.

PROC erase

Wirkung: Wie oben, aber die zuletzt benutzte Datei (siehe **std**).

PROC erase (TEXT CONST name, TASK CONST manager)

Wirkung: Wie oben, aber in der Task 'manager'.

PROC erase (THESAURUS CONST thes, TASK CONST manager)

Wirkung: Wie oben, aber in der Task 'manager'.

exists

BOOL PROC exists (TASK CONST manager, TEXT CONST pathname)

Wirkung: Liefert TRUE, falls es eine Objekt 'pathname' in 'manager' gibt.

fetch

PROC fetch (TEXT CONST name)

Wirkung: Der Datenraum- (meistens Datei-)name 'name' wird mit **current directory** zum vollständigenn 'pathname' extendiert. Der derart spezifizierte Datenraum wird aus der Vatertask (die in diesem Falle natürlich Manager sein muß) kopiert.

Bemerkung: Im wesentlichen stellt das eine Schreiberleichterung dar, da man in einem Unterverzeichnis befindlich, einfach den verkürzten Namen verwenden kann.

Beispiel:

```
REF HB\subdir1\subsubdir       (PRIVAT)
        fetch ("text")
```

wirkt wie: `copy (father, "\subdir1\subsubdir\text")`

PROC fetch (THESAURUS CONST thes)

Wirkung: Wie oben, mit allen Datenräumen aus 'thes'.

PROC fetch (TEXT CONST name, TASK CONST manager)

Wirkung: Wie oben, aber aus der Task 'manager'.

PROC fetch (THESAURUS CONST thes, TASK CONST manager)

Wirkung: Wie oben, aber aus der Task 'manager'.

fetchall

PROC fetchall (TASK CONST manager)

Wirkung: Holt alle Datenräume aus 'manager'. Die Verzeichnisstruktur von 'manager' wird vollständig übernommen.

PROC fetchall

Wirkung: Holt alle Datenräume aus der Vatertask 'manager'. Die Verzeichnisstruktur der Vatertask wird vollständig übernommen.

is empty directory

BOOL PROC is empty directory (TASK CONST manager, TEXT CONST dirname)

Wirkung: Liefert TRUE , falls das Verzeichnis 'dirname' in der Task 'manager' leer ist.

list

PROC list (TASK CONST manager)

Wirkung: Listet Verzeichnisnamen und Datenraumnamen aus dem »root directory« der Task 'manager' in einer temporären Datei, die mit **show** angezeigt wird. Von Verzeichnissen wird nur der Name, nicht der Inhalt, angezeigt.

PROC list (TASK CONST manager, TEXT CONST dirname)

Wirkung: Wie oben, aber Verzeichnis 'dirname' als Ausgangspunkt.

PROC list (TASK CONST manager, TEXT CONST dirname, SHAPE CONST shape)

Wirkung: Wie oben, aber nur Objekte der Gestalt 'shape'.

Beispiel: `list (archive, ".", dir)` zeigt alle Verzeichnisse des Archivs.

PROC list (FILE VAR file, TASK CONST manager)

PROC list (FILE VAR file, TASK CONST manager, TEXT CONST dirname, SHAPE CONST shape)

Wirkung: Wie oben, die Ausgabe erfolgt aber in eine Datei.

manager params

PROC manager params (TEXT CONST param1, TEXT CONST param2)

Wirkung: Mit diesen beiden Parameter kann einem Manager zusätzliche Information für besondere Aufgaben übermittelt werden. Beispielsweise wird dem »mail manager« so der Empfänger einer Botschaft übermittelt.

Siehe: Beispieldiskette, Datei "mail manager", "mail functions"

manager receive timeout

PROC manager receive timeout (INT CONST wait)

Wirkung: Wartet 'wait' Millisekunden darauf, daß ein Manager auf einen Auftrag der mit einer Sendung beantwortet werden soll, erwartungsgemäß reagiert.

Bemerkung: Dient insbesondere dazu, daß Programme, die mit Managern kommunizieren, die sich nicht wie Standardmanager verhalten, nicht hängen bleiben, wenn z.B. ein »list« keine Reaktion seitens des Managers hervorruft. Voreinstellung ist **never**!

INT PROC manager receive timeout

Wirkung: Liefert den aktuellen Wert.

manager send timeout

PROC manager send timeout (INT CONST wait)

Wirkung: Wartet 'wait' Millisekunden darauf, daß ein Manager auf einen Auftrag, bei dem dieser Manager etwas empfangen soll, erwartungsgemäß reagiert.

Bemerkung: Dient insbesondere dazu, daß Programme, die mit Managern kommunizieren, die sich nicht wie Standardmanager verhalten, nicht hängen bleiben, wenn z.B. ein »save« keine Reaktion seitens des Managers hervorruft. Voreinstellung ist **never**!

INT PROC manager send timeout

Wirkung: Liefert den aktuellen Wert.

move

Wirkung: **move** entspricht dem oben beschriebenen **copy**. Der Unterschied besteht darin, daß nicht kopiert wird, sondern die Quelle gelöscht wird. Es ist also als »Verschieben« aufzufassen. Die mögliche Parametrisierung entspricht **copy**.

print

PROC print (TEXT CONST name)

Wirkung: Die Datei 'name' wird zum Drucken an **printer** geschickt.

PROC print (THESAURUS CONST thes)

Wirkung: Alle in 'thes' angegebenen Dateien werden zum Drucken an **printer** geschickt.

PROC print

Wirkung: Die zuletzt benutzte Datei wird zum Drucken an **printer** geschickt.

PROC print (TEXT CONST name, TASK CONST printer manager)

Wirkung: Die Datei 'name' wird zum Druck an 'printer manager' geschickt.

PROC print (THESAURUS CONST thes, TASK CONST printer manager)

Wirkung: s.o.

remove

PROC remove (TASK CONST manager, TEXT CONST pathname)

Wirkung: In 'manager' wird 'pathname' gelöscht. Falls es ein Verzeichnis ist, wird nur dann gelöscht, wenn 'pathname' leer ist. Im Gegensatz zu **delete** wird nicht versucht zuerst die enthaltenen Objekte und schließlich 'pathname' zu löschen.

PROC remove (TASK CONST manager, TEXT CONST pathname, QUIET CONST quiet)

Wirkung: Wie oben, aber ohne Kontrollabfrage.

save

PROC save (TEXT CONST name)

Wirkung: Der Datenraum- (meistens Datei-)name 'name' wird mit **current directory** zum vollständigenn 'pathname' extendiert. Der derart spezifizierte Datenraum wird zur Vatertask (die in diesem Falle natürlich Manager sein muß) kopiert.

Bemerkung: Im wesentlichen stellt das eine Schreiberleichterung dar, da man in einem Unterverzeichnis befindlich, einfach den verkürzten Namen verwenden kann.

Beispiel:

```
REF HB:\subdir1\subsubdir      (PRIVAT)
        save ("text")
```

wirkt wie: `copy("\subdir1\subsubdir\text",father)`

PROC save (THESAURUS CONST thes)

Wirkung: Wie oben, mit allen Datenräumen aus 'thes'.

PROC save

Wirkung: Wie oben, mit der zuletzt benutzten Datei.

PROC save (TEXT CONST name, TASK CONST manager)

Wirkung: Wie oben, aber in die Task 'manager'.

PROC save (THESAURUS CONST thes, TASK CONST manager)

Wirkung: Wie oben, aber in die Task 'manager'.

saveall

PROC saveall (TASK CONST manager)

Wirkung: Kopiert alle Datenräume nach 'manager'. Die eigene Verzeichnisstruktur wird vollständig übergeben.

PROC saveall

Wirkung: Kopiert alle Datenräume in die Vatertask. Die Verzeichnisstruktur wird vollständig übergeben.

shape

SHAPE PROC shape (TASK CONST manager, TEXT CONST pathname)

Wirkung: 'manager' liefert die Gestalt des Objekts 'pathname'.

storage

INT PROC storage (TASK CONST manager, TEXT CONST dsname)

Wirkung: 'manager' liefert die Größe des Datenraums 'pathname' in Byte.

Fehler: Operation mit einem Verzeichnis nicht erlaubt

type

INT PROC type (TASK CONST manager, TEXT CONST dsname)

Wirkung: 'manager' liefert den Typ des Datenraums 'dsname'.

xlist

PROC xlist (TASK CONST manager)

Wirkung: Listet Verzeichnisnamen und Datenraumnamen aus dem »root directory« der Task 'manager' in einer temporären Datei, die mit **show** angezeigt wird. Die vollständige Baumstruktur der Verzeichnisse/ Datenräume wird dargestellt.

PROC xlist (TASK CONST manager, TEXT CONST pathname)

Wirkung: Wie oben, aber Verzeichnis 'dirname' als Ausgangspunkt.

PROC xlist (TASK CONST manager, TEXT CONST pathname, SHAPE CONST shape)

Wirkung: Wie oben, aber nur Objekte der Gestalt 'shape'.

Beispiel: `xlist (archive, ".", dir)` zeigt alle Verzeichnisse des Archivs.

PROC xlist (FILE VAR file, TASK CONST manager)

PROC xlist (FILE VAR file, TASK CONST manager, TEXT CONST pathname, SHAPE CONST shape)

Wirkung: Wie oben, die Ausgabe erfolgt aber in eine Datei.

8.4 Archivemanager

Archive sind im L3 Sprachgebrauch Medien zur Datensicherung. Derzeit sind Disketten- und Bandlaufwerke als Betriebsmittel für Archive verfügbar. Insbesondere Diskettenlaufwerke nehmen also eine andere Stellung ein, als bei DOS. Man kann kein originäres L3 Programm »von der Diskette starten«, Disketten und Bänder dienen ausschließlich der Auslagerung von Daten. Die in L3DOS enthaltenen Möglichkeit DOS Programme von Diskette zu starten ist eine aus Gründen der Kompatibilität implementierte »Unmöglichkeit«.

Zur Benutzung des Diskettenlaufwerks ist bei L3 eine Reservierungslogik notwendig, um die ein oder zwei Diskettenlaufwerke des Rechners beliebigen Tasks zuordnen zu können. Zu diesem Zweck wird ein Diskettenlaufwerk stets von einem Archivmanager verwaltet. Ein Archivmanager ist ein spezieller Dateimanager, dessen Dateien sich nicht auf der Festplatte, sondern auf Diskette befinden. Die Kommunikation mit dem Archivmanager erfolgt in der üblichen Art und Weise (**fetch, save, copy..**).

8.4.1 Disketten

Der Archivmanager für das erste Diskettenlaufwerk heißt **ARCHIVE A**, der Archivmanager für ein zweites Diskettenlaufwerk heißt **ARCHIVE B**.

Um eine Diskette zu lesen oder zu beschreiben, muß stets eine Reservierung des Laufwerks beim Archivmanager vorgenommen werden. Nach Beendigung der Aktionen muß die Reservierung aufgehoben werden. Um zu verhindern, daß das Laufwerk versehentlich von einer Task nicht wieder freigegeben wird, hebt der Archivmanager eine Reservierung auf, falls seit 5 Minuten kein Zugriff auf das Laufwerk stattgefunden hat und ein Reservierungsgesuch einer anderen Task vorliegt. Falls keine andere Task versucht, das Laufwerk zu reservieren, bleibt die Zuordnung erhalten.

Disketten können mit Namen (Labels) versehen werden. Falls eine Diskette benannt ist, muß zur erfolgreichen Reservierung auch noch der Diskettenname angegeben werden (`reserve ("my floppy", /"ARCHIVE A")`). Nicht nur bei Mehrbenutzersystemen können durch diese Maßname unerfreuliche Verluste verhindert werden.

Die möglichen Aufträge an einen solchen Archivmanager sind die, die an einen »normalen« Dateimanager gegeben werden können (copy, list, save..). Sie sind hier nicht gesondert aufgeführt, da Unterschiede lediglich im Zeitverhalten bestehen. Erläutert sind die zusätzlichen, spezifischen Aufträge zur Diskettenbehandlung.

Disketten werden in einem DOS kompatiblen Dateiformat beschrieben. Neben der Möglichkeit, dateiweise zu lesen und zu schreiben, ist auch blockweises Lesen und Schreiben möglich.

8.4.2 Operationen des Archivmanagers

archive	Abk. für /"ARCHIVE A"
archive a	Abk. für /"ARCHIVE A"
archive b	Abk. für /"ARCHIVE B"
check	Prüflesen
clear	Initialisieren
format	Formatieren
release	Reservierung aufheben
reserve	Reservierung
volume label	Diskettenlabel
volume storage	Kapazität und Belegung
read	Block lesen
write	Block schreiben
read diskette	Diskette blockweise lesen
write diskette	Diskette blockweise schreiben

archive

TASK PROC archive

Wirkung: Liefert den internen Taskbezeichner von ARCHIVE A. Schreibvereinfachung.

Beispiel: `reserve (archive)`

```
(* statt *)
reserve(/"ARCHIVE A")
```

archive a

TASK PROC archive a

Wirkung: Liefert den internen Taskbezeichner von ARCHIVE A. Schreibvereinfachung.

Beispiel: `reserve (archive a)`

```
(* statt *)
reserve(/"ARCHIVE A")
```

archive b

TASK PROC archive b

Wirkung: Liefert den internen Taskbezeichner von ARCHIVE B. Schreibvereinfachung.

Beispiel: `reserve (archive a);`

```
reserve (archive b);
fetchall(archive a);
save (ALL archive a, archive b);

(* statt *)

reserve(/"ARCHIVE A");
reserve(/"ARCHIVE B").....
```

check

PROC check (TEXT CONST name, TASK CONST archivetask)

Wirkung: Prüflesen des Datenraums 'name' durch den Archivemanager 'archivetask' auf der eingelegten Diskette. Es werden keine Daten übertragen. Falls fehlerfrei gelesen werden kann, erscheint eine dementsprechende Meldung, sonst eine Fehlermeldung.

PROC check (THESAURUS CONST thes, TASK CONST archivetask)

Wirkung: Prüflesen aller im Thesaurus 'thes' angegebenen Datenräume durch 'archivetask'.

clear

PROC clear (TASK CONST archivetask)

Wirkung: Initialisiert die eingelegte Diskette. Falls durch **reserve** ein Archivname eingestellt wurde, erhält die eingelegte Diskette diesen Namen. Da die Initialisierung dem **Löschen** aller auf der Diskette vorhandenen Dateien entspricht, wird vorher ein Kontrolldialog geführt.

Die Dateien werden *nicht* physisch gelöscht. Es werden nur die Einträge im Katalog der Diskette gelöscht. Der Katalog läßt sich aber nicht wiederherstellen.

format

PROC format (TASK CONST archivetask)

Wirkung: Formatiert die Diskette in dem von der 'archivetask' verwalteten Laufwerk im Standardformat des Laufwerks.

PROC format (INT CONST code,TASK CONST archivetask)

Wirkung: Formatiert die Diskette in dem von der 'archivetask' verwalteten Laufwerk. 'code' spezifiziert das gewünschte Format. Falls ein anderer Wert als unten beschrieben oder eine für das Laufwerk nicht durchführbare Option gefordert wird, wird eine Meldung ausgegeben.

Falls bei Reservierung des Archivs ein Name eingestellt wurde, erhält die Diskette nach erfolgter Formatierung diesen Archivnamen.

Bemerkung: **format** beinhaltet also **clear**.

Parameter:

```
code = 0     :  high density (1200 oder 1440)
code = 360   :  360 KB, 40 Spuren,  9 Sektoren
code = 720   :  720 KB, 80 Spuren,  9 Sektoren
code = 1200  : 1200 KB, 80 Spuren, 15 Sektoren
code = 1440  : 1440 KB, 80 Spuren, 18 Sektoren
```

Fehler: Falscher Formatschlüssel

release

PROC release (TASK CONST archivetask)

Wirkung: Die Reservierung des Archivmanagers wird beendet.

reserve

PROC reserve (TASK CONST archivetask)

Wirkung: Reserviert den Archivmanager. Falls eine andere Task "t" diesen bereits reserviert hat, erscheint die Meldung: `Archiv wird von Task "t" benutzt` Es findet im weiteren keine Überprüfung des Archivnamens statt.

PROC reserve (TEXT CONST name, TASK CONST archivetask)

Wirkung: Wie oben, es wird zusätzlich bei folgenden Archivoperationen (save, fetch, format, clear ..) geprüft, ob die Diskette den Archivnamen 'name' hat.

volume label

TEXT PROC volume label (TASK CONST archivetask)

Wirkung: Liefert der Archivnamen der Diskette, die in dem durch 'archivetask' verwalteten Laufwerk ist.

Fehler: Archiv nicht angemeldet

volume storage

PROC volume storage (TASK CONST task, INT VAR size, used)

Wirkung: Der Archivemanager 'task' übergibt in 'size' die Kapazität, in 'used' den bereits belegten Speicherplatz der eingelegten Diskette. Die Angabe erfolgt in Bytes.

Beispiel:

```
reserve (archive a);
INT VAR size, used;
volume storage (archive, size, used);

IF size - used {\tt >} 32 * 1024
   THEN put ("mehr als 32 KByte frei")
   ELSE put ("weniger als 32 KByte")
FI
```

read

PROC read (TASK CONST task, BOOL CONST cyl, INT CONST sector no, stream adress, stream length)

Wirkung: Von der von 'task' verwalteten Diskette werden ab Beginn des Sektors 'sector no' 'stream length' Byte gelesen und beginnend ab Adresse 'stream adress' in den eigenen Standarddatenraum geschrieben.

Parameter:

task: Archivmanager

cyl: TRUE zylinderorientiertes Lesen
FALSE seitenorientiertes Lesen

sector no: erster zu lesender Sektor auf Diskette

stream adr: Adresse eines Datenobjekt im STDDS

stream length: Anzahl Bytes, muß durch 512 ohne Rest teilbar sein.

write

PROC write (TASK CONST task, BOOL CONST cyl, INT CONST sector no, stream adress, stream length)

Wirkung: Auf die von 'task' verwaltete Diskette werden ab Beginn des Sektors 'sector no' 'stream length' Byte geschrieben. Beginnend ab Adresse 'stream adress' wird aus dem eigenen Standarddatenraum gelesen.

Parameter: siehe oben.

sector no: erster zu schreibender Sektor auf der Diskette

Beispiel:

```
INT VAR x := archive blocks;
ROW 512 BYTE VAR buffer;

page;
putline("PATSCH....");
put ("Archiveblocks:");put (x); line (2) ;

read (archive a, TRUE, 1000, ADDR (buffer [1]), 512);

INT VAR return := int (buffer [100]);

put ("Wert ändern: ");
TEXT VAR tr := text (return) ;
editget (tr);
return := int (tr);

IF (0 < return) CAND (return < 255 )
    THEN buffer [100]  := byte (return)
FI;

line(2);

IF yes ("zurückschreiben")
    THEN write (archive a, TRUE , 1000, ADDR (buffer [1]),
                512);
FI;
```

read diskette

PROC read diskette (TEXT CONST dsname)

Wirkung: Liest eine Diskette vollständig in den zuvor erzeugten Datenraum 'dsname' ein. Der Inhalt des ersten Sektors von Spur 0 der Diskette findet sich zu Beginn der zweiten Seite des Datenraums (also ab Adresse 0x1000). Das Einlesen einer 360K Diskette erzeugt also einen Datenraum mit 91 **ds pages**.

Bemerkung: Das Diskettenlaufwerk muß reserviert sein. Die Prozedur beinhaltet einen Kontrolldialog.

write diskette

PROC write diskette (TEXT CONST dsname)

Wirkung: Schreibt den Inhalt des Datenraums 'dsname' auf eine Diskette. Der Inhalt des ersten Sektors von Spur 0 der Diskette ergibt sich aus den ersten 512 Bytes der zweiten Seite des Datenraums usw. (also ab Adresse 0x1000).

Bemerkung: Der Datenraum 'dsname' sollte zuvor mit **read diskette** erzeugt worden sein. Das Diskettenlaufwerk muß reserviert sein. Die Prozedur beinhaltet einen Kontrolldialog.

8.4.3 Bandlaufwerk

Ein Bandlaufwerk ist ein Archiv zur Speicherung großer Datenmengen. Die Übertragung zum/vom Band erfolgt dateiweise. Die Struktur eines beschriebenen Bandes läßt sich so darstellen:

```
Volume header | file header, data, filemark | file header, data, ......
```

Im »Volume header« wird insbesondere ein Bandname **volume name** eingetragen.

Man kann auf einem Band nicht überschreiben, sondern nur anfügen. Das Band kann nicht teilweise, sondern nur vollständig zurückgespult werden.

Der Manager für das Bandlaufwerk heißt **STREAMER**. Dieser Manager kommuniziert systemseitig mit dem (ladbaren) Treiber des Streamercontrollers (**systemtask("S0")**) und benutzerseitig mit irgendeiner Task, die eine Reservierung des Bands vorgenommen hat. Die benutzerseitige Schnittstelle ähnelt der des Diskettenlaufwerks.

Erläutert sind nicht nur die spezifischen Aufträge an den Streamermanager, sondern auch die normalen Dateimanageraufträge, da hier gewisse Unterschiede zu beachten sind.

8.4.4 Operationen des Streamermanagers

check	Prüflesen
clear	Initialisieren
format	Formatieren
release	Reservierung aufheben
reserve	Reservierung
streamer	Abk. für /"STREAMER"
volume label	Diskettenlabel

check

PROC check (TEXT CONST name, TASK CONST streamer)

Wirkung: Prüflesen des Datenraums 'name' durch **streamer** auf der eingelegten Bandkassette. Es werden keine Daten übertragen. Falls fehlerfrei gelesen werden kann, erscheint eine dementsprechende Meldung, sonst eine Fehlermeldung.

PROC check (THESAURUS CONST thes, TASK CONST archivetask)

Wirkung: Prüflesen aller im Thesaurus 'thes' angegebenen Datenräume durch **steamer**.

clear

PROC clear (TASK CONST streamer)

Wirkung: Initialisiert das eingelegte Band. Falls durch **reserve** ein Archivname eingestellt wurde, erhält das eingelegte Band diesen Namen (**volume label**) Da die Initialisierung dem **Löschen** aller auf dem Band vorhandenen Dateien entspricht, wird vorher ein Kontrolldialog geführt.

Die Dateien werden *nicht* physisch gelöscht. Es wird nur »end of tape« direkt hinter den »Volume Header« geschrieben.

format

PROC format (TASK CONST streamer)

Wirkung: Formatiert das Band. Da ein Band automatisch beim Beschreiben formatiert wird, ist formatieren nur sinnvoll, falls ein beschriebenes Band physisch gelöscht werden soll. Beinhaltet **clear**.

release

PROC release (TASK CONST streamer)

Wirkung: Die Reservierung des Streamermanagers wird beendet.

reserve

PROC reserve (TASK CONST streamer)

Wirkung: Reserviert den Streamermanager. Falls eine andere Task "t" diesen bereits reserviert hat, erscheint die Meldung: `Band wird von Task "t" benutzt`

PROC reserve (TEXT CONST name, TASK CONST archivetask)

Wirkung: Wie oben, es wird zusätzlich bei folgenden Archivoperationen (save, fetch, format, clear ..) geprüft, ob das Band den Archivnamen 'name' hat.

volume label

TEXT PROC volume label (TASK CONST archivetask)

Wirkung: Liefert der Archivnamen der Diskette, die in dem durch 'archivetask' verwalteten Laufwerk ist.

Fehler: Archiv nicht angemeldet

8.4.5 Besonderes Verhalten

PROC copy (..., TASK CONST streamer, TEXT CONST pathname)
PROC move (..., TASK CONST streamer, TEXT CONST pathname)
PROC save (..., TASK CONST streamer, TEXT CONST pathname)

Wirkung: Anfügendes Schreiben. Auf einem Band kann nicht überschrieben werden. Falls ein Datenraum mit dem angegebenen Namen auf dem Band ist, so wird der »alte« Datenraum umbenannt und der in einer der obigen Prozeduren angegebene »hinten« auf das Band geschrieben.

Die Umbenennung des schon auf dem Band befindlichen Datenraums erfolgt durch:

```
text (pathname) + "." + text (filenumber) + "#"
```

Bemerkung: Wegen des Formats der Fileheader auf dem Band werden Namen so gekürzt, daß sie maximal 99 Zeichen lang sind:

```
subtext (pathname, 1, 90) + text (filenumber) + "#"
```

PROC create (TEXT CONST pathname, TASK CONST streamer)
PROC delete (TEXT CONST pathname, TASK CONST streamer)
PROC move (TASK CONST streamer, TEXT CONST pathname,...)

Wirkung: Diese Aktionen sind auf einem Band nicht durchführbar.

9 Kommunikation mit Terminals

9.1 Einführung

Um mit einem Benutzer zu kommunizieren, also Eingaben zu empfangen oder Ausgaben zu machen, muß eine Task an ein Terminal gekoppelt sein. Ein Terminal im Sinne von L3 ist eine Tastatur und ein Bildschirm, gleichgültig, ob »fest eingebaut« wie die CONSOLE oder über serielle Schnittstelle angeschlossen. Im System wird ein Terminal durch eine Terminaltask repräsentiert, die drei Sohntasks hat. Eine dieser Sohntasks ist das Terminalmenü, die anderen beiden sind Manager für Tastatur und Bildschirm. Den Ausgangspunkt aller Zuordnungen eines Terminals an irgendwelche Tasks bildet die »Menütask«, die das Terminalmenü ausgibt. Diese Task, die ja aus Benutzersicht das Terminal darstellt, heißt für die CONSOLE `CONSOLE.term`, für weitere Terminals dementsprechend `??.term`.

Tastatur- und Bildschirmmanager sind im Grundzustand des Systems mit »ihrer« Menütask gekoppelt. Von dieser Task wird das Terminal an andere Tasks gekoppelt. »Eine Task an ein Terminal koppeln« bedeutet demzufolge, daß eine Zuordnung vorgenommen wird, die zur Folge hat, daß eine Task Botschaften vom Tastaturmanager erhält und Botschaften an den Bildschirmmanager absetzt. Die Regelung der Zuordnung von Terminals zu Tasks obliegt der Task SYSIO.

Ein Beschreibung der Botschaften zwischen Tasks und *SYSIO* findet sich in Kap.13.7.

Ausschnitt aus **taskinfo(2)**:

```
SYSIO..........................   0:08:04   1   wait   -
    T 2........................   0:00:02   1   wait   -
        T 2.term...............   0:00:00   1   wait   "T 2"
        T 2.kbd................   0:00:00   1   wait   -
        T 2.dsp................   0:00:01   1   wait   -
    CONSOLE....................   0:04:39   1   wait   -
        CONSOLE.term...........   0:01:32   1   wait   -
        CONSOLE.kbd............   0:01:32   1   wait   -
        CONSOLE.dsp............   0:15:48   1   wait   -
```

SYSIO führt für jedes Terminal einen Terminalstack, auf dem zunächst nur die Terminaltask eingetragen ist. Von der Terminaltask *CONSOLE* kommt nun ein Auftrag an SYSIO das Terminal an die Task *OPERATOR* zu koppeln, von der Terminaltask *T 2* der Auftrag *Pascal 5.5* anzukoppeln, von dort der Auftrag die Task *PRINTER* anzukoppeln (`link(printer)`). Intern stellt sich der Zustand der Taskstacks in SYSIO dann so dar:

```
                 PRINTER

OPERATOR         PASCAL 5.5

CONSOLE          T 2
```

Offensichtlich ist nach Abkoppeln von OPERATOR wieder die Terminaltask CONSOLE am Bildschirm sichtbar, nach Abkoppeln von PRINTER die Task PASCAL 5.5.

Das Erzeugen und Ankoppeln von Task kann durch Passworte überwacht werden. Da Tasks autonom sind und das Passwort in der Task aufbewahrt wird, ist dieser Mechanismus *hart*.

Wenn eine Task durch ein Passwort vor dem Ankoppeln geschützt ist, läßt sich diese Task so lange nicht zum Dialog ankoppeln, bis ihr das korrekte Passwort eingegeben wurde. Wenn das Passwort richtig eingegeben wurde, wird die Task angekoppelt. Solange sie im Terminalstack des zugeordneten Terminals steht, ist keine erneute Eingabe des Passworts nötig. Es ist also durchaus möglich per HotKey via Terminalmenü in eine andere Task zu wechseln und anschließend wieder in die geschützte Task zurückzukehren, ohne sich erneut zu identifizieren.

Die Zuordnung von Tasks und Terminals bleibt auch über Systemabschaltungen hinweg bestehen. Um zu verhindern, daß mit Passwort geschützte Terminals nach Neustart des Systems (noch) einem Terminal zugeordnet sind, kann durch die Prozedur **auto unlink** erzwungen werden, daß die Task beim Neustart des Systems auf jeden Fall abgekoppelt ist.

9.2 Prozeduren

link	Task ankoppeln
on line	Am Terminal
unlink	Task abkoppeln
terminal task	Zugeordnetes Terminal
terminal info	Zeigt Taskstack
auto unlink	Abkoppelung beim Systemstart
max number terminals	Anzahl Terminals
terminal inquiery	Terminal anfordern
begin password	Passwort auf Taskerzeugung
task password	Passwort auf Ankoppeln
terminal broadcast	Nachricht an Terminal senden
terminal menu	Menü der Terminaltask
setup terminal	Terminal neu starten
TASKARRAY	spez. Typ für Terminalstack
get terminal stack	Terminalstack einer Terminaltask

link

PROC link (TASK CONST task)

Wirkung: Es wird versucht, die angegebene Task an das Terminal zu koppeln. Die aufrufende Task bleibt im Terminalstack des Terminals. Falls die Task, in der **link** versucht wird, nicht **on line** ist, passiert nichts.

PROC link

Wirkung: Die Terminaltask wird angekoppelt, ohne die aufrufende Task aus dem Terminalstack zu nehmen.

on line

BOOL PROC on line (TASK CONST task)

Wirkung: Liefert 'TRUE', falls 'task' an der Spitze eines Terminalstacks steht.

BOOL PROC on line

Wirkung: Wie `on line (myself)`.

unlink

PROC unlink

Wirkung: Die aufrufende Task koppelt sich vom Terminal ab.

terminal info

PROC terminal info

Wirkung: In einer temporären Datei wird der aktuelle Stand des Terminalstacks ausgegeben.

terminal task

TASK PROC terminal task (TASK CONST task)

Wirkung: Liefert die Terminaltask in deren Terminalstack 'task' enthalten ist. Es spielt dabei keine Rolle, ob 'task' an der Spitze des Stack steht (**on line** ist). Falls 'task' keinem Terminal zugeordnet ist, wird **niltask** geliefert.

auto unlink

PROC auto unlink (BOOL CONST op):

Wirkung: Tasks, in denen diese Option angewählt ist, werden bei einem Neustart des Systems (normales Starten nach **shutup** oder Rückfall auf Fixpunkt (RERUN)) aus dem Terminalstack entfernt. Auf diese Weise kann erzwungen werden, daß z.B. geschützte Tasks bei Neustart grundsätzlich abgekoppelt sind, so daß eine erneute Passworteingabe nötig wird, um in ihnen zu arbeiten.

Die Voreinstellung für alle Tasks ist `auto unlink (FALSE)`.

BOOL PROC auto unlink (TASK CONST task)

Wirkung: Liefert die Einstellung in der Task 'task'.

BOOL PROC auto unlink

Wirkung: Wie `auto unlink (myself)`

max number terminals

INT PROC max number terminals

Wirkung: Liefert die Anzahl von Arbeitsplätzen, die höchstens für das System konfiguriert werden können. Der Wert ist auf der ersten Systemdiskette eingestellt. Er ist verschlüsselt abgespeichert, so daß eine Änderung der Einstellung nur durch eine passend verschlüsselte und damit lizensierte Systemdiskette 1 vorgenommen werden kann. Der Wert ist natürlich unabhängig von der Anzahl der eingebauten seriellen Schnittstellen oder dergleichen.

terminal inquiery

PROC terminal inquiery (INT CONST timeout)

Wirkung: Die Task wartet 'timeout' Millisekunden darauf, von SYSIO ein Terminal zu erhalten. Auf diese Weise kann auch erzwungen werden, daß die Task an das Terminal gekoppelt bleibt.

PROC terminal inquiery

Wirkung: wie `terminal inquiery (0)`.

Beispiel:

```
PACKET inq beispiel DEFINES bsp:

PROC bsp:
INT VAR i;
FOR i FROM 1 UPTO 10
    REP create (hex(i), dir) PER;
{ Die Prozedur macht irgendwas }
    terminal inquiry (never)
{ wartet dann auf ein Terminal }

FOR i FROM 10 UPTO 15
    REP putline(hex(i)) PER;
    { und dann macht sie was im dialog }

monitor;
END PROC bsp;
END PACKET inq beispiel;

{.. so sieht das in der aufrufenden Task aus, }
{ nachdem 'bsp' insertiert wurde              }
TASK VAR t;

putline ("jetzt das begin");
```

```
begin ("son task 1", PROC bsp, t);
{ start, Sohntask legt Verzeichnisse an }

terminal inquiry ; { ich will erstmal am Terminal bleiben  }

putline ("Sohntask angelegt...");
TEXT VAR tx  := "...................." ;
editget (tx);  { Die Vatertask bleibt am Terminal }

link (/"son task 1");   { jetzt Sohntask ankoppeln und }
unlink;                 { selbst aus Terminalstack raus }
```

begin password

PROC begin password

Wirkung: Es wird ein Passwortdialog geführt. Falls bereits ein Passwort gesetzt ist, wird dieses zunächst abgefragt. Das neue Passwort muß zweimal eingegeben werden. Es wird anschließend immer dann abgefragt, wenn eine neue Task als Nachkomme der Task, in der dieser Dialog geführt wurde, eingerichtet werden soll. Desweiteren ist dieses Passwort direkt **task password** für die neu eingerichtete Task.

begin password (TEXT CONST old, new)

Wirkung: Ändern oder Überprüfen des eingestellten Passworts.

task password

PROC task password

Wirkung: Es wird ein Passwortdialog geführt. Das neue Passwort muß zweimal eingegeben werden. Es wird anschließend immer dann abgefragt, wenn die Task an ein Terminal gekoppelt werden soll.

task password (TEXT CONST old, new)

Wirkung: Ändern oder Überprüfen des eingestellten Passworts.

9.2.1 Meldungen auf dem Bildschirm

Gerade für Aufgaben der Systemverwaltung ist es zweckmäßig, allen oder einzelnen Benutzern des Systems Nachrichten zukommen lassen zu können. **terminal broadcast** bietet die Möglichkeit einzelnen Tasks oder allen Benutzern Texte zu schicken, die temporär in einem passenden Fenster über dem aktuellen Bildschirminhalt ausgegeben werden. Durch Drücken der ESC Taste kann der Benutzer die Nachricht nach Durchlesen verschwinden lassen. Der Empfang solcher Nachrichten läßt sich pro Terminal einstellen. In der Voreinstellung sind für alle Terminals »die Nachrichten eingeschaltet«. Durch die Hotkey Tastenkombination:

`Shift STRG ALT b` Broadcast Ein/Ausschalten

läßt sich für jeden Arbeitsplatz einstellen, ob Nachrichten angezeigt werden sollen oder nicht. Falls der Empfang abgeschaltet ist, wird das im Terminalmenü angezeigt.

terminal broadcast

PROC terminal broadcast (TEXT CONST message, TASK CONST recv, INT CONST timeout)

Wirkung: Die Nachricht 'message' wird in einem passenden Fenster auf dem Bildschirm ausgegeben, wenn die Task 'recv' an ein Terminal gekoppelt ist oder innerhalb des 'timeout' angekoppelt wird.

PROC terminal broadcast (TEXT CONST message)

Wirkung: Die Nachricht 'message' wird an alle Terminals gesendet. SYSIO stellt für jedes Terminal fest, welche Task angekoppelt ist, und stellt dieser Task die Nachricht zu.

terminal menu

PROC terminal menu

Wirkung: Startprozedur für das Terminalmenü einer Terminaltask. Siehe unten.

setup terminal

PROC setup terminal (TEXT CONST terminal name)

Wirkung: Das Terminal 'terminal name' wird neu gestartet. Es werden neue Manager für Tastatur und Bildschirm mit den in *configurator* eingestellten Tabellen gestartet und eine Menütask mit der Prozedur **terminal menu** gestartet. Das Terminal 'terminal name' muß noch nicht existieren (deshalb hat die Prozeduren eine Text- statt des naheliegenden Taskparameters).

Bemerkung: Falls die Prozedur **terminal menu** in *configurator* überinsertiert wurde, wird selbstverständlich diese nachinsertierte Prozedur gestartet.

9.2.2 Behandlung des Terminalstacks

Es kann auch von Interesse sein, innerhalb eigener Programme Informationen über eine Terminaltask zu erhalten. Falls es beispielsweise nötig ist, dem Benutzer eines bestimmten Terminals eine Mitteilung zu senden, so muß zunächst klar sein, welche Task an der Spitze des zu diesem Terminal gehörigen Terminalstacks steht, um dann per **terminal broadcast** die Mitteilung an genau diese Task zu senden.

Der Inhalt eines Terminalstacks wird in Datenobjekte des Typs **TASKARRAY** verwaltet. Ein TASKARRAY ist eine Folge von TaskIDs (interne Taskbezeichner), die dynamisch verwaltet wird. Zu diesem Zweck ist die interne Repräsentation nicht `ROW xy TASK`, sondern TEXT. Dieses ELAN Packet **TASKARRAY** ist auf der Diskette mit ELAN Beispielprogramme enthalten und sei jedem ELAN Interessenten wärmstens empfohlen. Es ist nichttrivial, kurz und zeigt bestens, wie man unter L3 mit ELAN programmiert.

TASKARRAY

TYP TASKARRAY

Wirkung: Datentyp für den Inhalt der Terminaltaskstacks.

get terminal stack

PROC get terminal stack (TASK CONST terminal, TASKARRAY VAR tarray)

Wirkung: Der aktuelle Taskstack der Terminaltask 'terminal'wird in die Variable 'tarray' geschrieben. Das oberste Element des Stacks liegt an letzter Stelle des TASKARRAYs.

Beispiel:

```
TASKARRAY VAR terminal stack;
TASK PROC terminal top (TASK CONST terminal) :
    get terminal stack (terminal, terminal stack);
    terminal stack SUB LENGTH terminal stack
END PROC terminal top;

put (name (terminal top (task ("CONSOLE"))));
```

Bemerkung: Der Operator **SUB** liefert das ite Element eines Taskarray, **LENGTH** die Länge des ARRAYs, also die Anzahl der Elemente. `tarray SUB »letztes Element«` liefert also gerade die Task an der Spitze des Taskstacks. `name (task)` schließlich liefert den Tasknamen.

9.3 Modifikation des Terminalmenüs

Das Terminalmenü stellt die äußerste Schicht des laufenden L3 Systems dar. Es wird durch die Menütask ausgegeben und kontrolliert. Diese Task ist im Normalfall Brudertask zu den Managern für Tastatur und Bildschirm. Um ein anderes Terminalmenü zu erhalten, muß das standardmäßig angebotene entweder ersetzt oder überlagert werden.

Eine vollständige Ersetzung des Menüs kann dadurch erreicht werden, daß in der Task *configurator* eine Prozedur **terminal menu** insertiert wird. Wenn im configurator eine modifizierte Prozedur **terminal menu** insertiert ist, werden anschließend alle neuinstallierten Terminals mit diesem neuen Menü gestartet. Dieses Vorgehen erfordert eine Ersetzung der Task *configurator*. Die dazu nötigen Aktionen sind am Ende dieses Abschnitts beschrieben.

Eine Alternative zu diesem Eingriff ins System ist die Erzeugung von Tasks mit modifiziertem Menü und die Zuordnung dieser Task als Menütask für ausgewählte Terminals. Zur Programmierung des Terminalmenüs sei an dieser Stelle auf das Standardmenü verwiesen, das auf der Diskette mit ELAN Beispielprogrammen enthalten ist.

Um den Terminaltasks neue Terminalmenüs zuzuordnen, sollte man im Systemzweig eine Task einrichten, unter der für alle Arbeitsplätze, denen ein neues Menü zugeordnet werden soll, eine neue Menütask erzeugt wird. Diese Tasks werden dann dem jeweiligen Arbeitsplatz als neue Menütask zugeordnet, indem ein dementsprechender Auftrag an die Terminaltask geschickt wird. Im nachfolgenden Beispiel sei die Ausgangslage, daß in einer eigens dafür erzeugten Task im Systemzweig eine passend gestaltete Prozedur `special terminal menu` insertiert wurde. Um die Terminaltask *oli 1* mit diesem neuen Menü zu starten, sind folgende Aktionen nötig:

Beispiel:

```
PACKET change terminal menu DEFINES
                          change terminal menu :

LET terminal menu code = 0x 06 00 00 11;

PROC change terminal menu (TASK CONST terminal) :

  TASK VAR menu;

  IF exists (terminal)
     THEN begin new terminal menu;
          send new terminal menu;
     ELSE errorstop ("Terminal existiert nicht");
  FI;

  . begin new terminal menu :
    IF exists task (menu name)
       THEN end (task (menu name), quiet)
    FI;
    begin (menu name, PROC special terminal menu, menu);

  . menu name : name (terminal) + ".menu"

  . send new terminal menu :
    INT VAR reply;
    new msg;
    put msg (terminal menu code);
    put msg (menu);
    call manager (terminal, reply);

END PROC change terminal menu;


PROC special terminal menu :

     new menue
        { eigenes Terminalmenü }
END PROC special terminal menu;
```

```
END PACKET change terminal menu;

change terminal menu (task ("oli 1"));
```

Auch die Option ein derart modifiziertes Terminal neu zu starten, sollte an die veränderten Gegebenheiten angepaßt werden. Da ein modifiziertes Terminalmenü wahrscheinlich dazu dient, den »freien« Umgang der Benutzer mit dem System unnötig zu machen, sollte auch die Möglichkeit, ein hängengebliebenes Terminal neu zu starten, für die Benutzer verpackt werden. Der Kern einer solchen Leistung könnte (in der Vatertask des selbstgebauten »Menüzweiges«) so aussehen:

Beispiel:

```
set up terminal ("oli 1");
    { starten }
change terminal menu (task ("oli 1"));
    { eigenes Menü installieren }
```

Austausch der Task configurator

Um das Terminalmenü auszutauschen, kann auch eine passende Task *configurator*, in der ein selbstgestaltetes Terminalmenü insertiert ist, in das System gebracht werden. Dazu muß zunächst eine neue Task als Sohntask von *UTILITIES* begonnen werden. In dieser neuen Task wird dann das neue Terminalmenü insertiert. Bei dieser Vorgehensweise muß diese neue Prozedur **terminal menu** heißen. Anschließend müssen alle Datenräume (Tastaturtabellen etc.) aus der Task *configurator* in die neue Task geholt werden. Anschließend wird die alte Task *configurator* gelöscht un in der neuen Task die Prozedur **configuration manager** aufgerufen. Diese benennt die Task in *configurator* um und startet das gewohnte Menü. Bei Benutzung des ersten Menüpunkts `terminal installieren` wird für das neue Terminal das eigene, neue Terminalmenü gestartet.

Beispiel:

```
begin ("conf", /"UTILITIES");

insert ("new terminal menu");
  { die Datei enthält ein ELAN Packet, das eine neue
    Prozedur 'terminal menu' definiert }

fetchall (/"configurator");
  { Der configurator ist ein 'configuration manager'.
    Im Systemzweig kann man deshalb seine Managerfunktionen
    benutzen. }

end (/"configurator");
  { Da die neue Task 'conf' als Sohntask von UTILITIES
    Systemprivileg hat, kann man beliebige Tasks löschen.
```

```
configuration manager;
  { Die Task, die nun alle benötigten Tabellen enthält,
    wird zum neuen configurator }
```

9.4 Treiber

9.4.1 Einführung

Zum Betrieb von Modems, Druckern oder anderen Geräten, die über parallele oder serielle Schnittstelle an das L3 System angeschlossen werden sollen, gibt es Prozeduren zur Entwicklung eigener Treiberroutinen. Die Benutzung dieser Routinen erspart die »Low Level Programmierung« auf GDP (General Driver Protocol) Ebene.

Eine solche Treiberroutine besteht aus:

- Ankoppeln der gewünschten Schnittstelle,
- Einstellung (nur bei V24)
- Senden/Empfangen von Bytestrings.

9.4.2 Identifikation der Realprozesse

Ein Schnittstellenbaustein mit V24 oder CENTRONICS Ausgang ist im Sinne von L3 ein Gerät. Wie ausführlicher in Kapitel 13 dargestellt ist, werden Geräte in einem L3 Systems durch besondere Tasks bedient. Diese Tasks sind Realprozesse, sie befinden sich permanent im Realspeicher und steuern die ihnen zugeordnete Hardware. Die Treibertasks finden sich im Taskbaum unterhalb der Task *SYSHW*. Aus historischen Gründen sind die Treiber für parallele Schnittstellen noch im Systemkern enthalten und werden erst in einer kommenden L3 Version als Tasks (= ladbare Treiber) realisiert. Es gibt jedoch Pseudonamen für die erste und zweite parallelen Schnittstelle, so daß sie wie Task angesprochen werden können. Sie lauten 'LPT1' bzw. 'LPT2'.

Für serielle Schnittstellen gibt es zwei ladbare Treiber im System. Für die integrierte serielle Schnittstelle und weitere auf »unintelligenten Karten« eingebauten Schnittstellen, gibt es die Task *COM8250*, für Schnittstellen auf »intelligenten Karten« von DIGIBOARD^{TM}, gibt es die *DIGI* Treiber. Da serielle Schnittstellen bidirektional funktionieren, können mit einem Realprozess zwei Pseudonamen assoziiert werden: 'COMx.out' als Ausgabetreibername und 'COMx.in' als Eingabetreibername. 'x' ist dabei die Schnittstellennummer der angesprochenen seriellen Schnittstelle. Falls Ausgabe- und Eingabetreiber auf der ersten seriellen Schnittstelle (COM1) eingerichtet werden sollen, wird COM1 also durch die Pseudonamen 'COM1.out' für den Ausgabetreiber und 'COM1.in' für den Eingabetreiber angesprochen. Entsprechend heißen die Treiber für die Bausteine auf einer intelligenten Karte 'DIGI1.in', 'DIGI1.out' usw.

9.4.3 Zuordnung von Treibern

out driver

PROC out driver(TEXT CONST out driver name)

Wirkung: Der aufrufenden Task wird der mit **out driver name** spezifizierte Realprozess als Ausgabetreiber zugeordnet. Der Aufruf beinhaltet eine Initialisierung, so daß beim Aufruf der Prozedur **open out driver** oder beim ersten folgenden Aufruf der Prozedur **driver out** der Ausgabetreiber geöffnet und, falls es ein V24 Treiber ist, konfiguriert wird.

TEXT PROC out driver

Wirkung: Liefert den Pseudonamen des eingestellten Ausgabetreibers.

in driver

PROC in driver(TEXT CONST in driver name)

Wirkung: Der aufrufenden Task wird der mit **in driver name** spezifizierte Realprozess als Eingabetreiber zugeordnet. Der Aufruf beinhaltet eine Initialisierung, so daß beim Aufruf der Prozedur **open in driver** oder beim ersten folgenden Aufruf der Prozedur **driver in** eine Sohntask als Eingabemanager eingerichtet wird. Dieser Eingabemanager öffnet den Eingabetreiber. Die Konfiguration dieses Treibers muß über den entsprechenden Ausgabetreiber erfolgen.

TEXT PROC in driver

Wirkung: Liefert den Pseudonamen des eingestellten Eingabetreibers.

stream transfer

PROC stream transfer (INT CONST baud, data bits, stop bits, parity)

Wirkung: Setzt den Übertragungsmodus für einen Treiber auf einer seriellen Schnittstelle fest. Beim nächsten Öffnen des eingestellten Ausgabetreibers wird dieser Treiber mit den angegebenen Werten konfiguriert.

Parameter: Einstellbare Werte sind:

Baudrate : 75, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Datenbits: 7 oder 8

Stopbits : 0 , 1 oder 2

Parity : Zur Einstellung der gewünschten Paritybits gibt es folgende Konstanten:

INT CONST no parity
INT CONST odd parity
INT CONST even parity
INT CONST high parity
INT CONST low parity

flow control

PROC dtr dsr flow control (BOOL CONST flag)

Wirkung: 'flag' = TRUE gibt an, daß für den gewählten Ausgabetreiber DTR/DSR Flußkontrolle benötigt wird. Falls die Prozedur vor der Initialisierung durch **driver out** nicht benutzt wird, wird »keine DTR/DSR Flußkontrolle« eingestellt.

PROC rts cts flow control (BOOL CONST flag)

Wirkung: 'flag' = TRUE gibt an, daß für den gewählten Ausgabetreiber RTS/CTS Flußkontrolle benötigt wird. Falls die Prozedur vor der Initialisierung durch **driver out** nicht benutzt wird, wird »keine RTS/CTS Flußkontrolle« eingestellt.

PROC xon xoff flow control (BOOL CONST flag)

Wirkung: 'flag' = TRUE gibt an, daß für den gewählten Ausgabetreiber XON/XOFF Flußkontrolle benötigt wird. Falls die Prozedur vor der Initialisierung durch **driver out** nicht benutzt wird, wird »keine XON/XOFF Flußkontrolle« eingestellt.

9.4.4 Öffnen und Schließen von Treibern

open out driver

PROC open out driver

Wirkung: Der mit **out driver** eingestellte Ausgabetreiber wird für die Task reserviert und wie eingestellt konfiguriert.

close out driver

PROC close out driver

Wirkung: Der mit **out driver** eingestellte Ausgabetreiber wird von der Task freigegeben.

open in driver

PROC open in driver

Wirkung: Der mit **in driver** eingestellte Eingabetreiber wird für die Task reserviert. Dazu wird eine unbenannte Task (Pseudoname -), die als Eingabemanager fungiert, angelegt.

close in driver

PROC close in driver

Wirkung: Der Eingabetreiber wird freigegeben, indem der Eingabemanager gelöscht wird.

9.4.5 Senden/Empfangen

driver out

PROC driver out (TEXT CONST string)

Wirkung: Der Bytesting 'string' wird an den eingestellten Ausgabetreiber geschickt. Falls der Treiber noch nicht konfiguriert ist, wird die Konfigurationsbotschaft vor dem String geschickt.

PROC driver out (TEXT CONST string, INT CONST from)

Wirkung: Wie oben, jedoch wird nur der Teilstring von Position 'from' bis zum Ende von 'string' geschickt.

PROC driver out (TEXT CONST string, INT CONST from, to)

Wirkung: Wie oben, jedoch wird nur der Teilstring von Position 'from' bis zur Position 'to' geschickt.

PROC driver out (TEXT CONST string address,string length)

Wirkung: Angefangen von der Adresse 'string address' werden 'string length' Bytes an den Ausgabetreiber geschickt. Die maximal zulässige Stringlänge beträgt 32 KB.

driver in

PROC driver in (INT CONST timeout, TEXT VAR string)

Wirkung: 'timeout' Millisekunden lang versucht der Inputmanager, einen Bytestring vom Eingabetreiber zu lesen. Wenn 'timeout' abgelaufen ist, ohne daß etwas empfangen wurde, ist die Länge von 'string' 0. Die Größe des Empfangspuffers im Eingabemanager - und somit die maximale Länge für einen in einem Zug zu empfangenden Bytestring - beträgt 4092 Byte. Die Variable 'string' muß im Standarddatenraum liegen. Bei vollem Eingabepuffer vom Eingabetreiber übertragene Botschaften gehen verloren.

PROC driver in (INT CONST timeout, string address, max string length, INT VAR string length)

Wirkung: Wie oben, es wird jedoch explizit die Adresse des Empfangspuffers im eigenen Standarddatenraum angegeben.

Beispiel: Auf der seriellen Schnittstelle 6 (COM6) sollen Daten eines Senders, der mit 9600 Baud, 8 Datenbit, einem Stopbit, even parity und RTS/CTS Protokoll sendet, empfangen und in einer Textdatei protokolliert werden.

```
FILE VAR file := sequential file (output, "protokoll");

in driver  ("COM6.in");
out driver ("COM6.out");
stream transfer  (9600, 8, 1, even parity);
rts cts flow control (TRUE);
open out driver ;

TEXT VAR t :=  "";

REP
   driver in (1000,  t) ;
   IF t <> "" THEN putline (t, file); t := ""; FI
PER;
```

10 Systeminfo/Systemsteuerung

10.1 Einführung

Unter dem Stichwort »Systeminfo« sind Abfragen und Einstellung von systemglobalen Werten, Informationen über alle Tasks und dergleichen beschrieben.

Dazu gehören die Zeitmessung im L3 System, Informationen über das gesamte Tasksystem, Versionsangaben und (privilegierte) Operationen für die Systemsteuerung.

Zeitmessung

DATETIME	TYP
nildatetime	Konstante
cputime	Rechenzeit
datetime	Uhrzeit
systemtime	Systemzeit
local datetime	Tasklokale Uhrzeit
:=, *, +, -	Operatoren
=, <>, <, >, <=,	>= für DATETIME

millisecond
second
minute
hour
day
week
month
year
is leapyear

Konversionen

date
time
dosdatetime

Taskinitialisierung

INITFLAG	TYPE
initialized	Test eines Initialisierungsflags
session	Aktuelle Sitzung

Systemversion

version	Systemversion
kernel info	Version der Systemkernmodule
installation report	Gesamtbeschreibung

Globale Informatione

storage	Speicherplatz des Gesamtsystems
storage info	Speicherplatz des Gesamtsystem
task info	Taskbaum zeigen
inf	— " —

Systemsteuerung

collect garbage blocks	Müllabfuhr
fixpoint	Systemglobaler Fixpunkt
block	Task sperren
unblock	Sperre aufheben
end	Task löschen
halt	Programm in Task unterbrechen
limit	Speicherlimit
prio	Taskpriorität

10.2 Zeitmessung in L3

Da die Zeitmessung in einem Multitasking/Multiuser System für mehrere, teilweise widersprüchliche Zwecke verwendet wird, führt L3 drei unabhängige Uhren.

CPU Zeit: Diese Uhr wird separat für jede Task geführt. Sie liefert die CPU Zeit (echte Rechenzeit), die die Task seit ihrer Erschaffung bis zum Zeitpunkt der Abfrage konsumiert hat. Die Uhr kann nicht verstellt werden. Ihr Stand wird durch **cputime** geliefert.

Echtzeit: Diese Uhr wird systemglobal geführt. Sie liefert die normale Uhrzeit. Nullpunkt dieser Uhr ist der erste Januar des Jahres 1. Die Uhr kann selbstverständlich eingestellt werden. Die Echtzeit kann also Sprünge vorwärts und rückwärts vollführen. Sie ist somit zur Regelung von Wartezeiten und dergleichen ungeeignet. Ihr Stand wird durch **datetime** geliefert. Um in einzelnen Tasks beliebige Uhren »mitführen« zu können gibt es die **local datetime**.

Systemzeit: Diese Uhr läuft mit derselben Geschwindigkeit wie die Echtzeituhr, wenn der Rechner unter Kontrolle von L3 läuft. Der Nullpunkt dieser Uhr liegt beim Zeitpunkt der Systemerzeugung. Danach läuft diese Uhr unveränderlich zur Betriebszeit weiter. Sie kann, da keine Sprünge möglich sind, zur Festsetzung von Pausen und **timeout** Angaben benutzt werden. Ihr Stand wird durch **systemtime** geliefert.

Intern werden die Resultate von Zeitabfragen als doppelt lange Integer (DINT) vom L3 Systemkern geliefert. Für die Behandlung der Echtzeit wird der Datentyp DATETIME zur Verfügung gestellt. Die darauf definierten Operationen bieten das nötige Werkzeug zur Behandlung von Datum und Uhrzeit oder Zeitmessungen. Im Anhang an die Beschreibung der Prozeduren sind ist ein Beispiel gegeben, wie Operationen auf diesem Datentyp eingesetzt werden können.

DATETIME

TYPE DATETIME

Wirkung: Datentyp für Werte der Echtzeituhr.

:=

OP := (DATETIME VAR left, DATETIME CONST right)

Wirkung: Zuweisungsoperator für DATETIME.

nildatetime

DATETIME CONST nil datetime

Wirkung: Liefert den undefinierten Datetime Wert. **nildatetime** kann zur Initialisierung (:=), zum Vergleich (=, <>) und in Botschaften (**put msg**, **get msg**) verwendet werden.

Bei Verwendung von **nil datetime** in anderen Operationen werden folgende Resultate geliefert:

```
DATETIME                           :  nil datetime
BOOL                               :  FALSE
DINT                               :  dint0
Jahr, Monate, Tag                  :  0
Stunde, Minute, Sekunde, Millisek  :  -1
```

cputime

DINT PROC cputime

Wirkung: Liefert den aktuellen Wert der tasklokal verbrauchten Rechenzeit.

datetime

DATETIME PROC datetime

Wirkung: Liefert den aktuellen Wert der Echtzeituhr.

local datetime

PROC local datetime (DATETIME CONST dt)

Wirkung: Stellt tasklokal eine Uhrzeit ein. Auf diese Weise ist es möglich in der Task eine Zeit, die von der Uhr des Rechners verschieden ist, zu führen. Die Voreinstellung ist die für den Rechner gültige Echtzeit. Um eine veränderte Einstellung von **local datetime** wieder auf den Wert der Echtzeit zu bringen kann man `local datetime (nil datetime)` angeben. Danach sind beide Zeiten (wieder) gleich.

DATETIME PROC local datetime

Wirkung: Liefert den aktuellen Wert der tasklokalen Uhrzeit.

systemtime

DINT PROC systemtime

Wirkung: Liefert den aktuellen Wert der unverstellbaren Systemuhr.

Verknüpfungen

DATETIME OP + (DATETIME CONST dt val, DINT CONST period)

DATETIME OP - (DATETIME CONST dt val, DINT CONST period)

DINT OP - (DATETIME CONST first dt, second dt)

DINT OP MOD (DATETIME CONST dt val, DINT CONST period)

DINT OP * (INT CONST fak, DINT CONST period)

Vergleiche

BOOL OP = (DATETIME CONST left dt, right dt)

BOOL OP <> (DATETIME CONST left dt, right dt)

BOOL OP < (DATETIME CONST left dt, right dt)

BOOL OP > (DATETIME CONST left dt, right dt)

BOOL OP <= (DATETIME CONST left dt, right dt)

BOOL OP >= (DATETIME CONST left dt, right dt)

Konstanten

DINT PROC millisecond

DINT PROC second

DINT PROC minute

DINT PROC hour

DINT PROC day

DINT PROC week

millisecond

INT PROC millisecond (DATETIME CONST dt value)

Wirkung: Liefert die Millisekunden des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 0 und 999.

second

INT PROC second (DATETIME CONST dt value)

Wirkung: Liefert die Sekunden des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 0 und 59.

minute

INT PROC minute (DATETIME CONST dt value)

Wirkung: Liefert die Minuten des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 0 und 59.

hour

INT PROC hour (DATETIME CONST dt value)

Wirkung: Liefert die Stunden des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 0 und 23.

day

INT PROC day (DATETIME CONST dt value)

Wirkung: Liefert den Tag des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 1 und 31.

weekday

INT PROC weekday (DATETIME CONST dt value)

Wirkung: Liefert den Wochentag des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 1 (Montag) und 7 (Sonntag).

month

INT PROC month (DATETIME CONST dt value)

Wirkung: Liefert den Monat des Echtzeitwertes 'dt value'. Der Wertebereich des Resultats liegt zwischen 1 und 12.

year

INT PROC year (DATETIME CONST dt value)

Wirkung: Liefert das Jahr des Echtzeitwertes 'dt value'. Das Resultat kann alle INT Werte (auch negative) annehmen, jedoch nicht den Wert 0.

is leapyear

BOOL PROC is leapyear (DATETIME CONST dt value)

Wirkung: Liefert 'TRUE', falls der Datumswert 'dt value' in einem Schaltjahr liegt.

BOOL PROC is leapyear (INT CONST year)

Wirkung: Liefert 'TRUE', falls das Jahr 'year' ein Schaltjahr ist.

Beispiel

Für das Verständnis des Typs Datetime und der darauf definierten Operationen sollte man sich kurz auf den feinen Unterschied zwischen den 'dt value' und den 'period' Werten besinnen. Der erste Wert bezeichnet ein (hochgeradig, bis zur Millisekunde) genaues Datum, der zweite eine Zeitspanne. Man kann also die Zeitspanne zwischen zwei Datumswerten berechen, die gewünschte Größenangabe wird in Einheiten einer Zeitspanne angegeben. Um z.B. zu berechnen, wieviele Stunden die guten Vorsätze im neuen Jahr gehalten haben, gehe man wie folgt vor:

Beispiel:

```
DATETIME VAR nie wieder    := date("01.01.91") + time("15:37"),
             egal ey       := date("04.01.91") + time("20:16");

put ((egal ey - nie wieder) DIV hour)
```

10.3 Konversion von Daten

date

TEXT PROC date (DATETIME CONST dt value)

Wirkung: Liefert einen Text mit dem Datumsanteil von 'dt value' in der Gestalt dd.mm.yy., oder falls das Datum außerhalb dieses Jahrhunderts liegt dd.mm.yyyy. Ein ungültiges Datum (**nildatetime**) wird als `--.--.--` dargestellt.

TEXT PROC date

Wirkung: Kurzform von **date (datetime)**

DATETIME PROC date (TEXT CONST date string)

Wirkung: Der Text 'date string' wird in einen DATETIME Wert konvertiert.

Beispiel:

```
date (" 89 ")      ->      1.1.1989
date ("1989 ")     ->      1.1.1989
date (" 089 ")     ->      1.1.0089
date ("3. 6. 89")  ->      3.6.1989
date (" -- ")      ->      nil datetime
date ("..89 ")     ->      Fehler
```

time

TEXT PROC time (DATETIME CONST dt value, INT CONST stringlength, mmssxxx)

Wirkung: Liefert einen Text mit dem Uhrzeitanteil von 'dt value'. 'stringlength' gibt vor, wie lang das Resultat höchstens sein darf, 'mmssxxx'legt die Genauigkeit fest. Falls das Resultat kürzer als 'stringlength' ist, werden führende Leerzeichen ergänzt, falls die Darstellung nicht möglich ist, wird `stringlength` mal `"*"` ausgegeben.

'mmssxxx' bezeichnet die Genauigkeit:

Beispiel:

```
time (datetime, 2, 0)       ->   12
time (datetime, 5, 2)       ->   12:34
time (datetime, 8, 3)       ->   12:34:56
time (datetime,10, 5)       ->   12:34:56.7
time (datetime,11, 6)       ->   12:34:56.78
time (datetime,12, 7)       ->   12:34:56.789
```

TEXT PROC time

Wirkung: Liefert Stunde, Minute der aktuellen Uhrzeit, entspricht also `time (datetime, 5, 2)`

TEXT PROC time (DATETIME CONST dt value)

Wirkung: Entspricht `time (dt value, 5, 2)`

TEXT PROC time (DINT CONST period, INT CONST stringlength, mmssxxx)

Wirkung: Der als Zeit interpretierte Wert 'period' wird in der Länge 'stringlength' und der Genauigkeit 'mmssxxx' geliefert. Die Stundenangabe kann dabei mehr als zwei Stellen annehmen.

Beispiel:

```
DINT VAR start;
start := cputime ;

(* getestetes Programm *)

put (time (cputime - start, 15, 7));
```

TEXT PROC time (DINT CONST period, INT CONST stringlength)

Wirkung: Wie : `time (period, stringlength, 4)`

TEXT PROC time (DINT CONST period)

Wirkung: Wie : `time (period, 8, 4)`

dos datetime

INT PROC dos datetime (DATETIME CONST dt value)

Wirkung: Konvertiert einen DATETIME Wert in DOS Datum/Uhrzeitformat. Bei der Konversion wird beachtet, daß DOS nur Datum und Uhrzeit zwischen dem 1.1.1980 00:00:00 und dem 31.12.2099 23:59:58 kennt. Der geringeren Präzision der DOS Uhr, die nur gerade Werte für Sekunden kennt, wird Rechnung getragen, indem stets abgerundet wird.

10.4 Initialisierungen in Tasks

Initflag und **Session** sind für die Handhabung größerer Programmsysteme nützlich. **Initflag** erlaubt die Initialisierung von Paketen beim Neueinrichten einer Task. Die Initialisierung wird beim Neueinrichten von Sohntasks automatisch auf FALSE gesetzt, so daß eine Initialisierung genau beim ersten Aufruf einer Prozedur in einer Task erfolgen kann.

Session liefert eine »Sitzungsnummer«, die nach jedem Systemstart verändert wird, unabhängig davon, ob das System vor dem Neustart korrekt abgeschaltet wurde (**shutup**) oder durch zurückgesetzt wurde (RESET).

INITFLAG

TYPE INITFLAG

Wirkung: Deklaration einer Variable des Typs INITFLAG. Variablen dieses Typs werden bei Anlage einer neuen Task systemseitig auf FALSE gesetzt. Somit können in Prozeduren, die von der Vatertask »ererbt« wurden, nötige Initialisierungen durchgeführt werden.

:=

OP := (INITFLAG VAR flag, BOOL CONST flagtrue)

Wirkung: Zuweisungsoperator für die Initialisierung von INITFLAGs

initialized

BOOL PROC initialized (INITFLAG VAR flag)

Wirkung: Liefert den Wert des INITFLAGs flag in der aktuellen Task.

Bemerkung: Wenn INITFLAG in der Task A auf TRUE oder FALSE gesetzt wurde, dann liefert sie beim ersten Aufruf den entsprechenden Wert, danach immer TRUE (in der Task A!). Beim Einrichten von Söhnen wird die Flagge in den Sohntasks automatisch auf FALSE gesetzt. So wird erreicht, daß diese Prozedur in den neu eingerichteten Sohn- und Enkeltasks genau beim ersten Aufruf FALSE liefert.

Beispiel:

```
PACKET stack DEFINES push, pop:

INITFLAG VAR in this task := FALSE ;
INT VAR stack pointer ;
ROW 1000 INT VAR stack ;

PROC push (INT CONST value) :

   initialize stack if necessary ;
   ....

ENDPROC push ;

PROC pop (INT VAR value) :

  initialize stack if necessary ;
  ....

ENDPROC pop .

initialize stack if necessary :
  IF NOT initialized (in this task)
     THEN stack pointer := 1
  FI .

ENDPACKET stack
```

session

INT PROC session

Wirkung: Liefert eine »Sitzungsnummer«. Diese wird automatisch bei jedem Systemstart verändert. Das Bit 24 von session zeigt an, ob das System mit Treibern gestartet wurde.

Bemerkung: Es spielt keine Rolle, ob das System korrekt abgeschaltet worden war, oder ob es zurückgesetzt wurde.

Beispiel:

```
REP
  INT VAR old session := session ;
  WHILE session = old session REP pause (100) PER ;
  putline ("Neuer Systemstart")
PER;

IF on (session, 24) THEN putline ("..mit Treibern..")FI;
```

10.5 Systemversion

Angaben zur Version des Gesamtsystems und der Module des L3 Systemkerns werden durch **version** bzw. **kernel info** ausgegeben.

kernel info

PROC kernel info

Wirkung: In einer temporären Datei werden Name, Erzeugungsdatum und Versionsnummer aller Module des L3 Systemkerns angezeigt.

PROC rts info

Wirkung: Analog für ELAN Laufzeitsystem (RTS).

PROC hardware info

Wirkung: Analog für Rechnerdaten.

version

TEXT PROC version

Wirkung: Liefert die Versionsnummer des L3 Systems.

installation report

PROC installation report

Wirkung: In eine Datei `installation report` werden Informationen zur verwendeten Hardware, Versionen der einzelnen L3 Komponenten etc geschrieben.

10.6 Globale Informationen

storage

PROC storage (INT VAR size, critical, used)

Wirkung: Liefert in den drei Variablen die Größe der L3 Partition ('size'), die Füllgrenze des Systems ('critical') und den aktuell vom System belegten Speicherplatz ('used') auf der Festplatte. Die Angabe erfolgt in Bytes.

Wenn das System größer als 'critical' Bytes wird, wird von der Speicherverwaltung kein Fixpunkt mehr geschrieben. Im Monitordialog erscheint dann die Meldung `Speicherengpass - Dateien löschen!`. Das System führt eine Notmüllabfuhr durch.

Falls vom Archiv eine Datei gelesen werden soll, die das System über die 'critical' Grenze wachsen ließe, so wird die Leseoperation mit der Meldung `Speicherengpaß` abgebrochen.

Bemerkung: Da ein Erreichen oder Überschreiten der 'critical' Grenze tunlichst vermieden werden sollte, kann für Tasks mitttels der Prozedur **limit** eingestellt werden, bei wieviel Prozent »Füllung« des Systems sie zeitweilig suspendiert werden sollen.

storage info

PROC storage info

Wirkung: Gibt im Monitordialog aktuelle Belegung und Größe der L3 Partition aus. Falls die Task nicht an ein Terminal gekoppelt ist, wartet die Ausgabe, bis wieder ein Terminal zugeordnet ist.

task info

PROC task info (INT CONST detailed)

Wirkung: Schreibt Informationen über alle Tasks in eine temporäre Datei und zeigt diese Datei. Durch den Parameter 'detailed' wird eingestellt, welche Angaben geliefert werden.

task info(1) gibt die Namen aller Tasks aus. Durch Einrückung wird die Vater/Sohn Beziehung dargestellt.

task info(2) gibt die Namen aller Tasks aus und für jede Task die bislang verbrauchte CPU Zeit, die Priorität, den Status und das zugeordnete Terminal.

task info(3) liefert Angaben wie task info (2), zusätzlich aber für jede Task noch den belegten Speicherplatz. Da systemweit »faul kopiert« wird, ist die Summe der hier angezeigten Speicherplatzangaben üblicherweise weitaus höher als der tatsächlich belegte Platz auf der Festplatte.

Bemerkung:

CPU Zeit: liefert die reine Rechenzeit der Task. Bei Löschung einer Task wird die von ihr verbrauchte CPU Zeit der Vatertask zugeschlagen.

Priorität: ist ein Wert zwischen 0 und 5. Diese Angabe besagt, jede wievielte prinzipiell zustehende Zeitscheibe an Rechenzeit die Task tatsächlich erhält. Intern wird eine Strategie verfolgt, die Ein-/Ausgabe - intensive Tasks vor rechenintensiven Tasks bevorzugt. Siehe dazu auch Kap. 4.10 Tasks.

Status: der Zustand einer Task kann sein:

– busy –: Task ist aktiv

wait: Task wartet auf Botschaft

dead: Keine Instruktion mehr ausführbar, die Task kann nur noch (von außen) gelöscht werden.

limt: Systemgröße hat limit der Task überschritten.

Terminal: Falls eine Task an ein Terminal gekoppelt ist, wird dessen Name angezeigt.

PROC task info

Wirkung: wie `task info(1)`

PROC task info (INT CONST detailed, TASK CONST root, BOOL CONST bros)

Wirkung: wie oben, jedoch wird das Taskinfo nicht für alle Tasks sondern nur für die Task 'root' und alle Nachkommen dieser Task angezeigt. 'bros' gibt an, ob das Taskinfo auch für die Geschwistertasks von 'root' und deren Nachkommen gezeigt werden soll.

PROC task info (INT CONST detailed, TASK CONST root)

Wirkung: wie `taskinfo (detailed, root, TRUE)`, die Geschwister werden also mit angezeigt.

PROC task info (FILE VAR infofile)

Wirkung: wie oben, `taskinfo (1)` wird aber nicht in einer temporären Datei direkt angezeigt, sondern in die Datei 'infofile' geschrieben.

PROC task info (INT CONST detailed, FILE VAR infofile)

Wirkung: wie oben, `taskinfo (detailed)` wird aber nicht in einer temporären Datei direkt angezeigt, sondern in die Datei 'infofile' geschrieben.

PROC task info (INT CONST detailed, FILE VAR infofile)

Wirkung: wie oben, `taskinfo (detailed)` wird aber nicht in einer temporären Datei direkt angezeigt, sondern in die Datei 'infofile' geschrieben.

PROC task info (INT CONST detailed, TASK CONST root, BOOL CONST bros, FILE VAR infofile)

Wirkung: wie oben, `taskinfo (detailed, root, bros)` wird aber nicht in einer temporären Datei direkt angezeigt, sondern in die Datei 'infofile' geschrieben.

inf

PROC inf

Wirkung: Schreibt, ähnlich **taskinfo(2)**, den Taskbaum mit zusätzlichen Informationen in eine temporäre Datei und zeigt diese an. **inf** zeigt für jede Task die vollständige TaskID, bestehend aus Taskindex und Versionsnummer an und zeigt den Kommunikationspartner der Task an. Die Tasknummern werden in hexadezimaler Darstellung angegeben.

Beispiel:

```
0071 0087  PRINTER   0:01:50  2 wait   -     -
0076 00b0     -      0:01:01  0 wait  0071   -
```

Die (unbenannte) Sohntask von PRINTER (Index 0076, Version 00b0) wartet auf einen Auftrag der Vatertask (Index 0071). PRINTER selbst steht im offenen Warten, nimmt also Aufträge beliebiger Tasks an.

PROC inf (TASK CONST root)

Wirkung: **inf** wird für den Teil des Taskbaums unterhalb der Task 'root' ausgegeben.

PROC inf (FILE VAR file, TASK CONST root)

Wirkung: **inf** wird für den Teil des Taskbaums unterhalb der Task 'root' in das **sequential file** 'file' ausgegeben.

10.7 Systemsteuerung

Die folgenden Operationen haben Wirkung auf das Gesamtsystem, da sie Aktionen des L3 Systemkerns zur Folge haben. Die Prozeduren sind nur im Systemzweig (SYSUR und Nachkommen), also in Task mit **system privilege** verfügbar.

collect garbage blocks

PROC collect garbage blocks

Wirkung: Löst eine außerplanmäßige Gesamtmüllabfuhr aus und wartet deren vollständige Durchführung ab. Die Wirkung der Müllabfuhr tritt erst nach Gültigwerden des darauffolgenden Fixpunkts zutage. Um also Platz im System freizuräumen, ist z.B. in der Task OPERATOR einzugeben:

Beispiel: `collect garbage blocks; fixpoint; unlink`

Wenn OPERATOR sich abkoppelt, wird die Wirkung der Müllabfuhr sichtbar.

fixpoint

PROC fixpoint

Wirkung: Löst einen außerplanmäßigen Fixpunkt aus und wartet dessen vollständige Durchführung ab.

block

PROC block (TASK CONST task)

Wirkung: Die Task wird solange von der Verarbeitung ausgeschlossen, bis die Blockade mit **unblock** aufgehoben wird oder der Task eine Botschaft zugestellt wird. **block** dient dazu, einer Task, die `-busy-` ist, die Rechenzeit zu entziehen. Wenn die Task Ein-/Ausgaben macht, wird die Blockade sofort wieder aufgehoben, wenn die Quittung einer Botschaft empfangen wird oder dergleichen.

unblock

PROC unblock (TASK CONST task)

Wirkung: Die Blockade der Task wird aufgehoben.

end

PROC end (TASK CONST victim)

Wirkung: Nach Kontrollfrage (mit Prozedur **yes**) wird an die Task SUPERVISOR der Auftrag geschickt, die Task 'victim' zu löschen. Der Auftrag wird nur akzeptiert, wenn die auftraggebende Task eine höhere Privilegstufe als 'victim' hat oder 'victim' ein Nachkomme des Auftraggebers ist. Aus dem Systemzweig können also Task des Benutzerzweiges (son(public)) gelöscht werden. Falls der Auftrag akzeptiert wird, beendet SUPERVISOR 'victim' und damit implizit auch alle Nachkommen von 'victim'.

PROC end (TASK CONST victim, QUIET CONST quiet)

Wirkung: Löschen ohne Kontrollfrage.

PROC end

Wirkung: Wie: end (myself)

halt

PROC halt (TASK CONST victim)

Wirkung: An die Task SUPERVISOR wird der Auftrag geschickt, der Task 'victim' einen Fehler zu senden. Die Wirkung in 'victim' entspricht dem Auftreten eines Laufzeitfehler im gerade laufenden Programm. Näheres siehe Kapitel 11.

Der Auftrag wird nur akzeptiert, wenn die auftraggebende Task eine höhere Privilegstufe als 'victim' hat oder 'victim' ein Nachkomme des Auftraggebers ist.

limit

PROC limit (TASK CONST task, INT CONST limit)

Wirkung: 'task' wird von der Speicherverwaltung blockiert, wenn 'limit' Prozent des Speicherplatzes des Systems belegt sind. Diese Einstellung kann nur von Tasks mit Systemprivileg vorgenommen werden!

Fehler: `'limit'-Angabe muß zwischen 1 und 100 liegen`

INT PROC limit (TASK CONST task)

Wirkung: Liefert den eingestellten Wert für 'task'.

prio

PROC prio (TASK CONST task, INT CONST prio)

Wirkung: Die Priorität von 'task' wird auf 'prio' gesetzt. Diese Einstellung kann nur von Tasks mit Systemprivileg vorgenommen werden!

INT PROC prio (TASK CONST task)

Wirkung: Liefert die Priorität von 'task'.

10.8 Wiederherstellung von Systemtasks

Ein »frisches« L3 System besteht aus einer Reihe von Tasks, die die Grundfunktionalität des Systems bereitstellen. Es kann durch Fehler (übrigens fast immer Benutzerfehler, aber Schwamm drüber..), dazu kommen daß solche Systemtasks verlorengehen. Die Tasks lassen sich jedoch meistenteils problemlos wieder einrichten. Für alle Tasks des Standard-Taskbaums ist in diesem Abschnitt angegeben, wie Sie notfalls neu gestartet werden können.

Die folgende Hinweise sind als Hilfe in der Not zu betrachten, denn munteres Löschen von Systemtasks und Experimente in diesen Tasks sind ein wohlfeiler Weg zum Totalschaden.

Standard-Taskbaum

```
SUPERVISOR
    UR
        PUBLIC
    SYSUR
        OPERATOR
            gendos
            kopieren
            fixpoint
                -
            shutup
        UTILITIES
            PRINTER
                -
            SCREEN PRINTER
                -
            configurator
            STREAMER
            POST
            EUMEL
            DOSB
            DOSA
            ARCHIVE B
            ARCHIVE A
        DRIVERS
```

```
        SYSHW
            COM8250
                COM1.out
                COM1.in
            DIGI
                DIGI8.out
                ....
                DIGI1.in
            SO
            hardware configurator
    SYSIO
        meinterminal
            meinterminal.term
            meinterminal.kbd
            meinterminal.dsp
        CONSOLE
            CONSOLE.term
            CONSOLE.kbd
            CONSOLE.dsp
```

SUPERVISOR

Renovierung: Wenn der SUPERVISOR weg ist, haben Sie kein L3 System mehr.

UR

Renovierung: Ebenfalls keine Abhilfe.

PUBLIC

Renovierung: Als Sohntask von UR einrichten, dann: **free manager**.

SYSUR

Renovierung: Siehe UR.

OPERATOR

Renovierung: Als Sohntask von SYSUR einrichten. Anschließend "gendos" etc.

gendos

Renovierung: In Operator: **generate gendos**

kopieren

Renovierung: In Operator: **generate kopieren**

fixpoint

Renovierung: In Operator: **generate fixpoint**

- (Unbenannte Sohntask von fixpoint)

Renovierung: In fixpoint Fixpunktintervall neu einstellen.

shutup

Renovierung: In Operator: **generate shutup**

UTILITIES

Renovierung: nein

PRINTER

Renovierung: Als Sohntask von UTILITIES einrichten. Von Diskette mit Druckeranpassungen passenden Druckertreiber insertieren.

- (Unbenannte Sohntask von PRINTER)

Renovierung: In PRINTER: **start spool**

SCREEN PRINTER

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando : **start screen printer**.

- (Unbenannte Sohntask von SCREEN PRINTER)

Renovierung: Wird bei obigem Kommando erzeugt.

configurator

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando: **configuration manager**. Anschließend müssen Konfigurationstabellen von Diskette geladen werden.

STREAMER

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando: **streamer manager**.

POST

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando: **mail manager**.

EUMEL

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando: **eumel archive manager (/"ARCHIVE A")**. Falls Sie eine zweite Task dieser Funktionalität einrichten wollen, um z.B. auch 3,5" EUMEL Disketten zu lesen, müssen Sie in dieser Task entsprechend das Kommando **eumel archive manager (/"ARCHIVE B")** geben.

DOSA / DOSB

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task **l3dos** starten. Anschließend:

```
net share archive c:\
        l3 cache manager (/"ARCHIVE A")
```

DOSB ist dementsprechend als »cache manager« von ARCHIVE B einzurichten.

ARCHIVE A / ARCHIVE B

Renovierung: Als Sohntask von UTILITIES einrichten. In der neuen Task Kommando: **archive a manager** bzw. **archive b manager**.

DRIVERS

Renovierung: Als Sohntask von SYSUR einrichten. In der neuen Task Kommando: **free manager**.

SYSHW

Renovierung: Als Sohntask von DRIVERS einrichten. In der neuen Task Kommando: **syshw manager**.

COM8250 COM1.out COM1.in

Renovierung: Diese und alle anderen Treiber (**S0**, **DIGI**) werden neu gestartet, wenn das System neu gestartet wird oder wenn der Task **SYSHW** ein HALT zugestellt wird. Natürlich werden die Treiber nur dann neu gestartet, wenn sie in hardware configurator aktiviert sind.

hardware configurator

Renovierung: Wird neu gestartet, wenn ein neuer **syshw manager** erzeugt wird.

SYSIO

Renovierung: Ohne SYSIO ist das System prinzipiell lebendig und faktisch tot.

meinterminal meinterminal.term meinterminal.kbd meinterminal.dsp

Renovierung: Wird durch entsprechende Auswahl in der Task **configurator** neu gestartet.

CONSOLE

Renovierung: Wird neu gestartet, wenn im **configurator** der Punkt »`Tastatur an der CONSOLE einstellen`« gewählt wird. Auch wenn der Systemkern von Diskette gestartet wird und der Punkt »Systemstart ohne Treiber« gewählt wird, wird die CONSOLE neu erzeugt. Das klappt (natürlich) nur, wenn es die Task **configurator** gibt *und* dort die benötigten Umsetztabellen sind.

11 Fehlerbehandlung

11.1 Einführung

Fehler zur Laufzeit treten auf, wenn Operationen (Prozeduren, Programmteile) geforderte Leistungen nicht erbringen können. Dabei sind Ursache und »Tiefe« des aufgetretenen Fehler zunächst unerheblich. Es ist gleichgültig, ob versucht wurde, eine nicht vorhandene Datei zu drucken, ob ein Überlauf bei Subskription auftrat oder ob eine Division durch Null bemerkt wurde. Wenn nachfolgende Programmanweisungen davon ausgehen, daß die gewünschten Leistungen erbracht wurden, ist es nicht sinnvoll, diese Anweisungen dennoch auszuführen. Ein Abbruch einer Operation liegt vor, wenn nach einem Fehler keine Anweisungen mehr ausgeführt werden, sondern die Operation verlassen wird.

Gerade bei Konstruktion komplexer Programmsysteme ist es erforderlich, Fehlerbehandlung innerhalb der Programme vornehmen zu können. Mögliche Fehlersituationen sollen innerhalb der Anwendung behandelt werden, nicht einfach vom Betriebssystem abgefangen werden.

In ELAN Programmen kann durch folgende Maßnahmen ein Abbruch eingeleitet werden:

- Aufruf der Prozedur **errorstop**: Die Operation wird mit einer Fehlermeldung abgebrochen, die man dem Aufruf von **errorstop** als Parameter beifügt.

- Zustellen eines »Fehlers« an die Task, in der das ELAN Programm läuft, von einer anderen Task aus. Durch **halt (/"taskname")** wird der adressierten Task ein Fehler zugestellt. Die Reaktion der Task entspricht dann der Reaktion bei Auftreten eines Laufzeitfehlers, die laufende Operation wird abgebrochen. Dieses **halt** wirkt wie ein **errorstop**, der »von außen« in das Programm induziert wird.

  Da damit ein Eingriff in die Autonomie der Task geschieht, unterliegt diese Art der Fehlerzustellung folgenden Regeln:

  Jede privilegierte Task (SYSUR und Nachfolger) darf jeder anderen Task ein **halt** zustellen und jede Task darf ihren Nachkommen ein **halt** zustellen.

  Da **halt** ein Auftrag an die Task SUPERVISOR ist (siehe auch Supervisor Protokoll), kann die Einhaltung dieser Regeln sicher geprüft werden.

- Vom ELAN Laufzeitsystem oder sogar der Hardware (DIV durch 0) entdeckte Fehler.

Da alle drei Maßnahmen zum Abbruch führen können und somit eine anormale (vorzeitige) Beendigung eines Programms bewirken, werden sie im L3 System als Fehler bezeichnet. Im Prozeßkontrollblock der Task wird nach Aufruf der Prozeduren **errorstop**, **stop** oder **halt** ein Fehlerzustand vermerkt.

Trotz Auftreten eines solchen Fehlers kann es sinnvoll sein, ein laufendes Programm nicht abzubrechen, sondern den Abbruch zu unterdrücken, z.B. wenn

- bestimmte Fehlerfälle vom aufrufenden Programm selbst behandelt werden sollen;

- eine Prozedur die Kontrolle auf jeden Fall behalten soll. Dies ist z.B. beim Monitor notwendig. Gleich welche Fehler vom Monitor gerufene Programme produzieren, der Monitor muß in der Lage sein, die weitere Bearbeitung zu ermöglichen;

- eine Operation nicht unterbrechbar sein darf. Beispielsweise dürfen Programm(teil)e, die Daten transportieren, nicht unterbrochen werden, da sonst ein Verlust dieser Daten eintreten könnte.

Die Prozeduren **enable stop** und **disable stop** bestimmen, ob Operationen im Fehlerzustand weiter bearbeitet oder abgebrochen werden. **Enable stop** legt fest, daß Programmabbrüche durch Fehler zulässig sind, **disable stop** besagt, daß das Programm trotz eines aufgetretenen Fehlers weiterlaufen soll.

11.2 Fehlerbehandlung innerhalb einer Task

Die Reaktion auf Fehler und die Möglichkeiten zum Aufbau von Fängerebenen sollen anhand einiger kleiner Beispiele entwickelt werden:

Beispiel:

```
INT VAR x;
get (x);
...
disable stop;
x := x * x;
...
```

Hier wird mit **disable stop** verhindert, daß ein Abbruch beispielsweise durch `INT-Ueberlauf` auftreten kann. Die Anweisungen nach 'x * x' werden also weiter bearbeitet.

Welchen Wert hat aber nun die Variable 'x', nachdem ein Fehler aufgetreten ist? Wenn das Ergebnis der Operation '*' außerhalb des gültigen Wertebereichs liegt, wird die Operation abgebrochen und liefert in der Regel keinen definierten Wert. Der Wert von 'x' in unserem Beispiel ist also nach einem Fehler bei '*' undefiniert und kann zu Folgefehlern führen, wenn mit diesem Wert weitergerechnet wird.

An diesem Beispiel ist bereits ersichtlich, daß mit der Anwendung der Prozedur **disable stop** äußerst vorsichtig zu verfahren ist, weil u.U. Werte verloren gehen können bzw. mit unerwarteten Werten weitergerechnet wird.

Um zu testen, ob ein Fehler aufgetreten ist, gibt es die Informations-Prozedur **is error** über den Fehlerzustand. Die Prozedur liefert den Wert TRUE, wenn ein Fehler vorliegt, andernfalls FALSE. Die Prozedur **clear error** »löscht« den Fehlerzustand, d.h. anschließende Abfragen mit **is error** liefern FALSE. (Die »richtige« Reaktion auf den Fehler muß ein Programmierer natürlich selbst bestimmen.)

Beispiel:

```
INT VAR x;
get (x);
...
disable stop;
```

```
x := x * x;
IF is error
  THEN put ("'x'-Wert unzulässig");
       x := 0;
       clear error
FI;
...
```

Da die Anweisung **disable stop** in einem Programm so lange gültig bleibt, bis sie aufgehoben wird, würden auch alle folgenden Anweisungen bei eventuellen Fehlern nicht abgebrochen, also auch in Situationen, in denen ein Abbruch erwünscht ist, um Programmierfehler zu erkennen. Deshalb können durch **enable stop** Abbrüche wieder zugelassen werden.

Beispiel:

```
INT VAR x;
get (x);
...
disable stop;
x := x * x;
IF is error
  THEN put ("'x'-wert unzulässig");
       x := 0;
       clear error
FI;
enable stop;
...
```

Durch diese Formulierung würden - wie gewünscht - eventuelle Fehler in den Anweisungen nach **enable stop** zu einem Abbruch führen.

Nicht mit **clear error** gelöschte Fehler führen bei **enable stop** ebenfalls zu einem Abbruch. In dem Programmteil

```
...
disable stop;
x := x * x;
enable stop;
...
```

würde der eventuell auftretende Fehler `INT-Ueberlauf` nicht abgefangen, sondern nur verzögert (nach Ausführung der Prozedur **enable stop**) wirksam, weil er nicht mit **clear error** gelöscht wurde.

Für die Behandlung von Fehlern durch Benutzer gibt es Prozeduren, die eine adäquate Reaktion auf den Fehler erlauben. Mit **error message** können Sie auf die erste Fehlermeldung (eines **error stop**) nach dem letzen **clear error** zugreifen (d.h. Folgefehler verändern nicht die Originalmeldung).

Die Prozedur **error code** liefert den Fehlercode, der bei der Prozedur **errorstop** zusätzlich zum Fehlertext angegeben werden kann (zur Festlegung von Fehlercodes vergl. **errorcode**).

error line liefert die Zeilennummer des zuletzt aufgetretenen Fehlers. Mit **put error** kann eine noch anstehende Fehlermeldung ausgegeben werden.

Beispiel:

```
INT VAR x;
get (x);
...
disable stop;
x := x * x;
IF is error
  THEN IF error message = "INT-Ueberlauf"
         THEN put ("'x'-wert zu gross");
         ELSE put error
       FI;
       clear error
FI;
enable stop;
...
```

Eine Fänger-Ebene ist eine Prozedur, die **disable stop** setzt und dann andere Operationen aufruft. Nach jedem dieser Aufrufe kann eine Fehlerbehandlung mit **clear error** durchgeführt werden. Damit ist gewährleistet, daß Fehler immer von der Fänger-Ebene »aufgefangen« und entsprechend behandelt werden.

Tritt ein Fehler auf, so wird die den Fehler auslösende Operation entweder abgebrochen oder »normal« weiter bearbeitet, je nachdem, ob **enable stop** oder **disable stop** gesetzt ist. Auf jeden Fall wird der Fehlerzustand an die aufrufende Operation weitergemeldet, die wiederum abgebrochen oder weiterbearbeitet werden kann usw. Die Weitermeldung eines Fehlers kann auch über mehrere Stufen erfolgen, solange bis der Fehler gelöscht wird. Das Setzen von **enable stop / disable stop** gilt nicht nur für die aktuelle Operation, sondern auch für aufgerufene Operationen (»Vererbung«). Die aufgerufenen Operationen können allerdings **enable stop / disable stop** neu festlegen.

Beispiel:

```
PROC a:              PROC b:           PROC c:
...                  ...               ROW 10 INT VAR x;
disable stop;        enable stop;      ...
b;                   ...               INT VAR i :: 4711;
IF is error          c;                x [i] := ...;
  THEN ...           ...               ...
     clear error     END PROC b        END PROC c
FI;
enable stop
END PROC a;
```

In der Prozedur »a« wird die Prozedur »b« aufgerufen. Diese ruft wiederum eine Prozedur »c« auf. Für die Prozedur »c« gilt nun der Zustand **enable stop** der Prozedur »b' (Vererbung von enable stop). Tritt jetzt in »c« der Subskriptions-Fehler auf, wird »c' abgebrochen. Die Wirkung der fehlerauslösenden Operation ist nicht definiert.

Da aber auch die Prozedur »b« im enable stop Zustand ist, wird auch die Prozedur »b' abgebrochen. Der Fehler bleibt jedoch erhalten, wird also weitergemeldet. Dies wirkt sich so aus, daß die Anweisung »c« nicht ausgeführt wird. Da die Prozedur »a« **disable stop** gesetzt hat, werden die auf den Aufruf von »b« folgenden Anweisungen durchlaufen und somit durch **clear error** der Fehler gelöscht. In diesem Beispiel "fängt" die Prozedur »a« Fehler auf, die in den Prozeduren »b« und »c« entstehen können.

Ein solcher Fänger wird durch zwei Prozeduren konstruiert. Der eigentliche Fänger (hier: Prozedur »a') ruft eine ausführende Prozedur (hier: »b') im disable stop- Zustand auf. Die aufgerufene Prozedur setzt sofort **enable stop** und führt dann die eigentlichen Aktionen aus. So wird die aufgerufene Prozedur abgebrochen (kann also im Fehlerfall nicht zuviel Schaden anrichten). Der Abbruch führt bis zur Fängerprozedur ('a') hinter den Aufruf der gerufenen Prozedur ('b'). Nach Löschung eventuell auftretender Fehler ist somit sichergestellt, daß der Fänger immer weiterarbeiten kann.

Falls ein Programm kritische Bereiche enthält, in denen mehrere Fehler auftreten können, ohne daß eine sofortige Fehlerbehandlung möglich oder gewünscht ist, so können aufgetretene Fehler durch **push error** auf dem Fehlerstack der Task abgelegt werden und zum Zeitpunkt einer Fehlerauswertung durch **pop error** vom Fehlerstack geholt werden.

Wichtiger Hinweis:

Da im disable stop-Zustand kein Fehler zum Abbruch führt, kann eine Operation in diesem Zustand auch nicht durch **halt** abgebrochen werden. Einerseits ist das für manche Systemteile wünschenswert, andererseits können Operationen, die auf Grund von Programmierfehlern nicht terminieren (Endlosschleifen), nicht unter Kontrolle gebracht werden. Also Vorsicht! (Letztes Mittel: Task löschen)

Es ist nicht (!) garantiert, daß im Fehlerzustand aufgerufene Prozeduren ihre normale Wirkung haben. Garantiert ist dies jedoch für alle Prozeduren und Operatoren, die in diesem Abschnitt aufgeführt werden.

11.3 Prozeduren zur Fehlerbehandlung

clear error	Fehlerzustand löschen
disable stop	Abbruch unterbinden
enable stop	Abbruch zulassen
error code	Liefert gesetzten Fehlercode
error line	Zeilennummer des Fehlers
error message	Fehlermeldung
error stop	Abbruch
is error	Fehlerabfrage
pop error	Fehler vom Fehlerstack holen
push error	Fehler auf Fehlerstack bringen
put error	Fehlermeldung ausgeben
stop	Abbruch
halt	Abbruch von außen

clear error

PROC clear error

Wirkung: Löscht den Fehlerzustand. **is error** liefert anschließend wieder FALSE. **error message**, **error code** und **error line** werden nicht gelöscht.

disable stop

PROC disable stop

Wirkung: Unterbindet den Abbruch in aufgerufenen Operationen. **disable stop** gilt für die Prozedur, in der sie aufgerufen wird und in allen nachfolgend aufgerufenen Prozeduren, es sei denn, sie wird durch **enable stop** außer Kraft gesetzt. Wird die Operation verlassen, in der **disable stop** aufgerufen wurde, wird der »alte« Zustand wiederhergestellt, der vor dem Aufruf der Operation galt. **disable stop** kann weiterhin in einer aufgerufenen Operation durch den Aufruf von **enable stop** in dieser und den folgenden Operationen außer Kraft gesetzt werden.

Bemerkung: INLINE Prozeduren sollten nie **disable stop** aufrufen, da in diesem Falle die Eigenschaft, im **disable stop** zu laufen, für den Aufrufer gilt. Die Fehlerunterdrückung gilt dann wahrscheinlich für eine Prozedur, in der dieser Zustand nicht erwartet ist.

enable stop

PROC enable stop

Wirkung: Setzt die Wirkung eines Aufrufs von **disable stop** zurück. Fehler (**errorstop**, **stop** oder **halt**) in der aktuellen Operation oder den folgenden aufgerufenen Operationen führen zum Abbruch. Bisher nicht gelöschte Fehler (siehe **clear error**) führen sofort zum Abbruch.

error code

INT PROC error code

Wirkung: Liefert den durch **errorstop** gesetzten Fehlercode.

Beispiel:

```
PROC test:
enable stop;
error stop (110, "Dies ist mein Abbruch!");
END PROC test;
...
disable stop;
test;
put (error code);   (* liefert 110 *)
clear error;
enable stop
```

Von L3 vorbelegte Fehlercodes:

0	kein Fehlercode spezifiziert (Standardwert)
1	'halt' vom Terminal
2	Stack-Ueberlauf
3	Heap-Ueberlauf
4	INT-Ueberlauf
5	DIV durch 0
6	REAL-Ueberlauf
7	TEXT-Ueberlauf
8	zu viele DATASPACEs
9	Ueberlauf bei Subskription
10	Unterlauf bei Subskription
11	falscher DATASPACE-Zugriff
12	INT nicht initialisiert
13	REAL nicht initialisiert
14	TEXT nicht initialisiert
15	nicht implementiert
16	Block unlesbar

error line

INT PROC error line

Wirkung: Liefert die Zeilennummmer des Fehlers.

Bemerkung: Voraussetzung für die Möglichkeit, die Zeilennummer auszugeben ist, daß die Übersetzung der Prozedur, die den Fehler erzeugt, im »checkon-Modus« erfolgte.

error message

TEXT PROC error message

Wirkung: Liefert die Fehlermeldung als Text. Anhand dieser Meldung kann entschieden werden, welcher Fehler vorliegt.

Bemerkung: Eine Fehlermeldung "" (also: **error stop ("")**) führt zum Fehlerabbruch mit der Bedeutung "Fehlermeldung wurde bereits ausgegeben". Dementsprechend erfolgt bei der Fehlermeldung 'niltext' keine Reaktion bei **put error**.

error stop

PROC error stop (TEXT CONST message)

Wirkung: Bricht ab und setzt die Zeilennummer (wenn man sich im »checkon-Modus« befindet), in der der Fehler aufgetreten ist, sowie den Text **message**. Der Abbruch kann mit **disable stop** unterbunden werden. **errorstop** hat keine Wirkung, wenn ein noch nicht gelöschter Fehler vorliegt. Zu einer Fehlermeldung "" siehe auch die Prozedur **error message**. Als **errorcode** wird 0 gesetzt.

PROC error stop (INT CONST code, TEXT CONST message)

Wirkung: Analog obiger **errorstop**-Prozedur, aber mit Angabe des Fehlercodes, der durch die Prozedur **error code** in einer Fängerebene erfragt werden kann.

is error

BOOL PROC is error

Wirkung: Informationsprozedur auf das Vorhandensein eines Fehlers.

pop error

PROC pop error

Wirkung: Ein mit **push error** auf dem Fehlerstack abgelegter Fehler wird vom Stack geholt.

Bemerkung: Jeder mit **push error** auf dem Stack abgelegte Fehler muß explizit vom Stack geholt und mit **clear error** gelöscht werden.

push error

PROC push error

Wirkung: Ein Fehlereintrag wird auf dem Fehlerstack abgelegt.

Bemerkung: Ein Fehlereintrag umfaßt Fehlercode, Zeilennummer, falls im »checkon-Modus« übersetzt wurde, und Fehlermeldung. Diese Einträge können nach **pop error** mit den entsprechenden Prozeduren benutzt werden.

Die Kapazität des Fehlerstacks beträgt 1000 Einträge.

put error

PROC put error

Wirkung: Gibt die durch **errorstop** gesetzte Fehlermeldung aus, falls ein Fehler noch nicht gelöscht ist (**error message**).

stop

PROC stop

Wirkung: Abbruch mit Fehlermeldung 'stop'.

halt

PROC halt (TASK CONST task)

Wirkung: Im Prozeßkontollblock der Task 'task' wird das Auftreten eines Fehlers signalisiert. In der Task läuft nun die Fehlerbehandlung an.

Fehler: 'halt' unzulässig Die aufrufende Task ist weder vom Privileg, noch vom Verwandtschaftsgrad her, berechtigt, 'task' anzuhalten.

11.4 Fehlersuche

Die Fehlersuche in ELAN (und möglicherweise auch anderen) Programmen in einem L3 System kennt zwei Schwerpunktthemen. Erstes ist die Überprüfung der Kommunikation zwischen Tasks und damit natürlich zwischen Programmen, also die Protokollierung von Botschaften in Aktion. Das Zweite ist die interne Überprüfung eines Programms, das *klassische* Debugging. Die Beschreibung des Debuggers ist Thema von Abschnitt 11.6.

11.5 Trace

Die Prüfung von Botschaften, aber auch anderer Datenobjekte, in einer Task, die keinen Bildschirmdialog erlaubt, ist mit Hilfe der **trace** Routinen möglich. Damit kann in einer Datei in der Task, oder aber auch in einer zu diesem Behufe angelegten Sohntask der zu prüfenden Task, auf einfache Weise beliebiges protokolliert werden. Insbesondere kann in der Sohntask die Protokollierung von Kontrollereignissen direkt am Bildschirm verfolgt werden.

Falls beispielsweise in einer Managertask die Aufträge protokolliert werden sollen, kann einleitend vor Aufruf der Prozedur **manager** eine Tracetask gestartet werden (**start tracer**). In der eigentlichen Managerprozedur werden dann empfangene Botschaften etc. durch **traceline (hex(msg))** direkt in der Tracetask angezeigt.

Prozeduren

trace	Schalter für trace
trace on	trace aktivieren
trace off	trace deaktivieren
trace	TEXT/INT in trace Datei schreiben
trace dump	Speicherauszug
trace line	Zeile in trace Datei ausgeben
show trace	trace Datei anzeigen
start tracer	Tracetask starten
stop tracer	Tracetask löschen
tracer timeout	Wartezeit einstellen

trace

PROC trace (BOOL CONST flag)

Wirkung: Allgemeine Kontrolle der trace-Operationen. Falls **trace** auf TRUE gesetzt ist, werden folgende traces in eine Protokolldatei "\TRACE1" geschrieben. Falls eine Kontrolltask eingerichtet wurde, wird auch diese Protokolldatei dort geführt. Die Protokolldatei wird beim ersten Aufruf von trace erzeugt. Falls die Datei voll ist (9996 Zeilen), wird eine Datei "\TRACE2" erzeugt. Falls **trace** auf FALSE gesetzt ist, sind die folgenden trace Prozeduren deaktiviert.

BOOL PROC trace

Wirkung: liefert Wert des trace Flag.

trace on

PROC trace on

Wirkung: trace (TRUE).

trace off

PROC trace off

Wirkung: trace (FALSE).

trace

PROC trace (INT CONST value)

Wirkung: Protokolliert 'value', gefolgt von einem Leerzeichen. Der Botschaftsvektor wird vor **trace** gesichert und anschließend wiederhergestellt.

PROC trace (TEXT CONST text)

Wirkung: Protokolliert 'text', gefolgt von einem Leerzeichen. Der Botschaftsvektor wird vor **trace** gesichert und anschließend wiederhergestellt.

trace dump

PROC trace dump (INT CONST addr, length)

Wirkung: Protokolliert ab Adresse 'addr' 'length' Bytes. Der Botschaftsvektor wird vor **trace dump** gesichert und anschließend wiederhergestellt.

PROC trace dump (TEXT CONST string)

Wirkung: Protokolliert 'string' als Textdenoter. Steuerzeichen in 'string' werden decodiert. Der Botschaftsvektor wird vor **trace dump** gesichert und anschließend wiederhergestellt.

trace line

PROC trace line

Wirkung: Zeilenvorschub in Protokolldatei/Bildschirmausgabe.

PROC trace line (TEXT CONST event)

Wirkung: Protokolliert 'event' und macht einen Zeilenvorschub. Falls mehrere Zeilen weniger als 4 Zeichen enthalten, werden sie zusammengezogen.

show trace

PROC show trace

Wirkung: Zeigt die aktuelle Protokolldatei am Bildschirm an.

start tracer

PROC start tracer

Wirkung: Eine Protokolltask namens `name(myself) + ".trace"` wird erzeugt und alle *traces* werden an diese Task geschickt. Die Protokolldatei befindet sich in dieser Task. Wenn die Protokolltask an einen Bildschirm gekoppelt wird, werden die trace Aufträge direkt am Bildschirm angezeigt. Folgende Aktionen sind im Monitordialog mit der Protokolltask zulässig:

`p:` pause. Das Programm wird gestoppt, bis irgendeine Taste gedrückt wird.

`s:` show trace (s.o.)

`m:` monitor. Startet den normalen Monitordialog in der Protokolltask. Durch `exit` wird der Protokollmonitor wieder gestartet.

`u:` unlink

`CursorUp/Down` verlängert/verkürzt die Verzögerung nach Ausgabe einer neuen Zeile.

stop tracer

PROC stop tracer

Wirkung: Löscht die Protokolltask. Protokollausgaben werden nun wieder in einer Datei in der Task selbst gespeichert.

tracer

TASK PROC tracer

Wirkung: Liefert den Taskbezeichner der Protokolltask.

tracer timeout

PROC tracer timeout (INT CONST new timeout)

Wirkung: Stellt die Verzögerung für Ausgaben des Protokollmonitors ein.

INT PROC tracer timeout

Wirkung: Liefert die eingestellte Verzögerung für Ausgaben des Protokollmonitors. Voreingestellt sind 200 Millisekunden.

11.6 Der ELAN Quellcode Debugger

Der ELAN-Quellcode-Debugger (ESD, ELAN Sourcecode Debugger) ist ein Debugger der in Privilegstufe 3 des Prozessors läuft. Folglich bietet er Zugriff nur auf Objekte innerhalb der Task, in der er aktiviert ist.

Der Debugger unterstützt

- die Nutzung der Breakpoint-Register des Prozessors,
- die schrittweise Abarbeitung auf der Hardware-Ebene,
- die schrittweise Abarbeitung auf der Quellcode-Ebene,
- das Setzen von Kontrollpunkten in ELAN-Prozeduren,
- den Zugriff auf jeden Datenraum der eigenen Task,
- die Eingabe symbolischer Adressen.

Der Aufruf des Debuggers erfolgt durch **Exceptions**.

11.6.1 Exkurs: Exceptions

Exceptions sind Unterbrechungen des Programmflusses, die durch den Prozessor selbst hervorgerufen werden. (Im Gegensatz dazu sind Interrupts diejenigen Unterbrechungen, die extern verursacht werden und dem Prozessor über Leitungen gemeldet werden.) Exceptions entstehen entweder durch Fehler, die vom Prozessor im Programmfluß entdeckt werden oder durch Anweisung im Programm (programmierte Exceptions). Standardbeispiele für Fehler sind die Division durch 0 und die beim 80386 allgegenwärtige, weil semantisch schwer beladene, **General Protection Exception**. Eine programmierte Exception ist insbesondere der Befehl `INT 3` (Breakpoint, Opcode `CC`)

11.6.2 Der Aufruf des ESD

Sobald der Prozessor eine Exception entdeckt, wird eine entsprechende Routine (exception handler) aufgerufen. Einige dieser Exceptions haben für den Benutzer keine Bedeutung (z.B. der Zugriff auf den Bildschirmspeicher), andere führen zum Aufruf des ESD. Eine spezielle Exception wird von der ELAN-Prozedur **debug** ausgelöst, die in den INT 3 Befehl übersetzt wird.

Wem das Wort **debug** zu lang ist oder unpassend erscheint, der kann auch **ke** benutzen. »ke« bedeutet »Kontrollereignis« und ist eine Referenz der L3 Entwickler an die Telefunken TR 440, den Rechner, mit dem für sie alles begann. Zugleich ist es nach Meinung des Autors der schlagende Beweis dafür, daß auch ein modernes, durchdesigntes Betriebssystem stets einige bizarre Details enthält, die keinesfalls auf Nachlässigkeit oder dergleichen beruhen.

Bei Aufruf des ESD wird der aktuelle Bildschirminhalt gesichert. Der ESD schaltet explizit in den alphanumerischen Darstellungsmodus um und eröffnet ein Bildschirmfenster.

11.6.3 Aufbau und Handhabung des ESD

In der Kopfzeile des ESD-Fensters wird der Grund für den Aufruf angezeigt. Folgende Anzeigen sind möglich:

- Divide Error
- Single Step
- Debug Exception
- NMI Interrupt
- CC (INT 3)
- INTO Detected Overflow
- BOUND Range Exceeded
- Invalid Opcode
- Coprocessor Not Available

- Double Exception
- PL 3 Exception
- Invalid Task State Segment
- Segment Not Present
- Stack Fault
- General Protection
- Page Fault
- Coprocessor Error

Ein **errorstop** (siehe Kap.12) führt z.Z. nicht zum Aufruf des ESD. Daher können Fehler wie "Text nicht initialisiert" nicht abgefangen werden.

Im Fenster ist fernerhin angezeigt, in welcher Umgebung unterbrochen wurde:

```
Interrupted within DOS

Interrupted within RTS

Interrupted within ELAN-Compiler

Interrupted at  LINE  xxxx
             in PACKET nnnn
```

In der unteren Zeile des ESD-Fensters sind die an dieser Stelle möglichen Kommandos aufgeführt. Einige Funktionen sind immer aufrufbar, auch wenn sie aus Platzmangel nicht explizit aufgeführt sind:

`F8`: Quittieren (verlassen) Zurück zum übergeordneten Menü. Falls diese Taste im Hauptmenü gedrückt wird, wird der ESD verlassen.

`F9`: Wiederholtes Quittieren Rücksprung zum Hauptmenü. Wird diese Taste im Hauptmenü gedrückt, hat sie keine Auswirkung.

`F10`: Monitor Der L3 Monitor wird *aufgerufen.* Mit dem **exit**-Kommando kommt man zurück in den ESD. Zur Erinnerung, daß der Monitor vom ESD aufgerufen wurde, wird die Schachtelungstiefe des Aufrufs angezeigt.

`Shift F5`: Der ursprüngliche Bildschirminhalt wird wiederhergestellt. Auf diese Weise lassen sich Ausgaben des unterbrochenen Programms besichtigen. Ein beliebiger Tastendruck läßt ESD weiterlaufen.

`TAB`: Zweites ESD Fenster Ein zweites ESD Fenster wird eröffnet. Falls es eröffnet ist, kann so zwischen den Fenstern gewechselt werden.

`?`: ESD Hilfe

Beim ESD Aufruf werden folgende Tasten angeboten:

`F1`: Dump; siehe Seite 235

`F2`: Disass; siehe Seite 236

`F3`: Regs; siehe Seite 237

`F4`: Trace; siehe Seite 239

`F5`: View; siehe Seite 239

`F6`: File; siehe Seite 239

`F7`: Config; siehe Seite 235

Eingabe symbolischer Adressen

Wenn ESD die Eingabe einer Adresse erwartet, erscheint:

```
Address:
```

Die Eingabe wird über die ELAN-Prozedur **editget** gelesen, d.h.

- die zuletzt eingegebene Adresse wird angezeigt,
- falls nötig, wird die Eingabe gescrollt,
- häufiger benötigte Adressen können gelernt werden.

Eine zulässige Adreßangabe kann bestehen aus:

- hexadezimaler Adresse `a100`
- Registername(n) `esp`
- Datenraumname `"DOSPROGS/no_use.exe"`
- Datenraumnummer `DS[1]`
- Operatoren `W [4 * 21]`

und daraus zusammengesetzten Ausdrücken.

Leerzeichen haben keine Bedeutung und können außer in den Datenraumnamen überall eingefügt werden. Auf die V86 Segment Register kann nur dann zugegriffen werden, wenn der Interrupt im V86 Modus ausgelöst wurde.

Hinweis: Im 386 Modus sind die Ausdrücke

```
Address: sp
```

und

```
Address: W esp
```

gleichbedeutend. Aus technischen Gründen wird im V86 Modus mit `esp` der 386- Stackpointer angesprochen. Daher werden bei `Address: W esp` die unteren 16 Bit des 80386-Stackpointers als Adresse angesprochen und bei `Address: sp` wird der V86- Stackpointer als Adresse betrachtet.

Beispiele:

Direkte Adressierung

```
Address: 100
```

Die Basisadresse wird auf 0x100 gesetzt.

```
Address: eip
```

Defaultwert. Die Basisadresse wird auf den Wert des EIP-Registers gesetzt.

```
Address: eip + 100
```

Die Basisadresse wird auf den Wert des EIP-Registers plus 0x100 gesetzt.

Indirekte Adressierung

```
Address: [ebp]
```

In ELAN enthält das EBP-Register einen Pointer auf den Prozedur-Stack. Mit indirekter Adressierung wird über das EBP Register die Basisadresse auf den Prozedur-Stack der Vorgängerprozedur gesetzt.

```
Address: [80 0a 00]
```

Die Basisadresse wird auf den Anfang einer ELAN-Prozedur gesetzt. Hier wird sie auf den Anfang des letzten Paketes gesetzt.

```
Address: [ebp+10]
```

In ELAN steht der letzte Parameter auf Adresse `EBP + 0x10`. Die Basisadresse wird nun auf den Parameter selbst gesetzt.

```
Address: [ebp] + 10
```

Jetzt wird die Basisadresse auf die Adresse des letzten Parameters des Vorgängers gesetzt.

Beschränkung von Werten

```
Address: B 128
```

Der Wert wird auf 8 Bit Länge begrenzt. Die Basisadresse wird auf 128 gesetzt.

```
Address: W [4*21]
```

Die Basisadresse wird auf den Offset-Teil (nur 16 Bits) der Adresse 4*21 gesetzt.

```
Address: W [4*21]*10 + W [4*21+2]
```

Die Basisadresse wird auf den Anfang der INT 21 Routine gesetzt.

V86 Pointer

```
Address: P [4*21]
```

Wie oben, die Basisadresse wird auf den Anfang der INT 21 Routine gesetzt.

OFF SEG Adressierung

```
Address: cs:100
```

Die Basis wird nach V86 Art berechnet.

```
Address: P [ss:sp]
```

Angenommen, ein INT xx Befehl wird ausgeführt. Die Basisadresse wird dann auf die Return-Adresse gesetzt.

Um dieses Beispiel noch etwas auszuweiten, nehmen wir an, daß die Interrupt- Routine auf ihren eigenen Stack zeigt. Die alten Werte von SS und SP werden dann in CS:400 und CS:402 gespeichert.

`Address: cs:400` zeigt diese Werte an.

`Address: P [cs:400]` zeigt den Inhalt des alten Stacks, also die Return-Adresse.

`Address: P [ P[cs:400] ] - 2`

Ansprechen von Datenräumen

Mit `Address: "table":1000` greift man auf einen benannten Datenraum zu.

Die Basisadresse wird auf 0x1000 relativ zum Anfang des Datenraumes "table" gesetzt. Solange kein neuer Datenraum spezifiziert wird, wird jede Adresse relativ zum Anfang von "table" interpretiert.

```
Address: [800]
```

Hier wird der Wert auf Adresse 0x800 im Datenraum "table" als neue Basisadresse genommen.

Wenn der Datenraum keinen Namen hat, wie z.B. der Standarddatenraum, kann über die Datenraumnummer auf den Datenraum zugegriffen werden:

```
Address: DS [1]:200 000
```

Jetzt weisen die Adressen auf den Standarddatenraum.

Eine Adresse kann auch einfach aus einem Datenraumnamen bestehen:

```
Address: "table1":
```

Der Wert der aktuellen Adresse bleibt dann unverändert, adressiert aber jetzt einen Speicherplatz im Datenraum "table1". Diese Möglichkeit kann sehr nützlich sein, wenn eine Menge von Datenräumen auf eine spezielle Zeichenfolge an einer festen Adresse untersucht wird.

Fehler:

`invalid`: Syntaxfehler. Ein vertikater Strich zeigt

- auf das erste Zeichen des Identifiers, der den Fehler verursacht hat, oder
- auf einen ungültigen Operator.

`DS does not exist`: Ein Datenraum mit dem angegebenen Namen existiert nicht.

`DSID invalid`: Ein Datenraum mit der angegebenen Datenraumnummer existiert nicht.

`out of bounds`: Der Wert der Adresse überschreitet die Grenzen des Datenraumes.

Bemerkung:

Der Lernmodus des Editors kann genutzt werden, um die Eingabe von Adreßausdrücken zu vereinfachen.

```
<ESC> <HOP> P [ ] <C> LEFT> <RUBIN> <ESC> <HOP> P
```

Mit `<ESC> P` wird der Zeigerausdruck vorbereitet. Eingaben werden sofort zwischen den Klammern eingefügt.

Adreßdarstellung

Intern führt der ESD eine **Basisadresse** und eine **aktuelle Adresse** (Offset). Die Basisadresse ist der Wert der symbolischen Adresse, die zuletzt berechnet wurde. Die aktuelle Adresse besteht aus der Basisadresse und einem Offset. Die Dump-Routine beginnt z.B. mit der Basisadresse. Beim Aufwärtsscrollen im Fenster bleibt die Basisadresse unverändert, während der Offset erhöht wird. Beim Abwärtsscrollen wird der Adreßoffset vermindert. Dadurch können negative Offsets auftreten.

Eine Adresse wird, falls der Offset ungleich Null ist, wie folgt angezeigt:

```
Address: aaaa          Offset: oooo
```

Falls zwei Fenster benutzt werden, hat jedes eigene Basis- und Offsetadresse (siehe auch Seite 239).

11.6.4 Dump

Mit Hilfe der Dump-Funktion kann der Inhalt eines Speicherplatzes gesucht, angezeigt und verändert werden.

Eine Dump-Zeile besteht aus

- einer hexadezimalen Adresse,
- 16 hexadezimalen Zahlen und
- den ASCII-Zeichen, die diesen Zahlen entsprechen, oder einem Punkt.

Der Cursor steht auf der aktuellen Adresse.

Beispiel für einen Speicherauszug (Dump):

```
002c1e00 ff 15 cc 03 80 00 cc-68-  ec d5 ac 00 ff 15 b4 19 .......h........
002c1e10 80 00 68 f4 d5 ac 00 ff   15 20 33 80 00 68 f8 d5 ..h...... 3..h..
002c1e20 ac 00 ff 15 b4 19 80 00   68 fc d5 ac 00 ff 15 b4 ........h.......
002c1e30 19 80 00 68 00 d6 ac 00   ff 15 b4 19 80 00 68 04 ...h..........h.
002c1e40 d6 ac 00 ff 15 b4 19 80   00 ff 35 44 26 80 00 ff ..........5D&...
002c1e50 15 cc 03 80 00 c9 8d 64   24 08 c3 ff 15 b4 19 80 .......d$.......
002c1e60 00 ff 35 44 26 80 00 ff   15 cc 03 80 00 c9 8d 64 ..5D&..........d
```

Die Kommandozeile bietet die folgenden Befehle an:

F1: Addr

F2: Disass

F3: BeginSearch

F4: ContSearch

F5: Stack

Wirkung der Kommandos:

Der Cursor kann beliebig positioniert werden. Eine Adresse unterhalb von 0 oder oberhalb von `pages(ds) * 4096` kann nicht erreicht werden.

HOP: Schreiben

Um versehentliches Schreiben zu verhindern ist die Tastatur gesperrt. Durch diese Taste wird die Sperre aufgehoben, es sind nun Eingaben möglich. Die Eingabe wird durch CR abgeschlossen.

F1: Addr

Liest eine neue Basisadresse. Die aktuelle Adresse entspricht nun der Basisadresse.

`F3:` Search

Eine Zeichenfolge kann beginnend bei der aktuellen Adresse gesucht werden. Bei:

```
Pattern (X:hex, C:char):
```

kann eine Zeichenkette eingegeben werden. Das erste Zeichen muß entweder ein `X` sein, um eine hexadezimale Eingabe zu kennzeichnen, oder ein `C` für die Eingabe von ASCII-Zeichen. Die zuletzt eingegebene Zeichenfolge wird angezeigt und kann ediert werden. Die Eingabe wird beendet mit der Taste CR . Der Cursor wird während der Suche auf das Feld `ContSearch` in der Kommandozeile gesetzt. Zum Lesen der "Zeichenfolge wird die ELAN Prozedur **editget** benutzt.

Die Suche wird beendet, wenn

- die Zeichenfolge gefunden wird. Die Basisadresse wird dann auf das erste Byte der gefundenen Zeichenfolge gesetzt.
- eine Taste gedrückt wird. Bevor eine neue Seite des Datenraums durchsucht wird, prüft ESD, ob eine Taste betätigt wurde. Falls ja, wird die Basisadresse auf das letzte Byte der aktuellen Seite gesetzt.
- die Adresse ungültig wird, d.h. die Datenraumgrenze erreicht ist. Die Basisadresse wird dann auf die letzte gültige Adresse `ds size-1` gesetzt.

`F4:` ContSearch

Die Suche wird mit der zuletzt eingegebenen Zeichenfolge fortgesetzt. Die aktuelle Adresse kann sich inzwischen geändert haben und es können in der Zwischenzeit beliebige andere Aktionen stattgefunden haben (incl. Beenden und Aufrufen des ESD).

11.6.5 Disassembler

Der Disassembler kann sowohl 80386 als auch V86 Befehle decodieren. 80387 Instruktionen werden nicht korrekt decodiert. Die Ausgabe des Disassemblers kann nicht als Eingabe für einen Assembler benutzt werden, da sie eine eigene Syntax hat.

Beispiel für eine Disassemblierung:

```
00201e07  68 ec d5 ac 00     push    '00acd5ec'
002c1e0c  ff 15 b4 19 80 00  call    [008019b4]
002c1e12  68 f4 d5 ac 00     push    '00acd5f4'
002c1e17  ff 15 20 33 80 00  call    [00803320]
002c1e1d  68 f8 d5 ac 00     push    '00acd5f8'
002c1e22  ff 15 b4 19 80 00  call    [008019b4]
002c1e28  68 fc d5 ac 00     push    '00acd5fc'
 Address: eip Offset: 26
```

Die aktuelle Instruktion (`EIP`) ist invers dargestellt.

Die Kommandozeile bietet die folgenden Befehle an:

`F1`: Addr

`F2`: V86 oder 386

`F3`: Regs

`F4`: GoTo

`F5`: GoSub

`F6`: Single Step

`F7`: Breakpoint

Mit den Cursortasten kann beliebig positioniert werden. Die aktuelle Adresse wird entsprechend verändert.

Hinweis: Befehle rückwärts zu dekodieren liefert unsichere Ergebnisse.

`F1`: Address

Liest eine neue Basisadresse. Die Disassemblierung beginnt mit dieser Adresse.

`F2`: V86 oder 386

Wenn `F2` gedrückt wird, wird der Modus umgeschaltet. Der Code wird erneut disassembliert, beginnend mit der aktuellen Adresse.

`F3`: Register

Der Inhalt der allgemeinen Register wird in einem neuen Fenster angezeigt. Falls der Interrupt im V86 Modus ausgelöst wird, werden auch die Segmentregister angezeigt.

Beispiel für einen Registerdump, wenn der Interrupt im 80386 Modus ausgelöst wurde:

```
Registers

EAX = 002bb544    EDI = 00ad3000    EIP = 002c1e07
EBX = 00a01530    ESI = 00ace000    INT = 4e490003
ECX = 000000d0    EBP = 001ff8f4 EFLAGS = 00000216
EDX = 00ad3000    ESP = 001ff8d0     CS = 53430017
```

Beispiel für einen Registerdump, wenn der Interrupt im V86 Modus ausgelöst wurde:

```
Registers

EAX = 0000ff00    EDI = 00000100    EIP = 0000dd9d I SP = 0520  SS = 0dd1
EBX = 00000063    ESI = 00000050    INT = 4e490001 I ES = 0dc1  FS = 00b8
ECX = 00000004    EBP = 001f0084 EFLAGS = 000233c6 I DS = 0dd1  GS = 0016
EDX = 000000a6    ESP = 001ff9e8     CS = 53430088 I CS = 0dd1
```

Wiederum ist die Tastatur gesperrt, um unabsichtliches Schreiben zu verhindern. Durch HOP kann die Sperre entriegelt werden. Mit den Cursortasten läßt sich das gewünschte Register erreichen. Nur eine hexadezimale Eingabe, die mit CR abgeschlossen wurde, ändert den Registerinhalt. Der Schreibschutz wird direkt wieder aktiviert. Die Inhalte der Segmentregister im V86 Modus lassen sich nicht ändern.

F4: GoTo

Nur, wenn das ausgewählte Statement ein jmp-, jcc- oder call-Befehl ist. Die Zieladresse des Befehls wird als neue Basisadresse genommen. Die Disassemblierung wird dann mit der neuen Basisadresse fortgesetzt.

F5: GoSub

Nur, wenn das ausgewählte Statement ein jmp-, jcc- oder call-Befehl ist. Die Zieladresse des Befehls wird als neue Basisadresse einer rekursiv eingebetteten Disassemblierung genommen. Wenn der eingebettete disassemblierte Teil verlassen wird, wird der vorherige Wert der aktuellen Adresse wieder eingestellt.

Hinweis: Die F9 Funktion (wiederholtes Quittieren) verläßt *alle* eingebetteten disassemblierten Teile und bewirkt einen Rücksprung zum Hauptmenü.

F6: Single Step

Das Programm wird schrittweise ausgeführt.

Shift F6: Single Step

Nur, wenn das ausgewählte Statement ein ELAN call-Befehl mit dem Opcode FF 15 ist. Wenn der Aufruf ausgeführt wird, wird die ELAN Prozedur schrittweise auf der Ebene der einzelnen Statements abgearbeitet.

F7: Breakpoint

Hiermit wird ein Breakpoint für die Befehlsausführung gesetzt, wobei die Debug Register des 80386 benutzt werden. Der Befehl wird mit einem B markiert. Wenn bereits ein Breakpoint gesetzt ist, wird dieser gelöscht. Falls kein freies Debug Register mehr verfügbar ist, wird eine Fehlermeldung angezeigt.

Beispiel für einen Codeausschnitt mit Breakpoint:

```
00201e07   68 ec d5 ac 00      push    '00acd5ec'
002c1e0c   ff 15 b4 19 80 00   call    [008019b4]
002c1e12   68 f4 d5 ac 00      push    '00acd5f4'
002c1e17 B ff 15 20 33 80 00   call    [00803320]
002c1e1d   68 f8 d5 ac 00      push    '00acd5f8'
002c1e22   ff 15 b4 19 80 00   call    [008019b4]
002c1e28   68 fc d5 ac 00      push    '00acd5fc'
 Address:  eip Offset: 26
```

11.6.6 Trace, View

Voraussetzung: Eine Quelldatei ist für das aktuelle Paket definiert. Das Fenster des L3 Editors wird in der unteren Hälfte des Bildschirmes geöffnet und zeigt die Quelldatei an. Der Cursor steht in der aktuellen Zeile.

11.6.7 File

In einigen Fällen muß der ESD den Namen der Datei kennen, die den Quellcode eine bestimmten Paketes enthält. Die Verbindung zwischen einer Datei und einem oder mehreren Paketen wird in drei Schritten hergestellt:

1. Auswahl der Eintrittsstelle.
2. Eingabe des Dateinamens.
3. Das Editorfenster wird angezeigt. Dann können die Namen der Pakete eingegeben werden, einer pro Zeile. Die Eingabe wird mit `ESC q` abgeschlossen.

Der ESD überprüft dann, ob die Datei existiert. Jetzt können **View** oder **Trace** benutzt werden.

11.6.8 Config

Hier kann die Einstellung globaler Parameter des ESD verändert werden. Es lassen sich einstellen:

1. Die Größe des ESD Fensters läßt sich auf einen Wert zwischen 7 und 12 Zeilen einstellen. Falls ein zweites Fenster eröffnet wird, erhält es den restlichen Platz des Bildschirms.
2. Standardadreßeinstellung für oberes Fenster.
3. Standardadreßeinstellung für unteres Fenster.
4. Startaktion bei ESD Aufruf. Die Voreinstellung ist `-.-`. Ist eine der Optionen `Dump`, `Regs` oder `Disass` eingestellt, wird nach Aktivierung des ESD das untere Fenster geöffnet und die eingestellte Funktion ausgeführt. Bei `Dump` oder `Disass` wird die Einstellung aus Punkt 3 als Startadresse genommen.

11.7 Das Modell des ELAN-Laufzeit-Systems

11.7.1 Prozeduren, Operatoren, Pakete

Im ELAN System gibt es PACKETs, PROCs (Prozeduren) und OPs (Operatoren). Intern werden Operatoren wie Prozeduren mit der gleichen Anzahl von Parametern behandelt. Das Ergebnis wird als zusätzlicher Parameter betrachtet. Daher werden beide im folgenden mit **Prozedur** bezeichnet. Der Code für ein Paket wird generiert, als sei das Paket eine parameterlose Prozedur mit zwei Ausnahmen:

- Der auf dem Stack reservierte Speicherplatz richtet sich nach dem maximalen Speicherbedarf aller Prozeduren des Paketes
- Paketcode wird nur einmal ausgeführt, und zwar direkt nach der Compilierung. Dieser Vorgang wird Initialisierung des Paketes genannt. Nach der Initialisierung ist der Code frei und wird bei der nächsten Compilation überschrieben.

Parameter werden **by reference** übergeben, das Ergebnis ist der letzte Wert. Für einen Prozeduraufruf generiert der ELAN Compiler den 80386 Befehl `call near, memory indirect`.

Beispiel:

```
push [a030010]
lea  eax, ebp-20
push eax
call [8003f0]
```

11.7.2 Benutzung der Register

Die Register EBP und ESP beschreiben den aktuellen Stack-Status, die anderen Register enthalten Operanden und Adressen. Beim Eintritt in eine Prozedur werden diese Register nicht gesichert.

11.7.3 Der Stack

Der ELAN Prozedurkopf enthält die folgenden Befehle:

```
push   linenr + packetlink SHL 16
push   [ebp+4]
enter  locoff
mov    size,[ebp+6]
```

Der erste `push`-Befehl schreibt ein Doppelwort auf den Stack. Die oberen 16 Bit enthalten einen Verweis auf den Namen des Paketes, zu dem die Prozedur gehört. Die unteren 16 Bit enthalten eine Zeilennummer relativ zum Anfang der Quelldatei.

Der zweite `push`-Befehl kopiert das Statuswort des dynamischen Vorgängers. Die oberen 16 Bit enthalten die Größe des Stacks. Dieser Wert wird durch den `mov`-Befehl ersetzt.

12 L3DOS

12.1 Einführung

L3 bietet die Möglichkeit, eine Ablaufumgebung für Programme, die für die Betriebssysteme MS-DOS/PC-DOS geschrieben sind, bereitzustellen. Diese Umgebung heißt **L3DOS**. L3DOS ist nicht an spezielle Tasks gebunden, sondern kann in jeder Task benutzt werden.

L3DOS basiert auf der Möglichkeit des 80386 Prozessors, daß jede Task, unabhängig von allen anderen Tasks, zwischen virtuellem 8086 (V86) Modus und originärem (Protected) Modus hin- und herschalten kann. Im V86 Modus verhält sich der 80386 wie ein schneller 8086. Um eine geeignete Ablaufumgebung für DOS Programme zu schaffen, wird in einer Task, die in den V86 Modus wechselt, ein Kontrollprogramm aktiviert, das alle Leistungen, die durch ein DOS Programm angefordert werden, im Sinne von L3 umsetzt. Dieses Kontrollprogramm heißt L3DOS.

Die Abbildung von L3DOS auf des Tasksystem eines L3 Systems folgt dem Vorbild eines lokalen Netzwerks. Eine Task entspricht einer Station in einem Netzwerk. Folglich ist Kommunikation zwischen DOS Programmen in Tasks, die sich im V86 Modus befinden, dann möglich, wenn sich diese Programme der unter DOS üblichen Mittel zur Kommunikation bedienen. In den einzelnen Task steht nicht nur eine Emulation von DOS Version 3.3 zur Verfügung, sondern auch die Grundfunktionen des Netzwerkprogramms PC-LAN Version 1.3.

12.2 Der virtuelle DOS Rechner

In der einzelnen Task wird ein virtueller DOS Rechner angelegt.

Das erste Megabyte des Adreßraums der Task bildet den DOS üblichen Arbeitsspeicher des virtuellen Rechners. Dieser Bereich wird grundsätzlich vom Standarddatenraum der Task eingenommen (siehe auch Kap. 1 und 2). Schon diese »örtliche« Anordnung zeigt, daß L3DOS grundsätzlich als normales Benutzerprogramm , wie z.B. der Editor aufzufassen ist.

Standarddatenraum

```
0x00000    0x003ff        Interruptvektoren
0x00400    0x004ff
0x00500    0x005ff        DOS/Basic Datenbereich gefüllt mit '0'
0x00600                   Anfang des DOS Arbeitsspeichers

0x9ffff                   Ende Arbeitsspeicher
0xa0000                   Anfang Bildschirmspeicher
           0xb0000               dos high memory
0xb8000    0xb8fff        Bildschirmspeicher CGA Textmodus

0xc0000    0xcffff               dos high memory

0xd0000    0xdffff               dos high memory, falls EMS
                                        nicht aktiviert
```

```
0xe0000    0xe1a00              dos high memory

0xf6000    0xf9fff          virtuelles BIOS : Tabellen

0xfa000    0xfffff          virtuelles BIOS : Code

0x0010ffff
```

Das (ELAN) Dateisystem der Task wird als Laufwerk C: betrachtet. Alle Dateien, deren Dateinamen den DOS Konventionen gehorchen, sind für L3DOS sichtbar. Dateien, deren Name kein zulässiger DOS Dateiname ist, sind »unsichtbar«. Die Ablage der Dateien ist nicht kompatibel zu DOS. Unter »echtem« DOS werden Dateien typischerweise auf einer Festplatte gespeichert, deren Aufteilung kürzest als Bootbereich, File Allocation Table(s), Wurzelverzeichnis und weitere Verzeichnisse mit den Dateien beschrieben werden kann. Diese Struktur ist unter L3DOS nicht nachgebildet. Programme, die Wissen um diese Strukturen nutzen und unter Umgehung der DOS Funktionsaufrufe z.B. direkt auch die FAT zugreifen möchten, laufen mangels FAT nicht unter L3DOS.

Das Diskettenlaufwerk wird als Station im Netz angesprochen.

12.3 DOS Standarddateien

Bei jedem Neustart von L3DOS werden die Dateien **CONFIG.SYS** und **AUTOEXEC.BAT** aus der Task SYSIO geholt. Das unterbleibt, falls es in der Task bereits Dateien dieses Namens gibt.

12.3.1 Inhalt der AUTOEXEC.BAT

```
@ECHO OFF
SET PROMPT=L3DOS [$P]
SET PATH=C:\;
SET COMSPEC=C:\COMMAND.COM
SET NOSNOWCONTROL=TRUE
SET DOS=L3DOS
DEL AUTOEXEC.BAT
```

12.3.2 Inhalt der CONFIG.SYS

```
FILES=100
BUFFERS=20
```

12.3.3 Zulässige Einträge für CONFIG.SYS

`BREAK = {ON/OFF}` BREAK=ON bewirkt, daß der Control-C Check in der Task eingeschaltet wird. Das bedeutet, daß bei jedem L3DOS Funktionsaufruf geprüft wird, ob diese Tastenkombination gedrückt ist. Falls ja, wird das Programm unterbrochen.

BREAK=OFF bewirkt, daß nur DOS Standardein/-ausgabefunktionen auf die Tastenkombination CONTROL-C reagieren.

`BUFFERS = {n}` n = (1..99). Anzahl der durch L3DOS emulierten Plattenpuffer. Der eingestellte Wert von 20 ist als Information für den Benutzer zu betrachten. Eine Änderung des Wertes wird ignoriert.

`COUNTRY = {001/049}` Umschaltung landesspezifischer DOS Eigenschaften. Der Eintrag 001 bewirkt eine »amerikanische« Ausgabe des Datums. Die Voreinstellung von L3DOS entspricht der deutschen Einstellung 049.

`DEVICE = filename` Derzeit darf nur ANSI.SYS eingestellt werden. Die Wirkung dieser Einstellung entspricht `set ansi (TRUE)`. Die Einstellung anderer Gerätetreiber hat keine Wirkung und wird mit der Meldung: '`load driver xxx *** not yet implemented`' quittiert.

`FCBS = {m,n}` Keine Wirkung.

`FILES = {n}` Keine Wirkung.

`LASTDRIVE = char` Letzter zulässiger Buchstabe für Laufwerksbezeichner.

`SHELL = filename` Die Datei 'filename' wird als Kommandoprozessor geladen und gestartet.

`STACKS = {n,m}` Keine Wirkung.

Alle andere Einträge werden mit der Fehlermeldung quittiert:

```
Unknown command in CONFIG.SYS : xxx
```

12.4 Prozeduren

Die hier zusammengefaßten Prozeduren verändern vor allem Einstellungen von L3DOS und dergleichen. Die eigentlichen L3DOS Kommandos liegen naturgemäß in Form ausführbarer Dateien vor (siehe 12.5).

l3dos	L3DOS »booten«
dos	Zu L3DOS umschalten
assign l3dos printer	Drucker zuordnen
set ansi	Ansi Treiber ein/aus
set ems	EMS Treiber ein/aaus
set extended dos memory	Extended Memory ein/aus
load vbios	Virtuelles BIOS erneuern

l3dos

PROC l3dos

Wirkung: L3DOS wird »gebootet«. Die DOS Emulation wird neu initialisiert. Die Datei **AUTOEXEC.BAT** wird, falls nicht in der Task vorhanden, von SYSIO geholt und ausgeführt. Anschließend wird die Datei **CONFIG.SYS**, falls nicht in der Task vorhanden von SYSIO geholt und ausgewertet. Desweiteren werden die Dateien **L3.COM NET.BAT COMMAND.COM**

aus der Task SYSIO geholt. Anschließend wird der Kommandoprozessor gestartet.

dos

PROC dos

Wirkung: Moduswechsel von ELAN Modus zu V86 Modus und L3DOS. Falls L3DOS in der Task noch nicht initialisiert ist, hat das Kommando keine Wirkung.

assign l3dos printer

PROC assign l3dos printer (TEXT CONST device, TASK CONST printer)

Wirkung: Der DOS Druckerkanal 'device' wird der L3 (Spool)Task 'printer' zugeordnet. DOS kennt als Druckerschnittstelle die Geräte "PRN", "LPT1", "LPT2", "LPT3" und "LPT4". Diese Geräte können verschiedenen Druckertasks (falls vorhanden) zugeornet werden.

Beispiel: `assign l3dos printer ("LPT1", /"PRINTER");`
`assign l3dos printer ("LPT2", /"LASER")`

l3dos printer

TASK PROC l3dos printer (TEXT CONST device)

Wirkung: Liefert die Task, der das Gerät 'device' zugeordnet ist.

Beispiel: `put (name (l3dos printer ("LPT2")))`

Bemerkung: Als Voreinstellung sind alle Geräte der Task "PRINTER" zugeordnet.

set ansi

PROC set ansi (BOOL CONST mode)

Wirkung: Ansi Escape Sequenzen zur Bildschirmausgabe werden ausgewertet. Falls Programme benutzt werden, die die ANSI Sequenzen verwenden, muß `set ansi(TRUE)` eingestellt sein. Diese Einstellung kann auch erreicht werden, wenn in der Datei CONFIG.SYS `DEVICE=ANSI.SYS` eingestellt ist.

set ems

PROC set ems (BOOL CONST mode)

Wirkung: EMS Treiber wird ein/ ausgeschaltet. Bei eingeschaltetem EMS werden im Segment 0xe000 vier EMS Seiten zu je 16KB eingeblendet. Damit ist geeigneten DOS Programmen die Benutzung von maximal 16 MB Expanded Memory nach Spezifikation EMS 3.2 möglich.

set extended dos memory

PROC set extended dos memory (BOOL CONST mode)

Wirkung: Extended Memory wird ein/ ausgeschaltet. Extended DOS Memory bezeichnet den Speicher zwischen 640Kb und 1 MB. Die durch **extended dos memory** zusätzlich allokierten Bereiche sind in 12.2 angezeigt.

load vbios

PROC load vbios (TEXT CONST vb)

Wirkung: In der Task wird der Datenraum 'vb' als neues virtuelles Bios für L3DOS geladen. Auf diese Weise ist es möglich, verbesserte Versionen oder Testversionen dieser Komponente zu installieren.

12.5 L3DOS Kommandos

l3	Zu L3 umschalten
net share	Netzlaufwerk erzeugen
net srv	Netzserver starten
net use	Netzlaufwerk benutzen

l3

l3 L3.COM

Wirkung: Moduswechsel vom V86 Modus in den ELAN Modus. Die Datei wird bei Initialisierung von L3DOS aus SYSIO geholt. Falls die Eingabezeile weiteren Text enthält, der durch ein Leerzeichen von `l3` getrennt ist, so wird dieser Text dem ELAN Compiler zur Analyse und Ausführung übergeben. Auf diese Weise ist es möglich, beliebige (ELAN) Programme von der L3DOS Kommandozeile aus zu geben und Leistungen von L3 aus Batchdateien heraus zu benutzen.

Beispiel:

```
cls
CALL net use a:
l3 reserve (/"DOSA")
A:
```

Bemerkung: Wenn in einer Batchdatei eine andere Batchdatei aufgerufen wird, so muß dieser Aufruf mit CALL erfolgen, da sonst der gesamte Batchablauf nach Abarbeitung der aufgerufenen Datei endet. Batchdateien müssen abschließend eine Leerzeile haben.

NET

NET SHARE

Wirkung: Für andere Tasks wird ein Pfad, der ausgewählte Verzeichnisse oder auch das ganze Dateisystem der Task umfaßt, zur Verfügung gestellt. Um die Benutzung unter L3DOS zu vereinfachen, ist eine passende Batchdatei vorbereitet, die Kommandos folgender Syntax zuläßt:

`NET SHARE [identifier] [path]`

`[identifier]` Name unter dem der Pfad zur Verfügung gestellt werden soll.

`[path]` Pfad der zur Verfügung gestellt werden soll.

Beispiel: `NET SHARE disk c:\`
In der Task "FILES" wird das gesamte Dateiverzeichnis als »Serverplatte« zur Verfügung gestellt.

Achtung: Will eine Task einen Pfad zur Verfügung stellen, so darf der Name der Task nur aus Großbuchstaben bestehen. Nachdem ein Pfad zur Verfügung gestellt wurde, muß die Task mit dem Befehl `NET SRV` vom Terminal abgekoppelt werden.

NET SRV

Wirkung: Die Task, in der zuvor ein Pfad für das virtuelle Netzwerk zur Verfügung gestellt wurde, wird vom Terminal abgekoppelt und wirkt fortan als **free manager**.

Der Name dieser Task muß ausschließlich aus GROSSBUCHSTABEN bestehen!

Bemerkung: Im Netzwerkerslang heißt ein solches Konstrukt »dedizierter Server«.

NET USE

Wirkung: Ein Verzeichnis, das von einem Server zur Verfügung gestellt ist, wird als zusätzliches Laufwerk benutzt. Um die Benutzung unter L3DOS zu vereinfachen, ist eine passende Batchdatei vorbereitet, die Kommandos folgender Syntax zuläßt:

`NET USE [drive] [netpath]`

`[drive]` Laufwerk unter dem der Pfad einer anderen Task angesprochen werden soll.

`[netpath]` Pfad der benutzt werden soll. Er muß die folgende Syntax haben:
\\taskname\identifier

Beispiel: Mit `NET USE D: \\files\disk` kann das gesamte Dateiverzeichnis der Task "FILES" unter dem Laufwerk D: angesprochen werden.

Bemerkung: Mit `NET USE A:` bzw. `NET USE B:` werden den Laufwerken A: bzw B: die Tasks "DOSA" bzw. "DOSB" zugewiesen. Diese Task stellen das gesammte Dateiverzeichnis der Tasks "ARCHIVE A" bzw. "ARCHIVE B" zur Verfügung. Bevor Disketten gelesen oder beschrieben werden können müssen die Tasks mit **reserve (/"DOSA")** bzw. **reserve (/"DOSB")** reserviert werden.

13 Was sind L3 Protokolle?

13.1 Einführung

L3 kennt als systemglobale Teile ausschließlich Prozesse. Es gibt bekanntlich zwei Typen von Prozessen in einem L3 System:

1. Realprozesse. Sie sind dem Systemkern zuzuordnen. Beispiele für Realprozesse sind z.B. Treiber für Bildschirm und Tastatur. Jedes Gerät, wie das Diskettenlaufwerk oder irgendeine Erweiterungskarte wird als Prozeß aufgefaßt. Das Gerät kann demzufolge durch einen internen Namen, einen Taskbezeichner identifiziert und adressiert werden. Realprozesse sind resident, sie unterliegen nicht dem Paging. Sie werden beim Anschalten des Systems gestartet und beim Abschalten beendet.

2. Virtuelle Prozesse. Sie gehören zur Systemschale. Ihre Aufgabe und Ausformung ist unterschiedlich. Beispielsweise sind alle benutzereigenen Tasks, also die Arbeitsumgebungen, virtuelle Prozesse. Auch bestimmte, für das Funktionieren des Systems unverzichtbare Tasks, wie z.B. SYSIO sind virtuelle Prozesse. Sie können jederzeit angelegt und beendet werden, ihre Lebensdauer ist jedoch nicht beschränkt, da sie dem Paging unterliegen.

Zwischen den verschiedenen Prozessen ist Kommunikation in vielfältiger Ausprägung nötig: Das Anlegen einer neuen Task erfolgt durch Senden eines Auftragscodes an den SUPERVISOR. Jede Bildschirmausgabe eines Anwendungsprogramms kommt durch Sendung der auszugebenden Zeichen von der Task an den zugeordneten Bildschirmtreiber zustande. Kommunikation findet bei L3 stets durch Übertragung einer Botschaft von einem Sender an einen Empfänger statt. Eine Botschaft ist ein Meldungscode, ein Auftragscode (eine Bitte) oder ein Objekt (z.B. ein Text oder eine Datei). Die Kommunikation ist synchron, d.h. der Empfänger muß empfangsbereit und -willig sein, bevor der Sender seine Botschaft überstellen kann. Natürlich kann ein Sender probeweise senden; falls sein Adressat jedoch nicht bereit ist, passiert nichts. Desweiteren gilt, daß ein Empfänger nach Erhalt einer Botschaft frei in seiner Reaktion ist. Es werden also nur solche Botschaften eine Reaktion des Empfängers hervorrufen, die den Anforderungen des Empfängers genügen. Kein Empfänger kann gezwungen werden, gegen seinen Willen zu handeln. Sicherheit, Eindeutigkeit und Effizienz der Botschaften wird durch Absprachen über die Gestalt und den Inhalt der Botschaften zwischen den beteiligten (Prozessen) erreicht. Solche Absprachen nennt man **Protokoll**.

L3 kennt eine Reihe von definierten Protokollen zur Unterstützung der L3-typischen Kommunikationsanforderungen.

General Driver Protocol (GDP) Das allgemeine Treiber Protokoll Es beschreibt die Kommunikation zwischen Treiberprozessen und ihren Partnern. Es unterstützt die Realisierung der Basisfunktionen von L3.

Supervisor Protokoll Es beschreibt die Kommunikation zwischen der Task SUPERVISOR und allen anderen Tasks. Mit Hilfe dieses Protokolls werden die globalen Objekte eines L3 Systems, die Tasks verwaltet.

Link Protokoll Es beschreibt die Kommunikation zwischen den Terminaltasks und anderen Tasks. Es dient somit vor allem zur Steuerung der Interaktion.

Dateimanager Protokoll Es beschreibt die Kommunikation zwischen einer als Dateimanager ausgewiesenen Task und ihren Klienten. Es liefert somit Grundfunktionen für die Erfordernisse der Multiuser Betriebs.

All diese Protokolle weisen eine gemeinsame Grundstruktur auf, die ihre Gestalt aus dem Aufbau von Botschaften in L3 gewinnt.

Für jedes Protokoll gilt, daß eine Botschaft stets mit einem Botschaftsabschnitt des Typs Integer beginnt. Diese Zahl ist eine Kennziffer zum Auftakt der Botschaft und setzt sich aus der Codierung des Protokolltyps und einer für diesen Protokolltyp definierten Funktion zusammen. Weitere Botschaftsabschnitte enthalten dann gegebenenfalls weitere Objekte, die als Funktionsparameter zu betrachten sind.

Ausschlaggebend ist jedoch, daß der Empfänger einer Botschaft bereits am ersten (Integer) Abschnitt der Botschaft sicher identifizieren kann, ob er generell als Empfänger in Betracht kommt und ob er eine geforderte Leistung erbringen kann. Protokolle bieten also Sicherheits- und Effizienzgewinn.

13.2 Einteilung

Allen Protokollen gemeinsam ist ein Integerabschnitt als erste Botschaft. Das erste Byte liefert dabei die Protokollidentifikation. Die Bedeutung der weiteren 3 Byte dieser Codierung ist abhängig vom jeweiligen Protokoll.

Codierung	*Protokol*
0x00 xx xx xx	reserviert
0x01 xx xx xx	GDP
0x02 xx xx xx	Supervisor
0x03 xx xx xx	Link Protokoll
0x04 xx xx xx	Dateimanager
0x05 xx xx xx	
0x06 xx xx xx	
...	reserviert für L3
0x3f xx xx xx	
0x40 xx xx xx	
...	Frei für L3
...	Anwendungen
0xff xx xx xx	

Die Codierungen von `0x00000000` bis `0x3fffffff` sind also vollständig für L3 Systemfunktionen reserviert. Falls Sie bei Erstellung irgendwelcher Anwendungen also ein eigenes Protokoll definieren möchten, wählen Sie eine Codierung ab `0x40000000`.

13.3 Das Treiberprotokoll

Das General Driver Protocol (GDP, allgemeines Treiber Protokoll) beschreibt die Kommunikation zwischen einem Prozeß, der als Treiber eines Geräts (z.B. Diskettenlaufwerk) dient, und seinem virtuellen Initiatorprozeß (z.B. der Task ARCHIVE A). Diese Realprozesse bewerkstelligen die Kommunikation zwischen virtueller Systemwelt (Task) und Realwelt (Hardware).

```
GDP Realwelt <---> Realprozeß <---> Systemwelt

Benutzer <---> Tastatur <---> Task
```

Treiberprozesse sind direkt an den L3 Systemkern angelehnt. Sie haben unbeschränkte Ein-/Ausgabe-Privilegien. Jeder Baustein in der Peripherie eines 386ers der der Ein-/Ausgabe dient, hat einen zugehörigen Treiberprozeß. Treiberprozesse können von privilegierten Benutzern ersetzt und das Gesamtsystem kann durch weitere Treiberprozesse ergänzt werden, wenn neue *Geräte* in den Rechner eingebaut werden.

Diktion:

> Ein *Gerät* ist eine Baugruppe mit eigener Funktionalität und eigener Programmierschnittstelle in der Peripherie eines 386er Rechners. Zum Beispiel ist ein 8272 Diskettencontroller mit *allen* angeschlossenen Laufwerken ein Gerät im Sinne dieser Beschreibung, da der 8272 die angeschlossenen Laufwerke nicht unabhängig voneinander steuert. Desgleichen sind auch Grafikkarten, Terminalemulationskarten etc. Geräte.

Das allgemeine Treiberprotokoll beschreibt nun die Gestalt von Botschaften, die zwischen virtuellen Prozessen und Treibern gesendet und empfangen werden. Die Aufgabe eines Treibers ist die korrekte und effektive Umsetzung der Aufträge in Aktionen des von ihm getriebenen Geräts.

Die folgenden Ausführungen beschreiben also zum einen für den Programmierer, der aus einem virtuellen Prozeß heraus Leistungen eines Realprozesses nutzen möchten, welche Botschaften er formulieren kann und muß. Zum anderen gibt dieser Abschnitt auch die Leistungen vor, die ein Treiber für ein Gerät innerhalb des L3 Systems erbringen muß.

13.4 Aufbau der GDP Codierungen

Alle GDP Botschaften haben den Protokolltyp 01. Ein Botschaftsvektor, der eine GDP Botschaft enthält, beginnt mit einer Zahl, deren erstes Byte (das High Byte, Bit 24 bis 31) eine 1 enthält. Desweiteren sind ein eventuell vorhandener *Kanal*, der *Objekttyp* und die *Funktion* codiert.

```
31 ......... 24 23 ............. 16 15 ...................... 4 3 2 1 0

     Typ 01            Kanal              Objekttyp        Funktion
```

Kanäle sind dort vorhanden, wo ein Treiber Aufträge an weitere Treiber verteilt oder mehrere Geräte bedient. Ein Beispiel ist wiederum der Diskettencontroller 8272, der bis zu 4 Disketten- und/oder Streamerlaufwerke steuern kann. Das einzige Gerät aus L3 Sicht ist der Contoller. Er steuert alle physisch vorhandenen Laufwerke.

```
Task        GDP Auftrag                               Kanal         Phys.Gerät

 ARCHIVE A                                                          LAUFWERK 1

 ARCHIVE B   -->      8272 CONTROLLER PROCESS    -->                LAUFWERK 2
```

Der *Objekttyp* spezifiziert die angesprochene Geräteklasse. Da nur bestimmte Operationen auf den jeweils adressierten Gerät sinnvoll sind, sind die möglichen Parameter durch den Objekttyp, also die Art des Gerätes und die Funktion vorbestimmt.

Die *Funktion* beschreibt die Leistung, die erbracht werden soll. Auf der Auftraggeberseite werden auf diese Weise die verschiedenen möglichen Ein-/Ausgabeoperationen beschrieben. Auf der Seite des Auftragnehmers (Treibers) wird entsprechend die Antwort formuliert.

13.5 GDP Codes

Wie oben beschrieben, enthält eine GDP Botschaft grundsätzlich als erste Komponente eine Zahl, die eine vollständige Auftragscodierung darstellt. Die Zahl enthält codierte Information über die GDP Funktion, den Typ der angesprochenen Objekte, sowie ggf. die Angabe des angesprochenen Unterkanals.

Aufbau der Botschaft

```
31 ......... 24 23 ............. 16 15 ..................... 4 3 2 1 0

   Typ 01           Kanal              Objekttyp           Funktion
```

Wertebereiche und Bedeutung:

01: Feste Kennung für GDP Botschaften.

Kanal: 0 bis 0xff. Ein Gerät kann bis zu 256 logische oder physische »Unter«geräte umfassen. Die Adressierung des gewünschten »Untergerätes« - z.B. eines von mehreren an den Diskettencontroller angeschlossenen Laufwerken - erfolgt durch die Angabe des ihm zugeordneten Kanals. Voreinstellung ist der Kanal 0.

Objekttyp: 0 bis 0xfff. Das GDP teilt die diversen möglichen Geräte in verschiedene Klassen und Unterklassen ein. Die Realisierung der geforderten Funktion enthält nur in Zusammenhang mit der angegebenen Geräteklasse eine Bedeutung, da sich z.B. die Aufforderung an den Treiber »Führe eine Ausgabe durch« bei Disketten- und Bildschirmtreibern deutlich unterscheidet.

Funktion: 0 bis 0xf. Elementarfunktionen eines Treibers.

13.5.1 GDP Kanäle

In der Version 1 des Betriebssystems L3 werden Kanäle z.B. innerhalb des Treiberprozesses für das Diskettenlaufwerk benutzt. Das erste Diskettenlaufwerk wird unter Kanal 0, das zweite unter Kanal 1 adressiert.

13.5.2 GDP Funktionen

Die Botschaften zwischen Auftraggeber und Treiber sind im Prinzip von der Gestalt »nimm diese Daten und führe den Auftrag aus«. Sie sind in folgende Funktionen untergliedert:

0	reply
1,2,3	reserviert
4	alternate in order requested
5	alternate out order requested
6,7	reserviert
8	in order
9	out order
10	early in order
11	early out order
12	interrogate in order
13	interrogate out order
14,15	reserviert

reply

Code: `0x01 xx xx x0`

Wirkung: Die Ausführung eines Auftrags oder die Anfrage nach einer durchführbaren Operation wird durch **reply** beantwortet. Zur Antwort reply gehört gegebenenfalls der Inhalt der Antwort. Der Aufbau des Antwortinhalts ist in der Objekttyptabelle für die jeweiligen Objekte angegeben.

alternate in order requested

Code: `0x01 xx xx x4`

alternate out order requested

Code: `0x01 xx xx x5`

Wirkung: Falls ein Treiber einen Auftrag oder eine Anfrage erhält, die für ihn nicht ausführbar ist, fordert er eine Alternative an, die für ihn durchführbar ist. Er liefert einen für ihn eher durchführbaren GDP-Auftrag zurück.

in order

Code: `0x01 xx xx x8`

Wirkung: Damit wird ein Treiber aufgefordert Daten oder Statusinformationen an den Auftraggeber zu senden. Der Treiber schickt sein `reply` nach Ausführung des Auftrags (synchron).

out order

Code: `0x01 xx xx x9`

Wirkung: Dem Treiber werden Daten für eine Ausgabe oder Statusinformationen zugestellt. Der Treiber schickt sein **reply** nach Ausführung des Auftrags (synchron).

early in order

Code: `0x01 xx xx xA`

Wirkung: Der Treiber erhält den Auftrag, Daten oder Statusinformationen an den Auftraggeber zu senden. Im Gegensatz zur **in order** ist dem Treiber jedoch freigestellt, den Auftrag sofort auszuführen oder asynchron zu antworten. Dafür stehen je nach Art des Treibers folgende Möglichkeiten zur Verfügung:

- Treiber mit voller Eigenkontrolle. Dieses sind solche Treiber, die keinen Kontakt zur Außenwelt haben, wie Diskette, Band oder Plattentreiber. Ein solcher Treiber gibt statt eines **reply** mit den geforderten Daten zunächst **reply** mit einer Wartezeit zurück. Dann leitet er die Maßnahmen ein, um die zu liefernden Daten zu gewinnen, z.B. Band starten. Nach Ablauf der Wartezeit erhält der Treiber die nächste **early in order**, die dieselben Daten betrifft. Nun kann er diese liefern oder aber erneut eine Wartezeit vorschlagen.
- Treiber ohne volle Eigenkontrolle. Dieses sind solche Treiber, die Kontakt zur Außenwelt haben, wie Tastatur, RS232 oder LAN Empfangsmodule. Da ein solcher Treiber nicht entscheiden kann, ob er nach Ablauf irgendeiner Zeitspanne Daten hat, muß er entweder sofort vorhandene Daten senden (wie **in order**) der eine leere Antwort **reply-nil object** geben.

early out order

Code: `0x01 xx xx xB`

Wirkung: Der Treiber kann die übergebenen Daten entweder sofort (wie **out order**) oder aber zunächst eine Antwort **early ack** ausgeben und die Ausgabeoperation danach vornehmen (asynchron).

Bemerkung: Falls **early ack** geantwortet wurde, kann natürlich bei der nachfolgenden Ausgabe ein Schreibfehler auftreten. In diesem Fall wird beim nächsten Auftrag eine Fehlerantwort **error status** gegeben. Diese Fehlerantwort kann sich auf mehr als die letzte Ausgabe beziehen. Es sind dann alle Operationen, die seit der letzten synchronen Operation durchgeführt wurden, als fehlerhaft zu betrachten.

interrogate in order

Code: `0x01 xx xx xC`

interrogate out order

Code: `0x01 xx xx xD`

Wirkung: Anfrage an den Treiber, ob er die spezifizierte Operation ausführen kann, und - wenn nein - welche Alternative er anbieten kann. Es werden keine Daten gelesen oder geschrieben. Der Zustand des Treibers bleibt unverändert.

13.5.3 GDP Objekte

Geräte und damit auch der Funktionsumfang ihrer Treiber werden durch ihren Objekttyp beschrieben. Dieser Objekttyp setzt zusammen mit der Funktion fest, wie die GDP Botschaft insgesamt aufgebaut sein muß.

Die Objekttypen sind hierarchisch gegliedert:

11 … 8	7 … 4	3 … 0
`Geräteklasse`	`Objektgruppe`	`Index`

0	.	.	Allgemeine Objekte	
	0	0	Nilobjekt	0x000
		1	Wartezeit	0x001
	1	.	Fehler	
		0	Fehlerstatus	0x010
		1	Fehlerbericht	0x011

1	.	.	Zeichenorientierte E/A	
	0	.	Datenobjekte	
		0	Bytestring	0x100
		1	Datagramm	0x101
	1	.	RS 232	
		0	Übertragungsmodu	s 0x110
		1	Control Line	0x111
		2	Unterbrechung	0x112
		3	Flusskontrolle Ausgabe	0x113
		4	Flusskontrolle Eingabe	0x114
	2	.	LAN	
		0	LAN Adresse	0x120
2	.	.	Blockorientierte E/A	
	0	.	Datenobjekte	
		0	Folge von Blöcken	0x200
	1	.	Diskette	
		0	Übertragungsmodu	s 0x210
		1	Diskettentyp	0x211
		2	Format Disk	0x212
		3	Format Spur	0x213
		4	Deleted Datamark	0x214
	2	.	Festplatte	
		0	Übertragungsmodus	0x220
		1	Format	0x221
		2	Format Record	0x222
	3	.	Streamer	
		0	Blocksequenz	0x230
		6	Filemark	0x236

3	.	.	Eingabegeräte	
	0	.	Tastatur	
		0	Scancode, Taste	0x300
		1	Scancodes, Tasten	0x301
		2	Shiftstatus	0x302
		3	HotKey	0x303
	1	.	Maus	0x310
4	.	.	Ausgabegeräte	
	0	.	Alphanumerisch, 8 bit	
		0	Attribut, Zeichenkette	0x400
		1	Attribut, Zeichenkette + Cursor	0x401
		2	(Attribut + Zeichen)kette	0x402
		3	(Attribut + Zeichen)kette + Cursor	0x403
		4	Fenster füllen	0x404
		5	Fenster scrollen	0x405
		5	Zeichenkette neu	0x406
	4	.	Graphik	
	8	.	Bildschirm	
		0	Beschreibung	0x480
		1	Speicher	0x481
		2	Seite	0x482
		3	Schreibzugriff	0x483
		4	Palette Beschreibung	0x484
		5	Palette Auswählen	0x485
		6	Hardware	0x486
	9	.	Fenster	
		0	Größe	0x490
	A	.	Cursor/Maus	
		0	Beschreibung	0x4a0
	B	.	Font	

5	.	.	Raw IO	
	0	.	über Port	
		0	Byte	0x500
		1	Wort	0x501
		2	Doppelwort	0x502
		3	Quadwort	0x503
		4	Bytestring	0x504
		5	Wortstring	0x505
		6	Doppelwortstring	0x506
		7	Quadwortstring	0x507

Eine vollständige GDP Botschaft besteht aus der GDP Codierung, die das angesprochene Objekt und die gewünschte Funktion spezifiziert, sowie weiteren Datenobjekten, die als GDP Funktionsparameter betrachtet werden können. Dabei spielt natürlich eine Rolle, ob die Botschaft in Zusammenhang mit einer **in order** oder **out order** oder als **reply** gesendet wird.

Dieser Abschnitt beschreibt die für die verschiedenen Geräteklassen (GDP Objekte) nötigen und möglichen Parametrisierungen.

Im folgenden erscheint diese Kennzeichnung:

`IN`: Repräsentation der Botschaft bei **in order, early in order, interrogate in order, alternate in order requested**.

`OUT`: Repräsentation der Botschaft bei **out order, early out order, interrogate out order, alternate out order requested**.

`RP`: Repräsentation der Botschaft bei **reply**.

Bei der Spezifikation der Parameter ist eine ELAN-artige Schreibweise gewählt.

Falls eine Botschaft nur aus dem Integer Abschnitt — also dem GDP Auftrag selbst — besteht, wird der leere Rest der Botschaft durch '`-`' beschrieben.

13.5.4 GDP Objekte und Parameter

Nullobjekt

Code: `0x000`

Parameter: `RP : -`

Wirkung: Bezeichnet den Typ »kein Objekt«. Es wird für einfache (»acknowledge«) Antworten verwendet, die keine weiteren Objekte benötigen.

Wartezeit

Code: `0x001`

Parameter: `RP : INT wartezeit`

Wirkung: Diese Antwort des Treibers kann nur auf eine *early in order* folgen. 'wartezeit' ist eine Angabe in Millisekunden. Nach dieser Zeit sollte noch einmal versucht werden, die Operation auszuführen.

Bytestring

Code: `0x100`

Parameter: `OUT/RP : ROW n BYTE string`
`IN : INT max stringlength`

Wirkung: Die Bytes sind Zeichen innerhalb eines sequentiellen Bytestroms. Sie bilden jedoch kein zusammenhängendes Objekt mit fester Bedeutung. Sie können somit vom Treiber in kleinere Stücke geteilt oder mit anderen Bytestrings zusammengekoppelt werden, ohne daß sich die Semantik des Bytestrings ändert.

Datagram

Code: `0x101`

Parameter: `OUT : ROW n BYTE string`
`IN : INT max stringlength`

Wirkung: Ein Datagramm ist ein Bytestrom, der ein abgeschlossenes Objekt darstellt. Datagramme müssen als unteilbare Objekte behandelt werden.

RS 232 Übertragungsmodus

Code: `0x110`

Parameter: `OUT/RP : INT baud, databits, stopbits, parity`
`IN : -`

Wirkung: Die Angaben übermitteln einem Treiber für eine serielle Schnittstelle die gewünschte neue Einstellung für Übertragungsrate und -Modus.

RS 232 Kontrolle

Code: `0x111`

Parameter: `OUT/RP : INT line mask bits, line value bits`
`IN : -`

Wirkung: Für eine RS 232 Schnittstellen werden eine Leitungsmaske und eine Wertemaske festgelegt. Die Numerierung der Leitungen ergibt sich aus RS 232. (Bit 2 = Leitung 2 usw. Achtung, die Numerierung der Bits in einen Doppelwort ist 0 bis 31, die der Leitungen 1 bis 25!)

Die Leitungsmaske gibt an, welche Leitungen beachtlich sind, die Wertemaske gibt die gewünschte Polarität an. Es gilt:

`line mask bits AND line value bits` legt Leitungen mit 1 Polarität fest,

`line mask bits AND NOT line value bits` legt Leitungen mit 0 Polarität fest.

Beispiel:
```
LET line mask bits = 0x00 10 00 00,
line value bits = 0x00 10 00 00,
control line out = 0x01 00 11 19;
```
Diese Botschaft setzt Leitung 21 auf 1-Polarität

Blocksequenz

Code: `0x200`

Parameter: `OUT/RP : INT blockno, ROW blocklength BYTE blocks`
`IN : INT blockno, INT blocklength`

Wirkung: Blöcke sind der Grunddatentyp für Geräte, die auf magnetischen Speichermedien basieren (Disketten, Bänder..). 'blocklength' muß die in *Transfer Modus* eingestellte Blockgröße oder ein Vielfaches davon sein.

Transfer Modus Diskette

Code: `0x210`

Parameter: `OUT/RP : INT blockno, sectors per track, sector size`
`IN : -`

Wirkung: Einstellung der Laufwerksparameter.

Diskettenformat

Code: `0x211`

Parameter: `OUT/RP : INT formatcode`
`IN :`

Wirkung: Akzeptierte Formate sind:

```
0 : undefiniert
1 : 3,5" oder 5,25" Diskette, DS, SD
2 : 3,5" oder 5,25" Diskette, DS, DD
3 : 5,25" Diskette, DS, HD
4 : 3,5"  Diskette, DS, HD
```

Wenn bei einer **out** Operation der Wert »undefiniert« an den Treiber geschickt wird, so soll der Treiber das Diskettenformat der Diskette im Laufwerk bestimmen und als Einstellung annehmen. Falls das gelingt, sendet er **ack nil**. Der Formattyp kann dann durch eine **in** Botschaft erfragt werden. Falls der Treiber den Formattyp nicht bestimmen kann, weil z.B. keine Diskette im Laufwerk ist, so sendet er **alternate out request**.

13.5.5 Geräte zur Eingabe

Scancode, Taste

Code: `0x300`

Parameter:

Scancodes, Tasten

Code: `0x301`

Parameter:

Shiftstatus

Code: `0x302`

Parameter:

HotKey

Code: `0x304`

Parameter:

13.5.6 Geräte zur Ausgabe/ Alphanumerisch 8 bit

Attribut, Zeichenkette

Code: `0x400`

Parameter: `OUT : INT x, y, attr, ROW n BYTE chars`

Wirkung: Die Bytefolge 'chars' wird an der Position 'x','y' ausgegeben. Die Farbdarstellung aller Zeichen wird durch 'attr' bestimmt (siehe Kap.5: *set attribute*). Der Cursor wird nicht mitgeführt, bleibt also an der Position, die er vor der Ausgabe von 'chars' hatte.

Falls 'chars' »zu lang« ist, also 'x' + 'n' (LENGTH(chars)) > 'width' (siehe *Bildschirmeinstellung*), wird die Ausgabe abgeschnitten.

Attribut, Zeichenkette + Cursor

Code: `0x401`

Parameter: `OUT : INT x, y, attr, ROW n BYTE chars`

Wirkung: Die Bytefolge 'chars' wird ab der Position 'x','y' ausgegeben. Die Farbdarstellung für alle Zeichen wird durch 'attr' bestimmt. Der Cursor wird mitgeführt. Er steht nach der Ausgabe hinter dem letzten Zeichen von 'chars', oder falls dieses Zeichen am Bildschirmrand (meistens x-Position 79) steht, auf dem letzten Zeichen von 'chars'.

Falls 'chars' »zu lang« ist, also 'x' + 'n' (LENGTH(chars)) > 'width' (siehe *Bildschirmeinstellung*), wird die Ausgabe abgeschnitten.

(Attribut + Zeichen)kette

Code: `0x402`

Parameter: `OUT : INT x, y, ROW n BYTE char, attr`

Wirkung: Die Bytefolge 'chars' wird an der Position 'x','y' ausgegeben. Die Farbdarstellung jedes Zeichens 'char' wird durch das nachfolgende Byte 'attr' bestimmt Der Cursor wird nicht mitgeführt, bleibt also an der Position, die er vor der Ausgabe hatte.

Falls 'chars' »zu lang« ist, also 'x' + ('n' DIV 2) (LENGTH(chars)) > 'width' (siehe *Bildschirmeinstellung*), wird die Ausgabe abgeschnitten.

(Attribut + Zeichen)kette + Cursor

Code: `0x403`

Parameter: `OUT : INT x, y, ROW n BYTE char, attr`

Wirkung: Die Bytefolge 'chars' wird an der Position 'x','y' ausgegeben. Die Farbdarstellung jedes Zeichens 'char' wird durch das nachfolgende Byte 'attr' bestimmt Der Cursor wird mitgeführt. Er steht nach der Ausgabe hinter dem letzten Zeichen 'char', oder falls dieses Zeichen am Bildschirmrand (meistens x-Position 79) steht, auf dem letzten Zeichen 'char'.

Falls 'chars' »zu lang« ist, also 'x' + ('n' DIV 2) (LENGTH(chars)) > 'width' (siehe *Bildschirmeinstellung*), wird die Ausgabe abgeschnitten.

Fenster füllen

Code: `0x404`

Parameter: `OUT : INT leftx, lefty, width, height, fillchar`

Wirkung: Das durch die »linke obere Ecke« 'leftx', 'lefty', seine Breite 'width' und seine Höhe 'height' beschriebene Fenster wird mit dem Zeichen 'fillchar' gefüllt, ohne das Attribut zu ändern.

Fenster scrollen

Code: `0x405`

Parameter: `OUT : INT leftx, lefty, width, depth, scroll lines, fillchar, fillattr`

Wirkung: Das durch die »linke obere Ecke« 'leftx', 'lefty', seine Breite 'width' und seine Tiefe 'depth' beschriebene Fenster wird um 'scroll lines' gescrollt. Ein positiver Wert für 'scroll lines' verschiebt den Inhalt des Fensters nach oben. Ein negativer Wert verschiebt nach unten. Das durch Scrolling freiwerdende Fenster wird mit Zeichen 'fillchar' in der Farbdarstellung 'fillattr' gefüllt.

13.6 Das SUPERVISOR Protokoll

Dieses Protokoll beschreibt Botschaften zur und von der Task **SUPERVISOR**. SUPERVISOR ist Verwalter des Tasksystems. Jede Änderung des Systems - wie Löschen oder Neueinrichten einer Task oder Information über das Tasksystem - läuft durch und über den SUPERVISOR ab. Der SUPERVISOR ist die einzige Instanz in einem L3 System, die die internen Prozeßbezeichner kennt. Voraussetzung zur Intertask Kommunikation ist gerade die Kenntnis dieser internen, eindeutigen Namen. Alle in diesem Abschnitt beschriebenen Leistungen stehen in Form von ELAN Prozeduren im ELAN System des Betriebssystems L3 zur Verfügung. Sollen Aktionen, deren Wirkung nicht auf eine Task beschränkt ist, in anderen Programmiersystemen verfügbar sein, so müssen die notwendigen internen Informationen über Prozeßbezeichner zunächst von der Task SUPERVISOR beschafft werden. Der SUPERVISOR hat jedoch nur Kenntnisse über die virtuellen Prozesse eines L3 Systems. Die internen Prozeßbezeichner von Realprozessen (Treiberprozessen) müssen mit Hilfe des Systemcalls Nummer 17 (**sys prc id**) geholt werden.

13.6.1 Die Botschaften des SUPERVISOR Protokolls

Botschaft	Code
begin	0x02 00 00 10
rename	0x02 00 00 11
end	0x02 00 00 12
halt	0x02 00 00 13
system catalogue	0x02 00 00 20
father	0x02 00 00 21
son	0x02 00 00 22
brother	0x02 00 00 23
name	0x02 00 00 24
task	0x02 00 00 25
exists task	0x02 00 00 26
less than	0x02 00 00 27

begin

Code: `0x02 00 00 10`

Parameter: `TEXT taskname INT startaddr DATASPACE stdds`

Wirkung: Eine neue Task wird angelegt. Der externe Name ist 'taskname'. Die Task erhält den Datenraum 'stdds' als Standarddatenraum. Der Befehlszähler für die Task zeigt auf die Adresse 'startaddr' im Standarddatenraum.

rename

Code: `0x02 00 00 11`

Parameter: `TEXT new taskname`

Wirkung: Die Task, die dem SUPERVISOR diesen Auftrag schickt, erhält den Namen 'new taskname'. Mit dieser Umbenennung erhält die Task auch eine neue Versionsnummer! Der alte interne Prozeßbezeichner bezeichnet dann keine existierende Task mehr. Ein **send**- oder **call**-Befehl irgendeiner anderen Task, der noch den alten Prozeßbezeichner — der vor 'rename' galt — benutzt, liefert dann den Antwortcode 1 : 'task not existing'.

end

Code: `0x02 00 00 12`

Parameter: `TASK task`

Wirkung: Der virtuelle Prozeß 'task' wird gelöscht. Damit sind auch alle Datenräume der Task gelöscht und alle Tasks, die Nachkommen der gelöschten Task sind.

Bemerkung: Ein 'end' Auftrag wird vom SUPERVISOR nur akzeptiert, falls er

- von der zu löschenden Task selbst kommt,
- von einer Task die Vorfahre der zu löschenden ist, gegeben wird,
- von einer Task aus dem Systemzweig kommt und nicht eine Task aus dem Systemzweig betrifft, die dem Auftraggeber übergeordnet ist.

halt

Code: `0x02 00 00 13`

Parameter: `TASK task`

Wirkung: Dem virtuellen Prozeß 'task' wird die Fehlermeldung 'halt from terminal' zugestellt.

Bemerkung: Ein 'halt' Auftrag wird vom SUPERVISOR nur akzeptiert, falls er

- von einer Task, die Vorfahre der zu löschenden ist, gegeben wird,
- von einer Task aus dem Systemzweig kommt und nicht eine Task aus dem Systemzweig betrifft, die dem Auftraggeber übergeordnet ist.

system catalogue

Code: `0x02 00 00 20`

Parameter: `TASK task`

liefert: `DATASPACE system catalog`

Wirkung: Die Task SUPERVISOR schickt den aktuellen Systemkatalog an die anfordernde Task.

father

Code: `0x02 00 00 21`

Parameter: `TASK task`

liefert: `TASK vatertask`

Wirkung: Die Task SUPERVISOR liefert den internen Taskbezeichner der Task, die unmittelbarer Vorfahre (Vatertask) von 'task' ist.

son

Code: `0x02 00 00 22`

Parameter: `TASK task`

liefert: `TASK sontask`

Wirkung: Die Task SUPERVISOR liefert den internen Taskbezeichner der Task, die unmittelbarer Nachkomme (Sohntask) von 'task' ist. Falls mehrere Nachkommen existieren, wird der zuletzt angelegte geliefert. Falls keine derartige Task existiert, wird 'niltask' geliefert.

brother

Code: `0x02 00 00 23`

Parameter: `TASK task`

liefert: `TASK brothertask`

Wirkung: Die Task SUPERVISOR liefert den internen Taskbezeichner der Task, die als nächstältere Sohntask unter der Vatertask von 'task' existiert. Falls keine derartige Task existiert, wird 'niltask' geliefert.

name

Code: `0x02 00 00 24`

Parameter: `TASK task`

liefert: `TEXT tasks name`

Wirkung: Die Task SUPERVISOR liefert einen Text mit dem externen Tasknamen von 'task'. Falls 'task' nicht existiert, wird der Fehler "'task' gibt es nicht" geliefert.

task

Code: `0x02 00 00 25`

Parameter: `TEXT taskname`

liefert: `TASK task`

Wirkung: Die Task SUPERVISOR liefert den internen Taskbezeichner von 'taskname'. Falls es keine Task dieses Namens gibt, wird der Fehler "'taskname' gibt es nicht" geliefert.

exists task

Code: `0x02 00 00 26`

Parameter: `TEXT taskname`

liefert: `INT flag`

Wirkung: Die Task SUPERVISOR liefert '0', falls es eine Task 'taskname' gibt, sonst '1'.

less than

Code: `0x02 00 00 27`

Parameter: `TEXT task1, task2`

liefert: `INT flag`

Wirkung: Die Task SUPERVISOR liefert '0', falls die Task mit dem Namen 'task2' Vorfahre von 'task1' ist, sonst '1'.

13.7 Das Link Protokoll

Das Link Protokoll des Betriebssystems L3 umfaßt die Bedingungen und Vorschriften für die Verbindung von Terminals und Tasks. Mit dem Begriff »Terminal« ist im weiteren ein aus Tastatur und Bildschirm bestehender Arbeitsplatz an einem L3 System gemeint. Es wird nicht zwischen der CONSOLE und weiteren Arbeitsplätzen unterschieden.

Terminals werden in einem L3 System durch die Treibermanager (`xy.kbd` für die Tastatur und `xy.dsp` für den Bildschirm) repräsentiert. Ein Treibermanager ist die Schnittstelle zu einem Realprozeß. Der Realprozeß ist der »wirkliche« Hardwaretreiber. Die Manager für Tastatur und Bildschirm sind wiederum Sohntasks einer Terminalmanagertask. Die Hierarchie für ein Terminal sieht also wie folgt aus:

```
TERMINAL 1                              (* Terminalmanager *)
    TERMINAL 1.term                     (* Terminalmenü (austauschbar) *)
    TERMINAL 1.kbd                      (* virt. Treiber *)
        SYSPRC TERMINAL 1.keyboard driver (* Realprozeß (Treiber) *)
    TERMINAL 1.dsp
        SYSPRC TERMINAL 1.display driver
```

Alle Terminals eines Systems sind Sohntasks von SYSIO, der Wurzel des Terminalzweiges innerhalb des Gesamtsystems.

Die Hierarchie aller Terminals eines Rechners stellt sich bei **taskinfo** wie folgt dar:

```
SYSIO
         ...
        TERMINAL 1
            TERMINAL 1.term
            TERMINAL 1.kbd
            TERMINAL 1.dsp
        CONSOLE
            CONSOLE.term
            CONSOLE.kbd
            CONSOLE.dsp
```

Ein Terminal in Gestalt seiner Treiberprozesse ist grundsätzlich mit irgendeiner Task verbunden. Diese Task ist »Besitzer« des Terminals. Die Ausgaben des Besitzers werden auf den Bildschirm geschrieben, Tastatureingaben an einem Terminal werden dem Besitzer zugestellt. Der Ausgangszustand ist, daß Besitzer eines Terminals der jeweilige Terminalmanager ist. Für die Koppelung eines Terminals an eine Task gilt:

- Der Anstoß zur Koppelung eines Terminals an eine Task kann nur vom Besitzer kommen. Der Besitz wird also weitergereicht.
- Wenn ein Besitzer das Terminal an eine andere Task weiterreicht, bleibt seine Ankoppelung bestehen. Er kann jedoch nicht mehr mit den Treibern kommunizieren. Die Task wird in einen sogenannten Terminalstack geschrieben. Das hat zur Folge, daß eine Task, die durch ein Paßwort geschützt ist, nur beim ersten Ankoppeln die Eingabe des Paßwortes verlangt. Solange sie an das Terminal gekoppelt bleibt, bzw. im Terminalstack steht, kann sie ohne erneute Paßworteingabe wieder aufgenommen werden.
- Wenn ein Besitzer das Terminal abgibt, ohne es weiterzureichen, erhält der vorherige Terminalbesitzer das Terminal zurück. Die Task wird nicht in den Terminalstack aufgenommen.
- Wenn der <**HotKey**> (Tasten `Strg` `Alt` `Del`) beim Besitzer betätigt wird, wird der Terminalmanager neuer Besitzer.
- Der Terminalmanager kann angekoppelte Prozesse anhalten. (HALT vom Terminal).

- Das Koppelungsmanagement wird von der Task SYSIO bewerkstelligt. Der Besitzer eines Terminals muß SYSIO bitten, eine Task als neuen Besitzer anzukoppeln. Eine Task, die ein Terminal benötigt, fragt bei SYSIO mit diesem Wunsch an.

Diese Organisation wird in der kontrollierenden Task SYSIO durch einen Terminalstack für jeden Terminalmanager erreicht. Die Task TERMINAL ist fest als unterste Task in diesen Stack eingetragen. Sie gibt das Terminal an eine andere Task `T` weiter, indem sie einen **link request** an SYSIO leitet. Dieser **link request** wird von SYSIO an die gewünschte Task `T` geleitet. Falls diese ihr Einverständnis durch **link exec** gibt, wird `T` an die Stackspitze gestellt und die Verbindung zwischen `T` und den Treibern hergestellt. Damit ist der Besitzerwechsel vollzogen.

ACHTUNG: Ein Besitzer, der das Terminal weitergeben will, muß den **link request** wiederholen, bis die Weitergabe möglich ist, da nur so sichergestellt werden kann, daß er noch Besitzer ist.

Eine Task, die ohne unbedingte Terminalausgaben (»im Hintergrund«) aktiv ist, kann nur dann jemals wieder an ein Terminal gekoppelt werden, wenn sie irgendwann wieder bereit ist, eine Botschaft zu empfangen. Diese Botschaft, die gerade der **link request** von SYSIO ist, muß von der Task zustimmend beantwortet werden (**link exec**). Implizit wird diese Forderung dadurch erfüllt, daß die Task nach Abschluß der Aktionen, die ohne Koppelung an ein Terminal durchgeführt wurden, auf eine Eingabe wartet.

13.8 Die Botschaften des Link Protokolls

Botschaft	Code
link ack	0x03 00 00 00
link not ack	0x03 00 00 01
error not ack	0x03 00 00 03
invalid partner	0x03 00 00 05
link order	0x03 00 00 10
link request	0x03 00 00 11
link exec	0x03 00 00 12
terminal request	0x03 00 00 13
terminal inquiry	0x03 00 00 14
unlink	0x03 00 00 15
forced link	0x03 00 00 16
terminal	0x03 00 00 22
on line	0x03 00 00 23
task stack	0x03 00 00 24

13.8.1 Botschaften an SYSIO

Die folgenden Meldungen sind diejenigen, die eine Task an SYSIO schicken kann, um das Terminal weiter zu geben oder Informationen zu erhalten.

link request

Code: `0x03 00 00 11`

Parameter: `TASK next`

Wirkung: Der Besitzer des Terminals will das Terminal an 'next' weitergeben. Diese Botschaft an SYSIO muß wiederholt werden, solange die sendende Task Besitzer des Terminals ist. Falls 'next' die von SYSIO weitergeleitete Anfrage akzeptiert hat, wird vor Ausführen der Koppelung zwischen Terminal und 'next' die Quittung **link ack** von SYSIO gegeben. In jedem anderen Fall ist die Quittung **link nack**.

link exec

Code: `0x03 00 00 12`

Parameter: `TASK terminal TASK nexttask`

Wirkung: Mit dieser Meldung stimmt die Task 'nexttask' einer von SYSIO zugestellten **link request** Anfrage zu. Falls die Task, die die Weitergabe angestoßen hat, noch Eigentümer des Terminals ist, wird 'nexttask' auf die Spitze des Terminalstacks gesetzt und sowohl an die entsprechenden Terminaltreiber, als auch an 'nexttask' eine **link order** von SYSIO geschickt.

terminal request

Code: `0x03 00 00 13`

Parameter: -

Wirkung: Terminalanfrage an SYSIO. Wenn eine Task ein Terminal benötigt, sendet sie diese Botschaft. Falls die Task bereits an ein Terminal gekoppelt ist und an der Spitze des entsprechenden Taskstacks steht, schickt SYSIO **link order** zurück. Falls ein **link request** für die Task gestellt wurde, schickt SYSIO diesen weiter. In jedem anderen Fall wird nicht geantwortet, die Task bleibt also im geschlossenen Warten auf SYSIO.

terminal inquiry

Code: `0x03 00 00 14`

Parameter: -

Wirkung: Diese Botschaft kann eine Task an SYSIO senden, um zu erfragen, ob sie an ein Terminal gekoppelt ist (`link order` ist bereits ausgeführt worden) oder die Zuweisung eines Terminals bevorsteht (es gibt ein `link request` für die fragende Task). Falls die Task am Terminal hängt, schickt SYSIO eine erneute **link order** an die Task, falls ein **link request** für die fragende Task gestellt wurde, antwortet SYSIO mit `link request`. In jedem anderen Fall antwortet SYSIO mit **link nack**.

unlink

Code: `0x03 00 00 15`

Parameter: `TASK linked task`

Wirkung: Eine Task 'task an terminal' kann sich vom Terminal abkoppeln und der Terminalprozeß kann eine Task, die in seinem Taskstack bei SYSIO verzeichnet ist, durch diesen Auftrag vom Terminal abkoppeln.

terminal

Code: `0x03 00 00 22`

Parameter: `TASK usertask`

liefert: `TASK terminal`

Wirkung: Diese Anfrage an die Task SYSIO liefert den Terminalmanager, in dessen Stack die Task 'usertask' verzeichnet ist. 'usertask' muß nicht **online** sein, d.h. 'usertask' muß nicht auf der Spitze des Stacks für das betreffende Terminal stehen.

on line

Code: `0x03 00 00 23`

Parameter: `TASK usertask`

liefert: `BOOL flag`

Wirkung: Diese Anfrage an die Task SYSIO liefert TRUE, falls 'usertask' an der Spitze eines Terminalstacks steht. Falls die Task nicht an ein Terminal gekoppelt ist, wird FALSE geliefert. Dabei ist es unerheblich, ob die Task in irgendeinem Terminalstack, aber dort nicht an der Spitze, oder überhaupt nicht in einem solchen Stack ist.

13.8.2 Botschaften von SYSIO

link ack

Code: `0x03 00 00 00`

Parameter: Weitere gelieferte Botschaftsabschnitte sind situationsabhängig.

Wirkung: **link ack**nowledge ist die Standardantwort auf irgendwelche Aufträge von Tasks, wenn diese Aufträge von SYSIO korrekt abgewickelt wurden.

link nack

Code: `0x03 00 00 01`

Parameter: -

Wirkung: **link n**ot**ack**nowledge ist die Standardantwort auf irgendwelche Aufträge von Tasks, wenn diese Aufträge von SYSIO nicht ausgeführt werden konnten. Der Auftrag sollte dann wiederholt werden.

link order

Code: `0x03 00 00 10`

Parameter:

liefert: `TASK terminal`

Wirkung: Diese Antwort erfolgt auf einen **link exec** Auftrag. Zunächst erhalten Tastatur- und Bildschirmtreiber eine **link order** für die Task. Anschließend erfolgt die **link exec** Botschaft an die Task, die nun mit dem Treibern in Verbindung steht.

13.9 Das Dateimanager Protokoll

Dieses Protokoll beschreibt Botschaften von und an Tasks, die Dateimanager sind. Eine Beschreibung der verschiedenen Typen von Managerprozeduren finden Sie in Kapitel 8.

13.9.1 Die Botschaften des Dateimanager Protokolls

Botschaft	Code
Antworten eines Manager	
manager ack	0x04 00 00 00
manager not ack	0x04 00 00 01
manager error not ack	0x04 00 00 03
manager note ack	0x04 00 00 04
free manager	
begin	0x04 00 00 10

Datei Manager

remove	0x04 00 00 20
create	0x04 00 00 21
(old copy to)	0x04 00 00 22)
copy from	0x04 00 00 23
(old move to)	0x04 00 00 24)
move from	0x04 00 00 25
shape	0x04 00 00 26
attributes	0x04 00 00 27
is empty	0x04 00 00 28
all	0x04 00 00 29
list	0x04 00 00 2a
xlist	0x04 00 00 2b
help	0x04 00 00 2c
ds params	0x04 00 00 2d
overwrite	0x04 00 00 2e
copy to	0x04 00 00 2f

Archivemanager

reserve	0x04 00 00 30
release	0x04 00 00 31
volume name	0x04 00 00 32
format	0x04 00 00 33
clear	0x04 00 00 34
check	0x04 00 00 35
volume storage	0x04 00 00 36
init medium	0x04 00 00 38
read	0x04 00 00 39
write	0x04 00 00 3a

Spoolmanager

start	0x04 00 00 40
stop	0x04 00 00 41
halt	0x04 00 00 42
wait for halt	0x04 00 00 43
entry line	0x04 00 00 44
kill	0x04 00 00 45
first	0x04 00 00 46
log	0x04 00 00 50

13.9.2 Antworten eines Managers

manager ack

Code: `0x04 00 00 00`

Parameter: -

Wirkung: Quittung einer Managertask, daß der Auftrag wunschgemäß und ohne Fehler ausgeführt wurde. Dahinter folgen Rückgabe- Parameter.

manager not ack

Code: `0x04 00 00 01`

Parameter: -

Wirkung: Falls eine Managertask einen Auftrag im Moment nicht ausführen kann wird diese Antwort geschickt. Der Auftraggeber sollte den Auftrag wiederholen.

manager error not ack

Code: `0x04 00 00 03`

Parameter: `TEXT errormsg, INT errorcode, errorline`

Wirkung: Bei der Bearbeitung des Auftrags ist ein Fehler passiert.

manager note ack

Code: `0x04 00 00 04`

Parameter: `TEXT message`

Wirkung: Antwort einer Managertask, die in Verbindung mit einer kurzen textlichen Mitteilung an den Auftraggeber zurückgeschickt wird. Beispielsweise wird die Mitteilung des Archivmanagers: `"Archiv nicht angemeldet"` mit diesem Antwortcode geschickt.

13.9.3 Aufträge an freie Manager

begin

Code: `0x04 00 00 10`

Parameter: `TEXT begin password`
`TEXT task name`
`INT begin privilege`
`INT begin start address`
`DATASPACE std ds`

liefert: `TASK new task`

Wirkung: Falls das 'begin password' korrekt ist, erzeugt der Manager eine neue Task nach folgendem Algorithmus:

```
IF begin start address >= 0
   THEN begin (taskname,
               (min (begin privilege, priv (myself)) ,
               begin start address,
               std ds,
               new task)
   ELSE begin (taskname,
               (min (begiv priv, priv (myself),
               address (PROC monitor),
               next (nilspace),
               new task)
FI
```

13.9.4 Aufträge an Dateimanager

remove

Code: `0x04 00 00 20`

Parameter: `TEXT object name`

Wirkung: Das durch 'object name' bezeichnete Objekt wird gelöscht. Falls das Objekt ein Verzeichnis ist, wird es nur gelöscht, wenn es leer ist.

create

Code: `0x04 00 00 21`

Parameter: `TEXT object name`
`SHAPE object shape`

Wirkung: Ein Objekt namens 'object name' der Form 'object shape' wird angelegt. Falls es bereits ein Objekt dieses Namens gibt, wird eine Meldung ausgegeben: '"object name" existiert bereits'.

copy to

Code: `0x04 00 00 22`

Parameter: `TEXT dest name`
`DATASPACE source`
`ATTRIBUTES source attr`

Wirkung: Der Datenraum 'source' wird von der sendenden Task unter dem Namen 'dest name in den Dateimanager kopiert. Unterverzeichnisse werden ggf. erzeugt.

copy from

Code: `0x04 00 00 23`

Parameter: `TEXT source name`

liefert: `DATASPACE dest`
`ATTRIBUTES dest attr`

Wirkung: Der Datenraum 'source name' wird von der Managertask zum Auftraggeber kopiert.

move from

Code: `0x04 00 00 25`

Parameter: `TEXT source name`

liefert: `DATASPACE dest name`
`ATTRIBUTES dest attr`

Wirkung: Der Datenraum 'source name' wird von der Managertask zum Auftraggeber geholt und im Manager gelöscht.

shape

Code: `0x04 00 00 26`

Parameter: `TEXT object name`

liefert: `SHAPE object shape`

Wirkung: Liefert die Gestalt des Objekts 'object name'. Falls es kein Objekt dieses Namens im Manager gibt wird **nilshape** geliefert.

attributes

Code: `0x04 00 00 27`

Parameter: `TEXT objekt name`

liefert: `ATTRIBUTES object shape`

is empty

Code: `0x04 00 00 28`

Parameter: `TEXT directory name`

liefert: `BOOL dir is empty`

Wirkung: Liefert TRUE, falls das Verzeichnis existiert und keine Einträge enthält. Falls das Verzeichnis existiert, aber nicht leer ist, wird FALSE geliefert.

all

Code: `0x04 00 00 29`

Parameter: `TEXT directory name`
`SHAPE certain shape`

liefert: `THESAURUS names`

Wirkung: Aus dem angegebenen Verzeichnis werden alle Objekte der gewünschten Gestalt in der Namensliste 'names' geliefert.

list

Code: `0x04 00 00 2A`

Parameter: `TEXT directory name`
`SHAPE certain shape`

liefert: `DATASPACE list space`

Wirkung: Aus dem angegebenen Verzeichnis werden alle Objekte der gewünschten Gestalt in 'list space' gelistet. Der 'list space' ist ein unbenannter Datenraum der inneren Struktur **FILE**, der zeilenweise Informationen über die gelisteten Objekte enthält (Datum und Uhrzeit der letzten Änderung, Name).

xlist

Code: `0x04 00 00 2B`

Parameter: `TEXT directory name`
`SHAPE certain shape`

liefert: `DATASPACE xlist space`

Wirkung: Aus dem angegebenen Verzeichnis werden alle Objekte der gewünschten Gestalt in 'xlist space' gelistet. Der 'xlist space' ist ein unbenannter Datenraum der inneren Struktur **FILE**. Es werden auch alle Unterverzeichnisse des angegebenen Verzeichnisses mit aufgelistet.

help

Code: `0x04 00 00 2C`

Parameter:

liefert: `DATASPACE help space`

Wirkung: Der Manager liefert eine unbenannten Datenraum der inneren Struktur **FILE**. Er enthält einen Hilfetext mit der Beschreibung der Funktionen des Managers.

ds params

Code: `0x04 00 00 2D`

Parameter: `TEXT dataspace name INT mode`

liefert: `INT dataspace params`

Wirkung: 'mode'= 0 liefert die Größe des angegebenen Datenraums in Byte, 'mode'= 2 liefert den Typ des Datenraums.

overwrite

Code: `0x04 00 00 2e`

Parameter: `TEXT source name`
`DATASPACE source`
`ATTRIBUTES source attributes`
`TEXT dest name`

Wirkung: Der Datenraum 'source' wird in den Manager kopiert. Falls dort ein Datenraum dieses Namens existiert, wird der überschrieben.

copy to

Code: `0x04 00 00 2f`

Parameter: `TEXT source name`
`DATASPACE source`
`ATTRIBUTES source attributes`
`TEXT dest name`

Wirkung: Der Datenraum 'source' wird in den Manager kopiert. Falls dort ein Datenraum dieses Namens existiert, liefert der Manager ein **manager not ack** mit folgenden Parametern:`DATASPACE source, TEXT extended dest name, SHAPE dest shape`

13.9.5 Aufträge an Archivmanager

reserve

Code: `0x04 00 00 30`

Parameter: `TEXT reserve param`

Wirkung: Der Auftraggeber reserviert den Archivmanager. Alle Aktionen des Archivmanagers werden ab Zeitpunkt der Reservierung exklusiv vom Auftraggeber angestoßen.

release

Code: `0x04 00 00 31`

Parameter: -

Wirkung: Die Reservierung wird beendet.

volume name

Code: `0x04 00 00 32`

Parameter:

liefert: `TEXT volume name`

Wirkung: Liefert den »volume name« des Archivmediums.

format

Code: `0x04 00 00 33`

Parameter: `INT format code`

Wirkung: Das Archivmedium wird für den erstmaligen Gebrauch vorbereitet, also formatiert und initialisiert. Es wird physisch gelöscht. Gültige Angaben für 'format code' bei Disketten sind:

```
   0      High Density
 360      DD 1S     360K
 720      DD 2S     720K
1200      HD 5.25"  1200K
1440      HD 3.5"   1440K
```

clear

Code: `0x04 00 00 34`

Parameter: -

Wirkung: Das Archivmedium wird reinitialisiert, also logisch (aber nicht physisch) gelöscht.

check

Code: `0x04 00 00 35`

Parameter: `TEXT object name`

Wirkung: Prüflesen.

volume storage

Code: `0x04 00 00 36`

Parameter:

liefert: `INT size INT used`

Wirkung: Liefert Größe und belegten Speicherplatz des Archivmediums in Byte.

init medium

Code: `0x04 00 00 38`

Parameter:

liefert: `INT medium type`

Wirkung: Initialisiert den Treiber des Archivmediums und liefert den Typ des vorhandenen Archivmediums.

read

Code: `0x04 00 00 39`

Parameter: `BOOL cylinder orientated`
`INT sector no`
`INT stream length`

liefert: `BYTESTRING stream`

Wirkung: Liest 'stream length' Bytes beginnend bei Sektornummer 'sector no' von Archivmedium. 'stream length' muß ein Vielfaches der Sektorgröße des Archivmediums sein.

write

Code: `0x04 00 00 3A`

Parameter: `BOOL cylinder orientated`
`INT sector no`
`BYTESTRING stream`

Wirkung: Schreibt 'stream length' Bytes beginnend bei Sektornummer 'sector no' auf das Archivmedium. 'stream length' muß ein Vielfaches der Sektorgröße des Archivmediums sein.

13.9.6 Aufträge an Spoolmanager

start

Code: `0x04 00 00 40`

Parameter: `TEXT password`

Wirkung: Der Spool wird aktiviert, indem eine neue Servertask (Taskname '-') gestartet wird. Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden. Falls der Spool bereits aktiviert ist, erfolgt eine Fehlermeldung.

Fehler: "Spool muß zuerst gestoppt werden"

stop

Code: `0x04 00 00 41`

Parameter: `TEXT password`
`TEXT deactivate message`

Wirkung: Der Spool wird deaktiviert, indem die Servertask gelöscht wird. Falls eine Auftrag in Bearbeitung ist, wird die Bearbeitung abgebrochen. Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden. Falls eine Meldung 'deactivate message' mitgegeben wird, wird diese Meldung bei **list** auf den Spoolmanager ausgegeben.

halt

Code: `0x04 00 00 42`

Parameter: `TEXT password`
`TEXT deactivate message`

Wirkung: Der Spool wird deaktiviert, indem die Servertask nach Beendigung des aktuellen Druckauftrags gelöscht wird. Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden. Falls eine Meldung 'deactivate message' mitgegeben wird, wird diese Meldung bei **list** auf den Spoolmanager ausgegeben.

wait for halt

Code: `0x04 00 00 43`

Parameter:

Wirkung: Der Spool wird deaktiviert, indem die Servertask nach Beendigung des aktuellen Druckauftrags gelöscht wird. Die Antwort wird verzögert, bis der Spool deaktiviert ist. Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden. Falls eine Meldung 'deactivate message' mitgegeben wird, wird diese Meldung bei **list** auf den Spoolmanager ausgegeben.

kill

Code: `0x04 00 00 45`

Parameter: `TEXT password`
`INT index`

Wirkung: Der indexte Eintrag in der Warteschlange des Spoolers wird gelöscht. Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden.

first

Code: `0x04 00 00 46`

Parameter: `TEXT password`
`INT index`

Wirkung: Der indexte Eintrag in der Warteschlange des Spoolers wird an die erste Stelle gesetzt.Falls ein **spool control password** eingestellt ist, muß dieses beigegeben werden.

log

Code: `0x04 00 00 50`

Parameter: `TEXT log message`

Wirkung: Der Text 'log message' wird in das »Logbuch« der Log Manager Task eingetragen.

14 Die Systemaufrufe

14.1 Einführung

Die Systemaufrufe stellen die unterste Ebene dar, auf der Leistungen von L3 in Anspruch genommen werden können. Dabei wird dem L3 Systemkern durch Interrupt 7 mitgeteilt, daß ein Systemaufruf ansteht. Die Identifikation des Systemaufrufs erfolgt stets durch Auswertung des Registers `AL`, in dem der Code des Aufrufs steht. Ein- und Ausgabeparameter werden in den beim jeweiligen Systemaufruf beschriebenen Registern übergeben.

Es handelt sich bei allen Operationen um Basisleistungen des L3 Systemkerns. Die involvierten Datentypen sind also Basistypen, die bereits auf Ebene des Systemkerns bekannt sind: Prozesse, Datenräume, Botschaften und die Inhalte des Prozeßkontrollblocks (PCB).

Objekte des Typs Prozeß werden durch den Datentyp *Prozeßbezeichner* identifiziert. Ein Prozeßbezeichner, in der Beschreibung der Systemaufrufe als `PRCID` bezeichnet, bestehend aus `(INT CONST process number, version number)`. Diese vom Systemkern für jeden Prozeß vergebene Kombination von Prozeß- und Versionsnummer stellt sicher, daß Prozesse innerhalb des L3 Systems auch über die Zeit eindeutig identifizierbar bleiben. Die Prozeßnummer beschreibt die Stellung des Prozesses im Gesamtsystem. Die Nummern 1 bis 42h bezeichnen Kern- und Treiberprozesse, die Nummern ab 43h bezeichnen Tasks. Die Versionsnummer garantiert die Eindeutigkeit des Prozeßbezeichners über die Zeit. Wenn ein Prozeß gelöscht wird und »an gleicher Stelle« ein Prozeß erzeugt wird, so wird durchaus die gleiche Prozeßnummer wieder zugeteilt. Die Versionsnummer wird jedoch hochgezählt, so daß nicht die Situation entstehen kann, daß ein Prozeß Botschaften erhält, die für seinen (zeitlichen) Vorgänger bestimmt waren.

Für die Parameterübergabe eines Prozeßbezeichners vom oder an den Systemkern sind zwei Doppelworte, mithin zwei 80386 Register erforderlich.

Objekte des Typs Datenraum werden durch Datenraumbezeichner (`DSID`) identifiziert. Da Datenräume nur innerhalb einer Task eindeutig bezeichnet sein müssen, reicht eine einfache Numerierung. Es gibt schließlich keine Möglichkeit, an die Datenräume einer anderen Task »heranzukommen« (Autonomieprinzip). Datenraumbezeichner sind also ganze Zahlen, Doppelworte. Die Parameterübergabe erfolgt somit über ein 80386 Register.

Um Zugriff auf den Inhalt eines Datenraums zu erhalten, ist es nötig, daß der Datenraum in den linearen Adreßraum des Prozesses abgebildet (»*gemappt*«) wird.

Die Objekte der Intertask Kommunikation, sind die Botschaftsvektoren, strukturierte Datenobjekte folgender Gestalt: ein Botschaftsvektor ist ein String von 32 Doppelworten. In diesen String werden in beliebiger Mischung und Anordnung Einträge für Botschaftsobjekte der Typen INT, Bytesequenz (z.B. TEXT) oder DATASPACE gemacht. Ein Eintrag für INT und DATASPACE ist ein Doppelwort, ein Bytesequenz-Eintrag zwei Doppelworte lang. Eine Botschaft kann einen bis sechzehn Integer, einen bis sechzehn Datenräume, einen bis sieben Texte oder beliebige Mischungen daraus bis zur maximalen Eintragslänge von sechzehn Doppelworten aufnehmen.

14.2 Der Aufbau des Botschaftsvektors

Der Botschaftsvektor ist ein 128 Byte langer Abschnitt im Prozeßkontrollblock einer Task.

```
MDV  RDV  B1  B2 .... B15  B16  E1  E2 .... E13  E14
```

Die Doppelworte 'MDV' und 'RDV' beschreiben den Datentyp der Objekte in den folgenden Botschaftsabschnitten. B1 bis B16 sind Botschaftsabschnitte, E1 bis E14 Eigenadressen.

MDV: Message Dope Vector Der Message Dope Vektor beschreibt die Typen der zu sendenden Datenobjekte in den Botschaftsabschnitten B1 bis B16. Für jeden Abschnitt B1 bis B16 muß in zwei Bit der Datentyp festgehalten werden, als den der Empfänger den Inhalt des Botschaftsabschnitts zu interpretieren hat. Verschlüsselung der Datentypen von Botschaftsabschnitten:

`int msg`: 0

`string msg`: 1

`ds msg`: 2

`nil msg`: 3

RDV: Receive Dope Vector Der Receive Dope Vektor beschreibt die Typen der zu empfangenden Datenobjekte in den Botschaftsabschnitten B1 bis B16. Für jeden Abschnitt B1 bis B16 wird in zwei Bit der Typ festgehalten (s.o.). Nach Empfang einer Botschaft kann durch Abgleich des bei der Botschaftsübermittlung mitübertragenen MDV und des selbst erzeugten RDV geprüft werden, ob die Sendung den eigenen Erwartungen entspricht und verarbeitet werden kann.

B1 bis B16: Botschaftsabschnitte Die Botschaftsabschnitte enthalten die eigentlichen Botschaften in Form von Integer Einträgen. Die Einträge unterliegen folgender Systematik:

Sender vor der Sendung:

`int msg`: Der Abschnitt Bx enthält die Zahl.

`ds msg`: Der Abschnitt Bx enthält den Datenraumbezeichner.

`string msg`: Der Abschnitt Bx+1 enthält die Startadresse des Bytestring in der Task. Der Abschnitt Bx enthält die Stringlänge.

Nach erfolgreicher Sendung ist ein gesendeter Datenraum gelöscht. INTs oder Strings werden als Kopien versendet.

Empfänger nach der Sendung:

`int msg`: Der Abschnitt Bx enthält die Zahl.

`ds msg`: Der Abschnitt Bx enthält den Datenraumbezeichner des empfangenen Datenraums.

`string msg`: Der Abschnitt Bx enthält die Stringlänge des Bytestrings. Der Abschnitt Ex+1 enthält die Adresse.

Da der Empfänger im Falle der *string msg* eine Fremdadresse erhält, muß noch die Angabe eines geeigneten Platzes im eigenen Adreßraum folgen. Nur zu diesem Zweck werden die Eigenadressen E1 bis E14 benötigt.

E1 bis E14: Eigenadressen In diesem Abschnitt werden für *string msg* die Adresse und die akzeptierte Länge für den jeweiligen Bytestring im eigenen Adreßraum festgehalten.

Dieser Abschnitt des Botschaftsvektors wird also nur benutzt, wenn Bytestrings empfangen werden sollen.

14.2.1 Botschaftstransfer im V86 Modus

Falls aus dem virtuellen 8086 Modus heraus Botschaften gesendet oder empfangen werden sollen, muß der Botschaftsvektor selbst verwaltet werden. Dazu muß eine Datenstruktur der oben beschriebenen Gestalt in einem gesonderten Segment, auf das das Register `GS` zeigt, aufgebaut werden.

```
GS:0 ->        MDV  RDV  B1  B2 .... B15  B16  E1  E2 .... E13  E14
```

14.3 Thematische Gliederung der Systemaufrufe

Funktionscode (in AL)	Funktionsname
Datenräume	
0	copy dataspace
1	delete dataspace
2	copy pages to dataspace
3	dataspace exists
5	next dataspace page
6	copy pages
7	open dataspace
8	close dataspace
9	associated segment
10	associated dataspace
11	next dataspace
12	set access right
13	dataspace alias
14	get dataspace params
15	set dataspace params

Prozeßkommunikation

16	myself id
17	sys process id
18	exists process
19	my task id
28	call process
29	send process
30	receive process
31	wait process

Prozeßkontrolle

32	get state
34	dataspaces
36	block process
37	unblock process
38	set prio

Verschiedenes

50	switch process
51	set alarm
52	special operation
53	get real time
54	get date time
61	control out

14.4 Systemaufrufe

copy dataspace

Code: `0`

Wirkung: Von einem existierenden Datenraum wird eine Kopie angelegt, deren Datenraumbezeichner in Register `EAX` geliefert wird. Der zu kopierende Datenraum braucht nicht »geöffnet« (gemappt) zu sein, er darf aber geöffnet sein. Insbesondere kann auch der leere Datenraum **nilspace** als Quelle herangezogen werden, um einen neuen Datenraum vor Benutzung zu »initialisieren«.

Register vor INT 7

```
 AL : 0
EBX : DSID source ds
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : DSID dest ds
EBX :
ECX :
EDX :
```

Parameter: 'source ds' ist der Datenraumbezeichner des Datenraums, der kopiert werden soll. 'dest ds' ist der Datenraumbezeichner der Kopie.

delete dataspace

Code: 1

Wirkung: Löscht einen Datenraum. Der durch seinen Datenraumbezeichner spezifizierte Datenraum ist danach für den aufrufenden Prozeß nicht mehr zugreifbar (adressierbar). Der zu löschende Datenraum muß nicht geöffnet (gemappt) sein, er darf aber geöffnet sein. Als Resultat wird ein Datenraumbezeichner geliefert, der in jedem Prozeß garantiert ungültig ist. Dieser Bezeichner liefert für jede Operation außer **delete dataspace** einen Fehler.

Register vor INT 7

```
 AL : 1
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : DSID no dataspace
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, der gelöscht werden soll.

'no dataspace' ist der immer ungültige Datenraumbezeichner.

copy pages to dataspace

Code: 2

Wirkung: Faules Kopieren von Datenraumseiten (Pages) aus dem Quelldatenraum in den Zieldatenraum. Beide Datenräume müssen gemappt sein. Sowohl für Quelldatenraum, als auch für Zieldatenraum wird eine Startadresse angegeben, ab der kopiert werden soll. Diese Adresse muß in beiden Fälle »page aligned«, d.h. ohne Rest durch 4096 teilbar sein. Im Zieldatenraum wirkt die Kopie überschreibend.

Register vor INT 7

```
 AL : 2
EBX : DSID source ds
ECX : DSID dest ds
EDX : ADDR from page
ESI : ADDR to page
EDI : INT pages
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'source ds' ist der Datenraumbezeichner des Datenraums, aus dem Seiten kopiert werden. 'dest ds' ist der Datenraumbezeichner des Datenraums in den Seiten kopiert werden. 'from page' ist die Startadresse, ab der 'pages' Seiten kopiert werden. Die Adresse muß »page aligned« sein, d.h. ohne Rest durch 4096 teilbar. 'to page' ist die Zieladresse, an die 'pages' Seiten kopiert werden. Die Adresse muß »page aligned« sein, d.h. ohne Rest durch 4096 teilbar. 'pages' ist die Anzahl der Seiten, die kopiert werden sollen.

dataspace exists

Code: `3`

Wirkung: Prüft, ob der gegebenene Datenraumbezeichner einen existierenden Datenraum bezeichnet. Der fragliche Datenraumbezeichner wird in Register `EBX` übergeben. Falls der Datenraum existiert wird in `EAX` eine `1` geliefert, sonst `0`.

Register vor INT 7

```
 AL : 3
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : BOOL exists
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner für den überprüft wird, ob er einen gültigen Datenraum bezeichnet. 'exists' liefert den Wert 1, falls der Bezeichner einen gültigen Datenraum bezeichnet, sonst 0.

next dataspace page

Code: 5

Wirkung: Liefert die Seitennummer der nächsten benutzten Seite (Page) des bezeichneten Datenraums. Die Suche startet bei der spezifizierten Seite 'from page'. Falls dafür der Wert -1 angegeben wird, so wird die erste benutzte Seite des Datenraums geliefert. Die Suche nach der nächsten benutzten Seite, die an der letzten benutzten Seite startet, liefert wiederum den Wert -1.

Register vor INT 7

```
 AL : 5
EBX : DSID dataspace
ECX : ADDR from page
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : INT next used page
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums. 'from page' ist die Seitennummer der Seite, ab der gesucht werden soll. 'next used page' ist die Seitennummer der nächsten belegten Seite.

copy pages

Code: 6

Wirkung: Faules Kopieren von Datenraumseiten (Pages) innerhalb eines Datenraumes. Der Datenraum muß gemappt sein. Sowohl die Startadresse, ab der kopiert werden soll, als auch die Zieladresse müsse »page aligned«, d.h. ohne Rest durch 4096 teilbar sein. Die Kopie wirkt überschreibend.

Register vor INT 7

```
 AL : 6
EBX : DSID dataspace
ECX : ADDR from page
EDX : ADDR to page
ESI : INT pages
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'source ds' ist der Datenraumbezeichner des Datenraums, in dem Seiten kopiert werden. 'from page' ist die Startadresse, ab der 'pages' Seiten kopiert werden. Die Adresse muß »page aligned« sein, d.h. ohne Rest durch 4096 teilbar. 'to page' ist die Zieladresse, an die 'pages' Seiten kopiert werden. Die Adresse muß »page aligned« sein, d.h. ohne Rest durch 4096 teilbar. 'pages' ist die Anzahl der Seiten, die kopiert werden sollen.

open dataspace

Code: 7

Wirkung: Der angeforderte Datenraum wird in den linearen Adreßraum des Prozesses »gemappt«. Für den Eintrag in die Mapping Liste des Prozeßkontrollblocks wird ein **bound** Eintrag, der praktisch eine Referenz auf den neu eröffneten Datenraum darstellt angelegt (s.u.). Falls kein Mapping möglich ist, wird der aufrufende Prozeß durch Systemaufruf 48 (set error) benachrichtigt.

Register vor INT 7

```
 AL : 7
EBX : DSID dataspace
ECX : INT size
DS:EDX : bound
ESI :
EDI :
```

Register nach INT 7

```
EAX : new bound (low)
EBX : new bound (high)
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, der gemappt werden soll. 'size' ist der Platzbedarf, der für diesen Datenraum eingefordert wird. Der gegebene Wert wird automatisch auf das nächst höhere Vielfache von 16 MB gerundet. Der Maximalwert beträgt ein Gigabyte.

Bemerkung: Ein Prozeß kann prinzipiell 16383 Datenräume besitzen. Der Prozeßkontrollblock erlaubt jedoch nur gleichzeitiges Mapping von 234 Datenräumen. Bei Abbildung eines Datenraums in den linearen Adreßraum des Prozesses wird daher ein Eintrag in die Mapping Liste des PCB gemacht. Dieser Eintrag ist als Zeiger auf den gemappten Datenraum zu interpretieren.

close dataspace

Code: `8`

Wirkung: Der bezeichnete Datenraum wird aus dem linearen Adreßraum des Prozesses entfernt.

Register vor INT 7

```
 AL : 8
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, der »unmappt« werden soll.

associated segment

Code: `9`

Wirkung: Liefert die lineare Adresse der ersten benutzten Seite des bezeichneten Datenraum, falls dieser Datenraum gemappt ist, sonst `0xffffffff`.

Register vor INT 7

```
 AL : 9
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : ADDR address
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, dessen erste benutzte Adresse im linearen Adreßraum des erfragt werden soll. 'address' enthält die Adresse, falls der Datenraum gemappt ist, sonst `0xffffffff`.

associated dataspace

Code: `10`

Wirkung: Liefert den Datenraumbezeichner des Datenraums, der an die angegebene lineare Adresse gemappt ist. Falls der Adresse kein Datenraum geordnet werden kann, wird '0', der Bezeichner von **nilspace** geliefert.

Register vor INT 7

```
 AL : 10
EBX : ADDR address
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : DSID dataspace
EBX :
ECX :
EDX :
```

Parameter: 'address' enthält die lineare Adresse, zu der der assoziierte Datenraum gesucht wird. 'dataspace' ist der Datenraumbezeichner des Datenraums, dessen Mapping die fragliche Adresse enthält. Falls es keinen solchen Datenraum gibt, wird DSID nilspace ('0') geliefert.

next dataspace

Code: `11`

Wirkung: Liefert den Datenraumbezeichner des nächsten existierenden Datenraums.

Die Datenräume sind zyklisch geordnet:

```
next dataspace (nilspace) = std dataspace
next dataspace (last dataspace) = nilspace
```

Auf diese Weise läßt sich einfach der Standarddatenraum identifizieren. Desweiteren können auch alle Datenräume der Task leicht durchmustert werden.

Register vor INT 7

```
 AL : 11
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : DSID next dataspace
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, zu dem der nächste Datenraum gefunden werden soll. 'next dataspace' ist der Datenraumbezeichner des gesuchten Datenraums. Wie oben beschreiben sind die Datenräume zyklisch geordnet.

set access right

Code: `12`

Wirkung: Verbietet oder erlaubt schreibenden Zugriff auf eine bestimmte Reihe von Seiten eines Datenraums. Der Datenraum muß im linearen Adreßraum der Task liegen (»gemappt sein«). Die Adressangabe für die erste zu Seite, die geschützt oder freigegeben werden soll, muß »page aligned«, d.h. ohne Rest durch 4096 teilbar sein.

Register vor INT 7

```
 AL : 12
EBX : DSID dataspace
ECX : ADDR from page
EDX : INT pages
ESI : INT access right
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, in dem ein Schreibschutz gesetzt oder gelöscht werden soll. 'from page' ist die Startadresse, ab der 'pages' Seiten geschützt werden. Die Adresse muß »page aligned« sein, d.h. ohne Rest durch 4096 teilbar. 'pages' ist die Anzahl der Seiten, die geschützt werden sollen. 'access right': `0` nur lesen der Seiten ist erlaubt. `1` lesen und schreiben ist erlaubt.

dataspace alias

Code: `13`

Wirkung: Von einem existierenden Datenraum wird ein »Alias« angelegt. Ein Alias ist im Gegensatz zu einer Kopie logisch mit seinem Original gekoppelt. Somit kann beispielsweise die Kommunikation von »shared memory« durch Aliases realisiert werden. Ein

Alias ist logisch einem Datenraum gleichwertig, erlaubt also alle üblichen Operationen. Die Löschung eines Alias hat jedoch keine Wirkung auf andere Aliases oder das Original. Die Löschung des Originals macht jedoch alle Aliases ungültig, der Eigentümer des Originals bleibt also entscheidender Besitzer. Falls irgendein Prozeß einen Schreibschutz auf ein Alias setzt oder löscht, so wirkt das auch auf andere Aliases und Original.

Falls irgend möglich, sollten die üblichen Methoden der Intertask Kommunikation benutzt werden!

Register vor INT 7

```
 AL : 13
EBX : DSID dataspace
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : DSID alias
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, von dem ein Alias angelegt werden soll. 'alias' ist der Datenraumbezeichner des Alias von 'dataspace'.

get dataspace params

Code: 14

Wirkung: Liefert Informationen über einen Datenraum. Diese globalen Informationen sind Typ des Datenraums, die Anzahl der belegten Seiten und die maximal zulässige Größe, die dieser Datenraum annehmen darf.

Register vor INT 7

```
 AL : 14
EBX : DSID dataspace
ECX : INT param
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : INT param value
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, über den Informationen gewüscht werden. 'param': 0 Belegte Seiten des Datenraums 1 Zulässige Maximalgröße 2 Typ des Datenraums

set dataspace params

Code: 15

Wirkung: Ermöglicht die Einstellung von Typ des Datenraums und der maximal zulässigen Größe, die dieser Datenraum annehmen darf.

Register vor INT 7

```
 AL : 15
EBX : DSID dataspace
ECX : INT param
EDX : INT param value
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'dataspace' ist der Datenraumbezeichner des Datenraums, über den Informationen gewüscht werden. 'param': 1 Zulässige Maximalgröße 2 Typ des Datenraums 'param value' ist der neu gesetzte Wert.

myself id

Code: 16

Wirkung: Liefert den Prozeßbezeichner des Prozesses.

Register vor INT 7

```
 AL : 16
EBX :
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : PRCID myself (low DWORD)
EBX : PRCID myself (high DWORD)
ECX :
EDX :
```

Parameter: 'myself' Prozeßbezeichner des aufrufenden Prozesses.

sys process id

Code: `17`

Wirkung: Liefert den Prozeßbezeichner des durch 'sys prc short name' bezeichneten Prozesses. Kurznamen, die bis zu vier Zeichen lang sind, gibt es für Prozesse, die dem Systemkern zugehören und den SUPERVISOR. Unter Benutzung dieser Kurznamen läßt sich der Prozeßbezeichner in Erfahrung bringen. Dieser wiederum wird dann benutzt um Botschaften an einen solchen Prozess zu senden.

Register vor INT 7

```
 AL : 17
EBX : sys prc short name
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : PRCID sys process id (low DWORD)
EBX : PRCID sys process id (high DWORD)
ECX :
EDX :
```

Parameter: 'sys prc short name' ist der Kurzname eines Systemprozesses. Kurznamen gibt es für SUPERVISOR und Treiberprozesse:

"SV" Supervisor "LPT1" Parallele Schnittstelle 1

'sys process id': Interner Prozeßbezeichner für den mit **sys prc short name** spezifizierten Prozeß. Falls dieser nicht gefunden wird, wird **nilprocess** (PRCID (0,0)) geliefert.

Bemerkung: Der Name 'short name' muß entsprechend den Konventionen *rückwärts* in das Register geschrieben werden. `mov ebx, '1TPL'` entspricht also LPT1 in Register `EBX`.

exists process

Code: `18`

Wirkung: Prüft, ob der angegebene Prozeß existiert.

Register vor INT 7

```
 AL : 18
EBX : PRCID process (low DWORD)
ECX : PRCID process (high DWORD)
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : exists
EBX :
ECX :
EDX :
```

Parameter: 'process': Prozeßbezeicher. 'exists' : EAX enthält den Wert 1, falls 'process' existiert, sonst 0.

my task id

Code: `19`

Wirkung: Liefert den Prozeßbezeichner der eigenen Task.

Register vor INT 7

```
 AL : 19
EBX :
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : PRCID my task id (low DWORD)
EBX : PRCID my task id (high DWORD)
ECX :
EDX :
```

Parameter: 'my task id': Interner Prozeßbezeichner für die Task, die diesen Systemaufruf abgesetzt hat.

call process

Code: `28`

Wirkung: Versucht einem Prozeß eine Botschaft zu senden und wartet nach erfolgter Sendung auf eine Antwort. `call process` ist eine *atomare Aktion*, d.h. es ist garantiert, daß keine Aktion nach Ende der Sendung und vor Beginn des Wartens auf die Antwort stattfindet. Falls der Empfänger nicht empfangsbereit ist, wird darauf gewartet, daß 'sendtimeout' abläuft, falls nach der Sendung keine Antwort kommt, wird bis zum Ablauf von 'receivetimeout' gewartet.

Register vor INT 7

```
 AL : 28
EBX : lower DWORD of PRCID partner
ECX : upper DWORD of PRCID partner
EDX : send timeout
ESI : receive timeout
EDI : 0  oder  MSGVEC in GS:0
```

Register nach INT 7

```
EAX : result
EBX :
ECX :
EDX :
```

Parameter: 'partner' ist der Prozeßbezeichner des Prozesses, dem die Botschaft gesendet werden soll. 'send timeout' ist die Zeitspanne, die maximal verstreichen darf, ohne daß der Versuch, die Sendung zuzustellen, wegen Erfolglosigkeit abgebrochen wird. 'receive timeout' ist die Zeitspanne, die maximal verstreichen darf, ohne daß der Versuch, die Antwort des Partners zu empfangen, wegen Erfolglosigkeit abgebrochen wird. 'result' liefert einen der folgenden Werte:

```
0 : communication ok
1 : task not existing
2 : send timeout
3 : receive timeout
5 : communication error
```

Bemerkung: Falls dieser Systemaufruf im virtuellen 8086 Modus benutzt wird, muß ein separates Segment auf das das Segmentregister `GS` zeigen muß, aufgebaut werden., Dieses Segment enthält dann einen Botschaftsvektor in der Struktur wie in 14.2 beschrieben.

send process

Code: 29

Wirkung: Versucht einem Prozeß eine Botschaft zu senden. Falls der Empfänger nicht empfangsbereit ist, wird bis zum Ablauf von 'sendtimeout' gewartet.

Register vor INT 7

```
 AL : 29
EBX : PRCID partner (low DWORD)
ECX : PRCID partner (high DWORD)
EDX : send timeout
ESI :
EDI : 0  oder MSGVEC in GS:0
```

Register nach INT 7

```
EAX : result
EBX :
ECX :
EDX :
```

Parameter: 'partner' ist der Prozeßbezeichner des Prozesses, dem die Botschaft gesendet werden soll. 'send timeout' ist die Zeitspanne, die maximal verstreichen darf, ohne daß der Versuch, die Sendung zuzustellen, wegen Erfolglosigkeit abgebrochen wird. 'result' liefert einen der folgenden Werte:

```
0 : communication ok
1 : task not existing
2 : send timeout
5 : communication error
```

Bemerkung: Falls dieser Systemaufruf im virtuellen 8086 Modus benutzt wird, muß ein separates Segment auf das das Segmentregister `GS` zeigen muß, aufgebaut werden., Dieses Segment enthält dann einen Botschaftsvektor in der Struktur wie in 14.2 beschrieben.

receive process

Code: 30

Wirkung: Versucht von einem bestimmten Sender eine Botschaft zu empfangen (Geschlossenes Warten). Es wird bis zum Ablauf von 'receive timeout' auf das Eintreffen einer Sendung gewartet.

Register vor INT 7

```
 AL : 30
EBX : PRCID partner (low DWORD)
ECX : PRCID partner (high DWORD)
EDX : receive timeout
ESI :
EDI : 0  oder MSGVEC in GS:0
```

Register nach INT 7

```
EAX : result
EBX :
ECX :
EDX :
```

Parameter: 'partner' ist der Prozeßbezeichner des Prozesses, von dem die Botschaft empfangen werden soll. 'receive timeout' ist die Zeitspanne, die maximal verstreichen darf, ohne daß der Versuch, die Sendung zu empfangen, wegen Erfolglosigkeit abgebrochen wird. 'result' liefert einen der folgenden Werte:

```
0 : communication ok
1 : task not existing
3 : receive timeout
5 : communication error
```

Bemerkung: Falls dieser Systemaufruf im virtuellen 8086 Modus benutzt wird, muß ein separates Segment auf das das Segmentregister `GS` zeigen muß, aufgebaut werden., Dieses Segment enthält dann einen Botschaftsvektor in der Struktur wie in 14.2 beschrieben.

wait process

Code: `31`

Wirkung: Versucht von einem beliebigen Sender eine Botschaft zu empfangen (Offenes Warten). Es wird bis zum Ablauf von 'receive timeout' auf das Eintreffen einer Sendung gewartet.

Register vor INT 7

```
 AL : 31
EBX :
ECX :
EDX : receive timeout
ESI :
EDI : 0  oder MSGVEC in GS:0
```

Register nach INT 7

```
EAX : result
EBX : PRCID partner (low DWORD)
ECX : PRCID partner (high DWORD)
EDX :
```

Parameter: 'partner' ist der Prozeßbezeichner des Prozesses, der die Botschaft gesendet hat. 'receive timeout' ist die Zeitspanne, die maximal verstreichen darf, ohne daß der Versuch, eine Sendung zu empfangen, wegen Erfolglosigkeit abgebrochen wird. 'result' liefert einen der folgenden Werte:

```
0 : communication ok
1 : task not existing
3 : receive timeout
5 : communication error
```

Bemerkung: Falls dieser Systemaufruf im virtuellen 8086 Modus benutzt wird, muß ein separates Segment auf das das Segmentregister `GS` zeigen muß, aufgebaut werden., Dieses Segment enthält dann einen Botschaftsvektor in der Struktur wie in 14.2 beschrieben.

get state

Code: 32

Wirkung: Liefert einen Auszug aus dem Prozeßkontrollblock eines Prozesses. Dieser sogenannte Statusvektor enthält Angaben über Rechenzeit, Speicherplatzbelegung, Kommunikationspartner etc.

Register vor INT 7

```
 AL : 32
EBX : PRCID process id (low DWORD)
ECX : PRCID process id (high DWORD)
 DS :  EDX :<STATUSVECTOR statevec>
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
 DS :  EDX :<STATUSVECTOR statevec>
```

Parameter: 'processid' Prozeßbezeichner 'statevec' Statusvektor:

Code: 34

Wirkung: Liefert die Anzahl der Datenräume einer Task.

Register vor INT 7

```
 AL : 34
EBX : PRCID process id (low DWORD)
ECX : PRCID process id (high DWORD)
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : INT no of dataspaces
EBX :
ECX :
EDX :
```

Parameter: 'taskid' Prozeßbezeichner der Task, deren Bestand an Datenräumen geprüft wird. 'no' Anzahl der Datenräume, die diese Task besitzt. Falls der Prozeßbezeichner keinen gültigen Prozeß bezeichnet, wird 0 geliefert.

block process

Code: 36

Wirkung: Blockiert den angegebenen Prozeß; d.h. er wird solange von der Zuteilung von Rechenzeit ausgeschlossen, bis er eine Botschaft oder einen Interrupt erhält. Dieser Systemaufruf erfordert mindestens das Privileg 1.

Register vor INT 7

```
 AL : 36
EBX : PRCID process id (low DWORD)
ECX : PRCID process id (high DWORD)
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'taskid' Prozeßbezeichner des Prozesses, der blockiert werden soll.

unblock process

Code: 37

Wirkung: Entblockiert den angegebenen Prozeß; d.h. falls er von der Zuteilung von Rechenzeit ausgeschlossen war, wird diese Blockierung aufgehoben. Falls keine Blockade vorlag, hat der Aufruf keine Wirkung. Dieser Systemaufruf erfordert mindestens das Privileg 1.

Register vor INT 7

```
 AL : 37
EBX : PRCID process id (low DWORD)
ECX : PRCID process id (high DWORD)
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'processid' Prozeßbezeichner des Prozesses, der entblockiert werden soll.

set priority

Code: 38

Wirkung: Legt die Priorität des angegebenen Prozesses neu fest. Dieser Systemaufruf erfordert mindestens das Privileg 1.

Register vor INT 7

```
 AL : 38
EBX : PRCID process id (low DWORD)
ECX : PRCID process id (high DWORD)
EDX : INT priority
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'processid' Prozeßbezeichner des Prozesses, dessen Priorität neu festgelegt wird. 'prio' neue Priorität dieses Prozesses. Dieser Wert, der zwischen 0 und 15 liegen darf, gibt an, jede wievielte mögliche Zeitscheibe der betroffene Prozeß erhalten soll. Folglich stellt 0 (keine Zeitscheibe auslassen) die höchste, 15?? (14 Zeitscheiben auslassen, 15te zuteilen) die niedrigste Priorität dar.

switch process

Code: 50

Wirkung: Der aufrufende Prozeß informiert den Systemkern, daß er in der aktuellen Zeitscheibe keine Rechenzeit mehr benötigt. Der Kern teilt dann sofort dem nächsten Prozeß die CPU zu.

Register vor INT 7

```
 AL : 50
EBX :
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: -

Bemerkung: Gerade Realprozesse (mit Priorität 0) sollten in jeder geeigneten Situation mit diesem Systemaufruf den Scheduler veranlassen, den Prozessor weiterzureichen.

Beispiel:

```
DO
  mov   ebx,'GOL'
  mov   al,sys_prc_id
  int   7
  mov   [log_prc],eax
  mov   [log_prc+4],ebx
  test  eax,eax
  EXITNZ
  mov   al,switch_prc
  int   7
REPEAT
OD
```

set alarm

Code: 51

Wirkung: Der aufrufende Prozeß informiert den Systemkern, daß er im angegebenen Takt (alle x Millisekunden) einen Interrupt erhalten möchte. Der Interrupt wird als INT 2 (NMI) geliefert. Dieser NMI hat natürlich nichts mit dem Hardware NMI des Prozessors zu tun (der wird nur vom Systemkern behandelt).

Register vor INT 7

```
 AL : 51
EBX : INT period
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'period' Zeit in Millisekunden zwischen zwei Interrupts.

special operation

Code: 52

Wirkung: Der aufrufende Prozeß fordert mit diesem Aufruf vom Systemkern eine besondere in Register EBX textuell spezifizierte Leistung.

Register vor INT 7

```
 AL : 52
EBX : TEXT spec
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'spec' Kurzbezeichnung für die angeforderte Leistung. Die über diesen Systemaufruf abrufbaren Leistungen werden in einer gesonderten Beschreibung zu Hardwaretreibern für ein L3 System erörtert.

get real time

Code: 53

Wirkung: Der Aufrufer erhält vom Systemkern den aktuellen Wert der Systemuhr. Der Wert wird in Millisekunden geliefert. Er sollte beispielsweise zu Zeitmessungen herangezogen werden. Für Datum/Uhrzeitabfragen sollte **get date time** benutzt werden.

Register vor INT 7

```
 AL : 53
EBX :
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : lower DWORD of real time
EBX : upper DWORD of real time
ECX :
EDX :
```

Parameter: 'real time' enthält den Wert »Millisekunden seit Christi Geburt« (1.Januar 1, 0:00 Uhr)

get date time

Code: 54

Wirkung: Der Aufrufer erhält vom Systemkern den aktuellen Wert der Systemuhr. Der Wert wird in BCD Codierung, passend für Datum/Uhrzeitabfragen geliefert.

Register vor INT 7

```
 AL : 54
EBX :
ECX :
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX : BCD time
EBX : BCD date
ECX :
EDX :
```

Parameter: 'time':

```
Stunde    := h2 * 10 + h1
Minute    := m2 * 10 + m1
Sekunde   := s2 * 10 + s1
Sommerzeit  :=  (d = 1)
```

'date':

```
c2 c1 y2 y1 m2 m1 d2 d1

Jahrhundert := c2 * 10 + c1
Jahr        := y2 * 10 + y1
Monat       := m2 * 10 + m1
Tag         := d2 * 10 + d1
```

control out

Code: 61

Wirkung: Gibt ein in Register BL übergebenes Byte aus. Es wird direkt in den Bildschirmspeicher geschrieben. Die erste Ausgabeposition ist (0,5), weitere Kontrollausgaben werden auf Position (2,5), (4,5) usw. bis maximal (78,10) ausgegeben. Maximal werden also 6 Bildschirmzeilen von den Kontrollausgaben überschrieben.

Register vor INT 7

```
 AL : 61
 BL : BYTE char
ECX :
```

```
EDX :
ESI :
EDI :
```

Register nach INT 7

```
EAX :
EBX :
ECX :
EDX :
```

Parameter: 'char' kann ein beliebiges ASCII Zeichen sein.

A Anpassung von Terminals

A.1 Einführung

Die Anpassung von Terminals wird notwendig, wenn Terminals an ein L3 Multiuser- System angeschlossen werden sollen, für die keine Anpassungen in der Task **configurator** vorhanden sind. Um die Erfordernisse einer Anpassung zu verstehen, lohnt ein kurzer Blick auf das Ziel einer solchen Anpassung und die verschlungenen Pfade zwischen Tastatur, Task und Bildschirm.

Ziel ist, daß Tastaturen und Bildschirme stets ein Verhalten zeigen sollen, das einem PC möglichst ähnlich ist. Da ggf. auch DOS Programme ausgeführt werden, wird erwartet, daß sich eine Tastatur wie eine PC Tastatur mit 101/102 Tasten verhält und ein Bildschirm wie ein PC Bildschirm im Textmodus aussieht. Er habe 25 Zeilen zu 160 Byte in denen 80 Zeichen/Attributpaare gespeichert sind.

Handelsüblich sind zwei Typen von Terminals: *PC Term* und *ASCII Terminals*. Die Klasse *PC Term* umfaßt neben echten PC's, für die eine Terminalemulation im L3 Diskettensatz beiliegt, auch viele neuere Terminals, deren Firmware neben diversen Emulationen auch den PC Term Modus anbietet. Dieser Modus bewirkt, daß Tastatureingaben als »Make- Breakcode« an das L3 System gesendet werden und für Ausgaben der vollständige PC Zeichensatz zur Verfügung steht.

Exkurs: Scancode und Zeichencode

Der Scancode ist die Tastennummer der gedrückten Taste. Die *ESC Taste* hat die Nummer 1, anschließend folgt die obere Ziffernreihe ab Taste Nummer 2 usw. Diese Tastennummer wird vom Tastaturprozessor im Zusammenhang mit den Umschalttasten *SHIFT*, *CNTR* und *ALT* zu einem Zeichencode ausgewertet. Zum Beispiel liefert ein Druck auf die Taste Nummer 30 den Zeichencode 97, das `a`, ein Druck auf die Taste Nummer 30, während die Taste Nummer 42 (*SHIFT*) gedrückt ist, 65, das `A` und so weiter.

Vorsicht Falle: Terminals, die im PC Term Modus angeschlossen sind, können nicht mit dem Softwareprotokoll *XON/XOFF* betrieben werden, da die für *XON/XOFF* benutzten Codes 17 und 19 die Scancodes von `w` und `r` sind. Falls ein Softwareprotokoll benötigt wird, kann *PC Handshake* benutzt werden.

Ende Exkurs

Ein ASCII Terminal ist ein Terminal, das für jedes Zeichen einen Code oder eventuell eine *Escape-Sequenz* sendet oder erwartet. Diese Terminals stammen aus dem UNIX- oder Großrechnerbereich. Sehr weite Verbreitung haben z.B. VT 200 Terminals und kompatible. Für diesen Typ sind Umsetztabellen in L3 vorhanden. Da grundsätzlich gesehen ASCII Terminals nicht die nötigen Eigenschaften für L3 besitzen (nur 24 Zeilen, evtl. keine Blockgrafik etc.) ist ihre Verwendung nicht empfehlenswert.

Typ	Umsetzung
ASCII Keyb	asciizeichen(kette) > ascii,scan
ASCII Display	zeichen/attr(kette) > ascii(kette)
PC Keyboard	ascii,scan > ascii,scan
PC Display	–

Beteiligte Komponenten und Typ der Botschaften zwischen den Komponenten lassen sich folgender Skizze entnehmen:

```
                                         GDP Bytestream

Tastatur      --->   Treiber SIxx     --->   Keyb Manager/Tabelle

                                                   GDP Scan,Taste
                                                          \
                                                    RTS oder direkt
                                                          /
                                                    GDP Alpha8

Bildschirm    <---   Treiber SOxx     <---   Dsp Manager/Tabelle

                                         GDP Bytestream
```

Die Einstellung eines bestimmten Terminaltyps erfolgt interaktiv in der Task **configurator**. Nach Festsetzung aller Einstellung werden mit den Standarddatenräumen **display.com.bin** und **keyboard.xx.bin** und den eingestellten Tabellen Keyboard- und Displaymanager erzeugt.

Um ein Terminal anzupassen, für das es noch keine Umsetztabellen gibt, ist bei den Funktionen 7 und 8 der Typ `<neue tabelle>` auszuwählen. Als Vorbild kann häufig ein Typ gewählt werden, aus dessen Tabellen viele Umsetzungen bestehen bleiben können.

A.2 Tastaturanpassung

Eine Tastaturanpassung wird durch die Zusammenstellung einer Umsetztabelle bewerkstelligt. Diese Tabelle gibt an, wie Codes, die beim Drücken einer Taste gesendet werden im Keyboardmanager zu Paaren **scancode**, **zeichencode** umgesetzt werden sollen. Diese Paare (scan, zeichen) werden dann dem ELAN Laufzeitsystem der Task oder dem direkten Partner des Keyboardmanagers zugestellt.

Im **configurator** (Option 7) sind Umsetztabellen für PC Terminals und für ASCII Terminals enthalten:

A.2.1 PC Terminals

Die Tabellen haben für jede Taste eine Angabe der Art:

```
Modus              System-Code      scan        ascii
    ....
8                  8           :   255,          255
7                  8           :     8,            ""123""
5                  8           :   126,            0
2                  8           :     8,            "/"
1                  8           :     8,            "7"
        ....
```

Der Modus beschreibt die Umschalttasten:

Modus	gedrückte Umschalttasten
1	keine
2	SHIFT
3	CNTL
4	SHIFT & CNTL
5	ALT
6	SHIFT & ALT
7	CNTL & ALT
8	SHIFT & CNTL & ALT

A.2.2 ASCII Terminals

Die Tabellen enthalten Angaben der Art:

```
ascii          scan        Tastatursequenz
            ....
    24,          45;        ""24""
    25,          44;        ""25""
    26,          21;        ""26""
   253,         255;        ""27"[34"126""
     0,         134;        ""27"[33"126""
     0,         133;        ""27"[32"126""
     0,          68;        ""27"[31"126""
   255,         255;        ""27"[29"126""
            ....
```

Hier wird für jede vom Terminal gesendete Sequenz angegeben, in welchen scancode, asciicode sie umgesetzt werden soll.

Auf der Diskette mit Beispielprogramm findet sich ein einfaches ELAN Programm namens `tasttest`. Mit die Programm an der CONSOLE kann man sich einfach die Codes, die beim Drücken von Tasten(kombinationen) erzeugt werden, ansehen.

A.3 Hot Key

Für jedes Terminal müssen insbesondere auch **HotKeys** definiert werden. Diese Tasten(kombinationen) sind für **forced link**, Anhalten und Wiederanlauf der Bildschirmausgabe etc. erforderlich.

forced link bewirkt das Ankoppeln der Terminaltask.

Der Hotkey **stop** bewirkt das der Displaymanager die laufende Ausgabe pausiert. Das Anhalten der Bildschirmausgabe wird durch Drücken irgendeiner Taste beendet.

scan ascii	Wirkung
255, 255	forced link
255, 254	Bildschirm-Refresh
255, 253	Stop, Ausgabe anhalten

A.4 Bildschirmanpassung

In den Zeichentabellen des *configurators* werden nötige Zeichenumsetzungen, eine angemessene und mögliche Darstellung der Farbattribute und die Codierung von Aktionen wie »Rest der Zeile löschen« eingetragen.

Zeichenumsetzungen sind z.B. für die Zeichen nötig, die nicht dargestellt werden sollen. In diesem Falle ist eine Umsetzung von `code(char) => " "` einzutragen.

Folgende Leistungen können als **STEUERUNG <no>** in die Tabelle aufgenommen werden. Es handelt sich um Aktionen, die eventuell von der Terminalfirmware unterstützt werden. Aus Gründen der Lesbarkeit sind die Aktionen teilweise nicht als *Streuerung xy*, sondern »im Klartext« in die vorhandenen Tabellen eingetragen.

Bedeutung	<no>
set_cursor_abs	0
cursor_left	1
cursor_down	2
cursor_right	3
cursor_up	4
cursor_n_left	5
cursor_n_down	6
cursor_n_right	7
cursor_n_up	8
cleop	9
cls_to_cursor	10
cls	11
cleol	12
cll_to_cursor	13
cll	14
insert_one_line	15
insert_n_lines	16
delete_one_line	17
delete_n_lines	18
restore_cursor_and_att	r 19
save_cursor_and_attr	20
define_scroll_area	21
scroll_screen_1_up	22
cr	23
crlf	25
home	26
bell	27
special_televideo	254
terminal_init	255

Index