Arbeitsberichte aus der Informatik
(Technical Reports and Working Papers)
Fachhochschule Nordostniedersachsen
Volgershall 1, D-21339 Lüneburg
Phone: xx49.4131.677175 Fax: xx49.4131.677140

FINAL

F achhochschule Nordostniedersachsen
IN formatik
A rbeitsberichte
L üneburg

Systemnahe Programmierung

(3.,überarbeitete Auflage 1999)

Ulrich Hoffmann

FINAL, 9. Jahrgang Heft 1, Juli 1999, ISSN 0939-8821

Inhaltsverzeichnis

Literaturauswahl

[AHU] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

[BAU] Baumgarten, C.: **Systemnahe Programmierung mit Borland Pascal**, Vieweg, 1994.

[BLU] Blum, N.: **Theoretische Informatik**, Oldenbourg, 1998.

[BP7] **BORLAND PASCAL MIT OBJEKTEN 7.0 Dokumentation**, Borland, 1992.

[BRA] Brause, R.: **Betriebssysteme, Grundlagen und Konzepte**, Springer, 1997.

[Bua] Beck, M; Böhme, H.; Dziadzka, M.; Kunitz, U.; Magnus, R.; Verworner, D.: **Linux-Kernel-Programmierung**, 2. Aufl., Addison-Wesley, 1994.

[COM] Comer, D.: **Computernetzwerke und Internets**, Prentice Hall, 1998.

[COX] Cox, B.J.: **Object Oriented Programming**, Addison-Wesley, 1986.

[CUS] Custer, H.: **Inside WINDOWS NT**, Microsoft Press, 1993.

[DEW] Dewdney, A.K.: **Der Turing Omnibus**, Springer 1995.

[GKP] Graham, R.L.; Knuth, D.E.; Patashnik, O.: **Concrete Mathematics**, Addison-Wesley, 1989.

[G/J] Ghezzi, C.; Jazayeri, M.: **Konzepte von Programmiersprachen**, Oldenbourg, 1989.

[H/D] Herschel, R.; Dieterich, E.-W.: **Turbo Pascal 7.0**, Oldenbourg, 1994.

[H/H] Herrtwig, R.G.; Hommmel, G.: **Nebenläufige Prozesse**, Springer, 1994.

[H/P] Hennessy, J.L; Patterson, D.A.: **Rechnerarchitektur**, Vieweg, 1994.

[HEN] Hendrich, N.: **Java für Fortgeschrittene**, Springer 1997.

[HOF] Hoffmann, U.: **Einführung in die systemnahe Programmierung**, Walter de Gruyter, 1990.

[ISP] International Organisation for Standardization: **Specification for Computer Programming Language Pascal**, ISO 7185, 1985.

[J/W] Jensen, K; Wirth, N.: **PASCAL. User manual and report**, 3. Aufl., Springer, 1985.

[KAI] Kaier, E.: **Delphi Essentials**, Vieweg, 1997.

[MÄR] Märtin, C.: **Rechnerarchitektur**, Hanser, 1994.

[MAT] Matthes, W.: **Intel's i486**, Elektor, 1992.

[M/O] Maekawa, M.; Oldehoeft, A.E.; Oldehoeft, R.R.: **Operating Systems, Advanced Concepts**, Benjamin Cummings, 1987.

[MES] Messmer, H.-P.: **Pentium**, Addison-Wesley, 1994.

[NEL] Nelsin, R.P.: **Programmierhandbuch 80386**, Vieweg 1989.

[O/W] Ottmann, T.; Widmayer, P.: **Algorithmen und Datenstrukturen**, 2. Aufl., B.I. Wissenschaftsverlag, 1992.

[P/K] Prinz, P.; Kirch-Prinz, U.: **Objektorientiert programmieren mit ANSI C++**, Prentice Hall, 1998.

[P/Z] Pratt, T.; Zelkowitz, M.: **Programmiersprachen**, Prentice Hall, 1997.

[STR] Stroustrup, B.: **Die C++-Programmiersprache**, 2. Aufl., Addison-Wesley, 1990.

[TAN] Tanenbaum, A.S.: **Moderne Betriebssysteme**, Hanser, 1994.

[THI] Thies, K.-D.: **80486 Systemsoftware-Entwicklung**, Hanser, 1992.

[TIS] Tischer, M.: **Turbo Pascal intern**, (Version 5.0 und 5.5), Data Becker, 1989.

[WAR] Warken, E.: **Delphi 4**, Addison-Wesley, 1999.

[W/C] Wilson, L.B.; Clark, R.G.: **Comparative Programming Languages**, Addison-Wesley, 1988.

[WIR] Wirth, N.: Gedanken zur Software-Explosion, Informatik-Spektrum, 17, 5-10, 1994.

1 Einleitung

Ein **Datenverarbeitungssystem (DV-System)**, sei es ein einzelner Personal Computer (PC) oder ein Großrechner, ist logisch in hierarchischen Schichten um die Hardware organisiert (Abbildung 1-1). Die prinzipiellen Funktionen der einzelnen Schichten lassen sich wie folgt beschreiben:

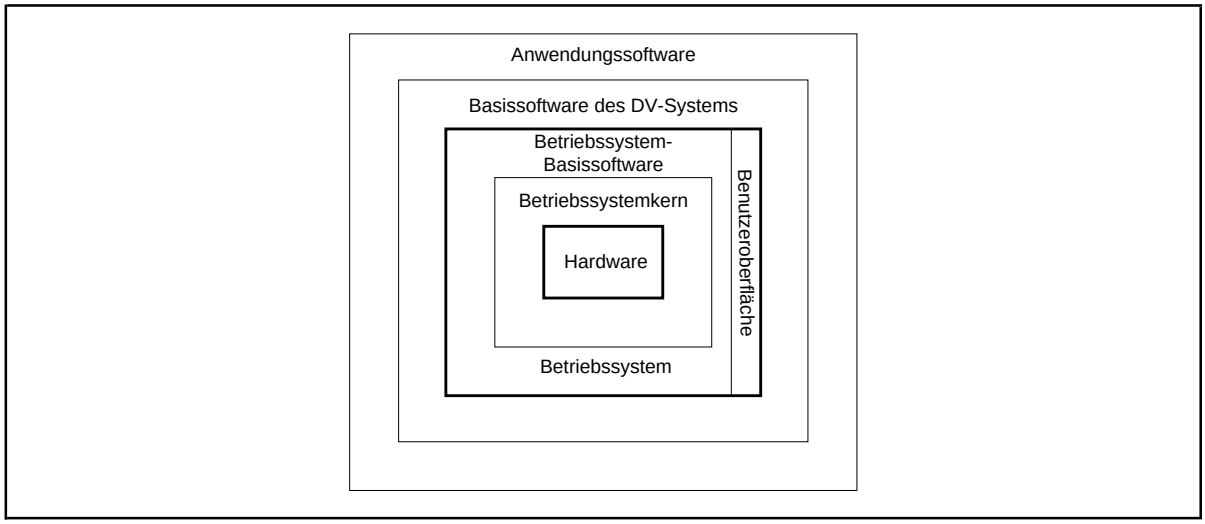


Abbildung 1-1: Schichtenmodell eines DV-Systems

- Das **Betriebssystem** kann als Puffer zwischen der realen Hardware und den abstrakten Algorithmen und Anwendungen gesehen werden. Aus Benutzersicht teilt es die i.a. knappen Betriebsmittel (Ressourcen) eines Systems auf die sich um deren Benutzung konkurrierenden Anwender auf und verwaltet den Zugriff. Aus Programmierersicht stellt es Befehls- und Systembibliotheken bereit, die die Möglichkeiten der "nackten" Hardware erweitern und den Zugriff auf die Rechnerkomponenten erleichtern. Häufig wird (zumindest von den Funktionen her) zwischen Betriebssystemkern und Betriebssystem-Basissoftware unterschieden:

  -- Der **Betriebssystemkern** verwaltet die einzelnen Komponenten der Hardware wie Prozessor, Arbeitsspeicher, Peripheriegeräte, logische Dateien und Prozesse. Er stellt den weiter außen liegenden Schichten eine einheitliche Softwareschnittstelle zur Hardware bereit. Hinzu kommen Funktionen zur Prozeßkommunikation und Prozeßsynchronisation, zum Prozeßmanagement, zur Adreßraumverwaltung und Treiber für die Rechnerkomponenten und angeschlossenen Geräte.

  -- Die **Betriebssystem-Basissoftware** verwaltet komplexe logische Funktionsbereiche oberhalb des Betriebssystemkerns und ist bei entsprechendem Design hardwareunabhängig. Zur Analyse und Ausführungsinitialisierung eines Benutzerkommandos an das Betriebssystem ist eine entsprechende meist graphische Benutzeroberfläche vorgesehen. Weitere typische Teile sind z.B. Auftragsverwaltung (Jobmanagement) bei

Großrechnern, Objektmanagement, Dateimanagement, Datenkommunikationssystem und Netzwerksteuerung, Transaktionssystem und Systemadministrationssystem (Accounting).

- Die **Basissoftware des DV-Systems** umfaßt eine Menge systemnaher anwendungsneutraler Programme, die im wesentlichen Hilfsmittel zur Anwendungsprogrammentwicklung darstellen. Dazu gehören Texteditoren, Sprachübersetzer (Compiler), Binder, Testhilfeprogramme (Debugger), Entwicklungsumgebungen, Office-Softwaresystem, Datenbankverwaltungssysteme, Transaktionsmonitore, Dienstprogrammen zur Datensicherung und -archivierung, Sortierprogramme usw.

- Die **Anwendungsprogramme** lassen sich als anwendungsbezogene systemneutrale Softwareteile charakterisieren.

Im Mittelpunkt der **Systemprogrammierung** steht die Entwicklung, Anpassung, parametergesteuerte Generierung und Betreuung der gesamten Betriebssystemsoftware eines DV-Systems, sowie eine Unterstützung bei der Anpassung von Basissoftware und Anwendungssoftware an ein Betriebssystem. Früher wurde mit dem Begriff Systemprogrammierung Programmierung in einer Assembler-Sprache, also in einer auf einen speziellen Maschinentyp abgestellten Sprache, assoziiert. Systemnahe Programme erfordern eben den direkten Zugriff auf die darunterliegende Hardware, und Zeit- und Speichereffizienz sind dabei grundlegende Faktoren. Seitdem systemnahe Software mit Hilfe geeigneter höherer Programmiersprachen formuliert werden kann, ist die Bedeutung der Assembler-Programmierung auch in der Systemprogrammierung stark zurückgegangen. Folglich reduziert sich der Assembler-Anteil im vorliegenden Text auf wenige Beispiele. Da nun hauptsächlich höhere Programmiersprachen in der Systemprogrammierung eingesetzt werden, erscheint es sinnvoll, die Konzepte dieser Sprachen, insbesondere die Abbildung der Datendefinitionen auf interne Darstellungsformate, die Realisierung der Prozedurkonzepte und der objektorientierten Methoden der Programmiersprache, genauer zu studieren. Gerade für Wirtschaftsinformatikerinnen oder Wirtschaftsinformatiker, die sich bei Programmentwicklungen primär mit Anwendungsproblemen beschäftigen, können diesen Kenntnisse auch einen Gewinn für die Anwendungsprogrammierung darstellen. Denn diese Kenntnisse fördern ein allgemeines Verständnis für die Vorgänge im Rechner und für die Einsatzmöglichkeiten der jeweiligen Sprachmittel. Natürlich werden weitere Schwerpunkte auf spezifischen Methoden der Systemprogrammierung liegen, wobei eine (subjektive) Auswahl getroffen werden muß. Der **Inhalt des vorliegenden Textes** läßt sich wie folgt umreißen:

- Beschreibungen einiger grundlegender Aspekte von Rechnerarchitekturen

- Sprachkonzepte, die heute in höheren Programmiersprachen üblich sind, und ihre Abbildung auf eine maschinennahe Umgebung

- systemnahe Aspekte der Programmentwicklung, Modularisierung und Bibliothekskonzepte

- exemplarische Behandlung der Realisierungen wichtiger Konzepte der objektorientierten Programmierung

- ausgewählte Datenstrukturen, Methoden und Algorithmen der Systemprogrammierung, die auch Bedeutung für die anwendungsorientierte Programmierung haben

- spezielle Fragestellungen der systemnahen Programmierung wie die (exemplarische) Realisierung eines Prozeßmodells in einer objektorientierten Multitask-Umgebung, Prozeßsynchronisation und -kommunikation und ihre Anwendung in ausgewählten Beispielen.

Der folgende Text versteht sich keineswegs als Lehrbuch einer speziellen Programmiersprache. Er behandelt allgemeingültige **sprachübergreifende Methoden**. Der Einsatz einer ausgewählten Programmiersprache dient lediglich der exakten Formulierung dieser Mechanismen. Die meisten Beispiele sind in Pascal formuliert, und zwar in dem in der PC-Welt weitverbreiteten Dialekt Borland Pascal ([H/D], [P/Z]) bzw. Object Pascal. Durch die Wahl dieses Sprachdialekts entfallen einerseits weitgehend die Eingeschränkungen, die der Verwendung von Standard-Pascal (vgl. [ISP]) für die Systemprogrammierung und die in modularisierenden und objektorientierten Entwicklungskonzepten entgegenstehen würden. Andererseits zeichnet sich Pascal als "Stammvater" einer ganzen Klasse von Programmiersprachen durch seine stringenten Konzepte aus (z.B. sein Typkonzept, kontrollierte Übergabemethoden für Unterprogramme und eine Reihe sinnvoller Einschränkungen, auf die sich ein Compiler zur Konsistenzprüfung stützen kann), die sich auch in anderen gängigen Sprachen (z.B. Modula-2, Ada, Oberon, C++, Java) wiederfinden. Die Klarheit des Sprachkonzepts (und damit eine gewisse Ästhetik) und die gute Lesbarkeit der Quellprogramme unterscheidet Pascal von anderen Programmiersprachen, insbesondere auch von C, C++ oder Java. Dadurch kann in der Darstellung eine verstärkte Hinwendung zu den inhaltlichen Fragestellungen eines Problems erreicht und dessen Überdeckung durch sprachtechnische Hindernisse der Programmiersprache vermieden werden. Hinzu kommen eine sich im PC-Bereich abzeichnende Renaissance dieser Sprache[1] und die hier verfügbaren sehr schnellen und komfortablen integrierten Entwicklungsumgebungen. Gelegentlich wird auf Eigenarten der weitaus maschinennäheren Programmiersprache C eingegangen (C kann als eine mehr oder weniger "komfortable Assembler-Sprache" angesehen werden und wird auch als *low-level high-level*-Programmiersprache bezeichnet[2].

Im folgenden ist mit PASCAL (in Großbuchstaben) ***im Zusammenhang sowohl Standard-Pascal als auch Borland Pascal und Object Pascal*** (der Firma Inprise, vormals Borland) ***gemeint, die Bezeichnung Pascal*** (in Kleinbuchstaben) ***steht für Standard-Pascal. Pascal-Schlüsselwörter werden meistens in Großbuchstaben geschrieben, Schlüsselwörter der Pascal-Dialekte mit großen Anfangsbuchstaben.***

---

[1] Beispielsweise generiert die in der PC-Welt weitverbreitete Entwicklungsumgebung Delphi optimierten Object Pascal-Code für alle WINDOWS Systeme.

[2] Die Tatsache, daß C bzw. C++ als Ersatz für die Assembler-Programmierung in der Praxis weitverbreitet ist, läßt sich nicht durch die "Qualität" dieser Programmiersprachen begründen, eher in der Verbreitung des Betriebssystems UNIX, in das C integriert ist. Und der Einsatz dieser Sprachen in der Anwendungsprogrammierung zeugt eher von einem (leider weitverbreiteten) Unverständnis für moderne Programmiersprachenkonzepte ([WIR]).

2 Das Modell eines Rechners

Das "Grundmodell" eines DV-Systems orientiert sich am **(von-Neumann-) Universalrechnermodell** (1946), das ein DV-System in einzelne funktionale Komponenten aufteilt und in Abbildung 2-1 zu sehen ist (vgl. [MÄR]). Die wesentlichen Komponenten dieses Modells sind

- das **Steuerwerk** (Leitwerk, control unit), das neben dem Befehlsregister und dem Befehlszähler eine Einheit zur Befehlsentschlüsselung und eine Einheit zur Steuerung der Befehlsausführung enthält; das Steuerwerk überwacht und steuert die Abläufe im Rechner

- das **Rechenwerk** (arithmetic logic unit) zur Ausführung arithmetischer und logischer Operationen und Shift-Operationen; dazu enthält es eine Reihe spezieller gleichgroßer Register, deren Inhalte miteinander verknüpft oder durch Operationen verändert werden können

- das **Speicherwerk**, bestehend aus einem in gleichgroße, adressierbare Speicherzellen eingeteilten **Arbeitsspeicher**, ein Adreßregister und ein Datenregister; das Speicherwerk dient der Speicherung von Daten und auszuführenden Programmen

- das **Ein-/Ausgabewerk** zur Übernahme bzw. Übergabe von Programmen und Daten von bzw. an externe Speichereinheiten

- der (System-) **Bus**, der die Teilwerke des Rechners miteinander verbindet und bitparallel jeweils eine Adresse, ein Datenwort oder einen Befehl überträgt. Daneben gibt es **Steuerleitungen** zur Übertragung von Steuersignalen und Rückmeldungen zwischen dem Steuerwerk und den anderen Rechnerteilwerken.

Wesentliche Merkmale der klassischen von-Neumann-Architektur sind weiterhin:

- Die Struktur des Rechners ist unabhängig von dem zu bearbeitenden Problem. Zur Problemlösung wird in den Arbeitsspeicher eine Bearbeitungsvorschrift, das (problemabhängige) **Programm**, geladen. Dieses setzt sich aus Befehlen des **Befehlssatzes des Rechners** zusammen. Der Befehlssatz enthält neben den arithmetischen, logischen und Shift-Befehlen bedingte und unbedingte Sprungbefehle sowie Speicherbefehle und Befehle für das Ein- und das Ausgabewerk.

- Für Programme und Daten wird ein einheitlicher Speicher verwendet, d.h. für Programme bzw. Daten werden nicht spezielle Speicherbereiche reserviert. Am Inhalt einer Speicherzelle ist nicht erkennbar, ob es sich um ein Datenelement oder den Teil eines Befehl handelt. Jede Speicherzelle ist über eine eindeutige **Adresse** ansprechbar. Die Speicherzellen sind fortlaufend numeriert. Die Lage eines Befehls im Speicher wird durch seine Adresse identifiziert. Die Befehle eines Programms belegen i.a. aufeinanderfolgende Speicherzellen. Bei Sprungbefehlen wird die Programmausführung an der Sprungzieladresse fortgesetzt. Wie bei einem Befehl wird die Lage eines Datenelements im Speicher durch seine Adresse bestimmt. Der Zugriff auf einen Speicherzelleninhalt erfolgt dadurch, daß die Adresse der entsprechenden Speicherzelle in das Adreßregister des Speicherwerks geladen wird. Bei

einem Lesezugriff wird der Inhalt des adressierten Speicherbereichs in das Datenregister übertragen; bei einem Schreibzugriff wird der Inhalt des Datenregisters an die adressierte Speicherstelle geschrieben.

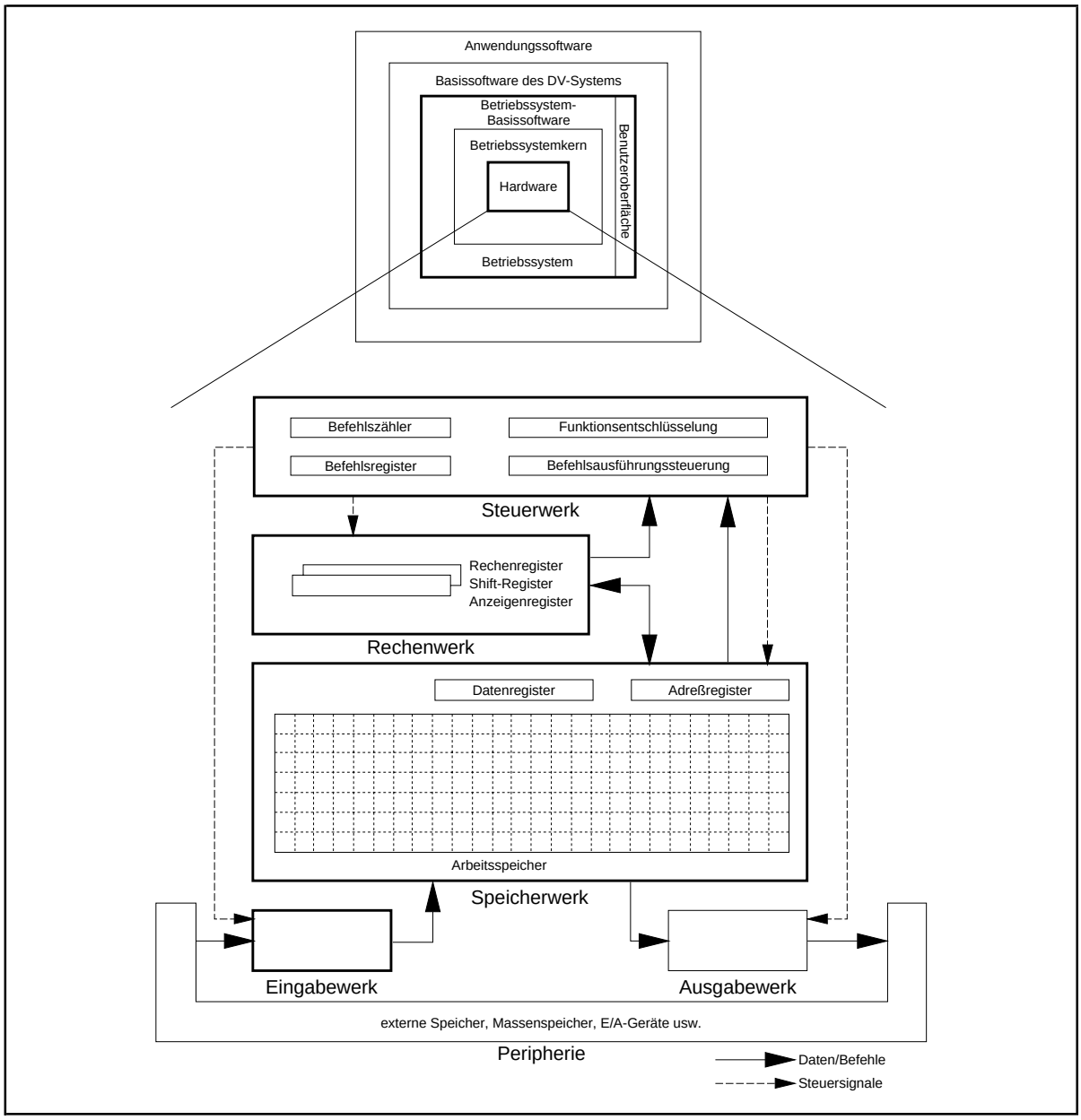


Abbildung 2-1: Grundmodell eines Rechners

Auch gängige moderne Rechnersysteme orientieren sich noch grundsätzlich an diesem Architekturprinzip, wobei die einzelnen Teilwerke zu größeren Funktionseinheiten physisch zusammengefaßt bzw. jeweils mehrfach implementiert sind. Beispielsweise bilden Steuer- und Rechenwerk den **Prozessor (CPU, central processing unit)** eines Rechners. Arbeitsspeicher, Prozessor und eventuell weitere Einheiten zur Steuerung des Rechnerablaufs werden in der

Zentraleinheit zusammengefaßt bzw. sind auf einer **Systemkarte** integriert. Zentraleinheit, Teile der Peripheriegeräte wie Festplatten und Teile des Ein-/Ausgabewerks wie Tastaturinterface, diverse Controller oder Diskettenlaufwerke befinden sich in einem Rechnergehäuse.

Ein Beispiel der Realisierung des Konzepts zeigt Abbildung 2-2: die Architektur eines typischen **Arbeitsplatzsystems** ([MÄR]); die Arbeitsweise eines Rechners, der sich an dieser Architektur orientiert, wird in den folgenden Kapiteln erläutert (vgl. [H/P], [HOF]). Die Systemkomponenten sind über eine lokale Busschnittstelle an einen CPU-Block mit integrierter Gleitpunkteinheit, MMU (memory management unit, zur Organisation des schnellen Speicherzugriffs) und Cache (schneller Zwischenspeicher) gekoppelt. Der DRAM-Arbeitsspeicher (dynamic random access memory) wird über einen DRAM-Controller angesteuert. Graphik-Controller und DMA-Controller (direct memory access) für den Arbeitsspeicherzugriff von Laufwerken und anderen datenintensiven Peripheriegeräten können ebenfalls über das Bussystem angesteuert werden. CPU-Unterbrechungen durch externe Geräte werden von einem Interrupt-Controller verwaltet und an die CPU weitergeleitet.

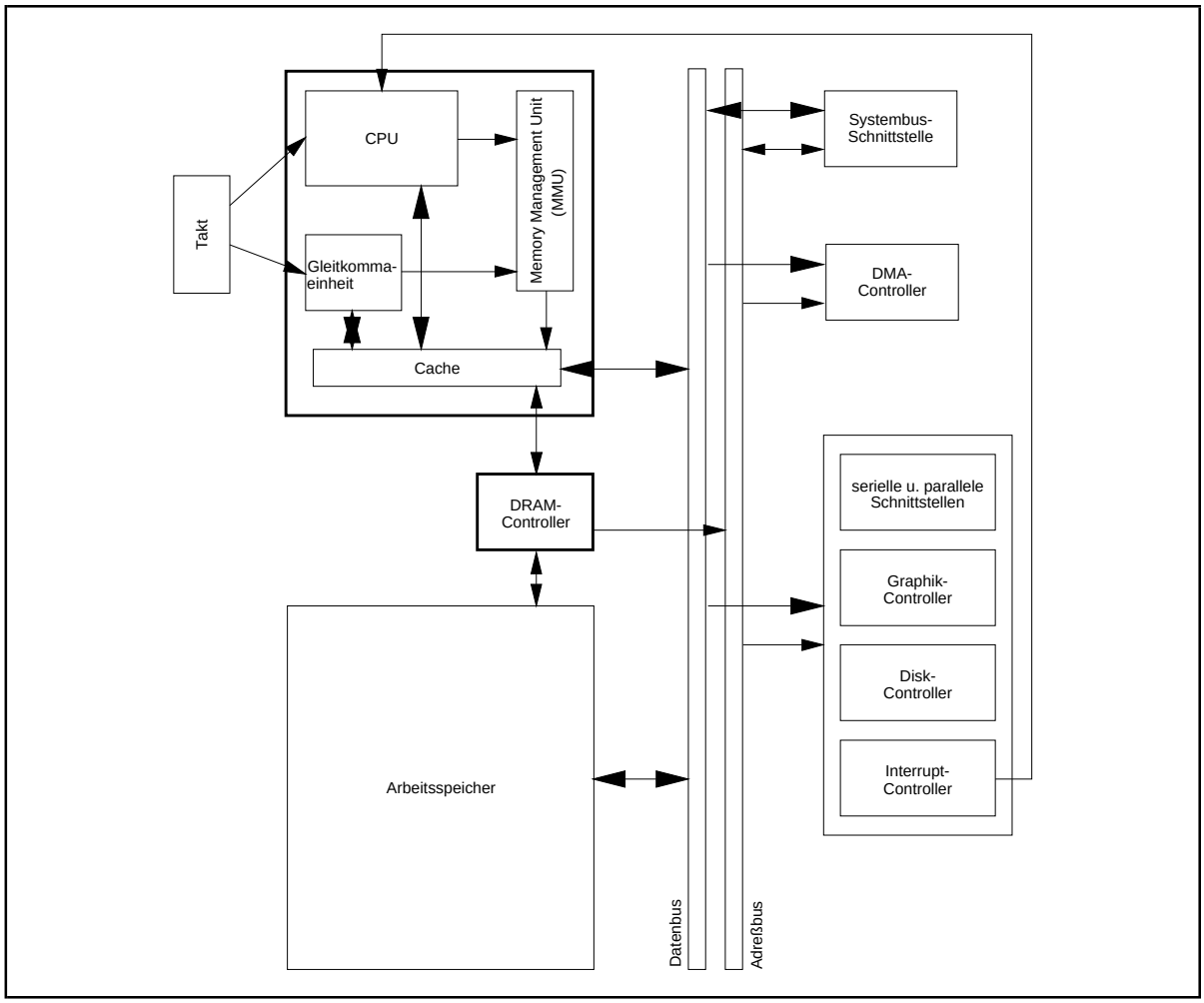


Abbildung 2-2: Exemplarische Architektur eines Arbeitsplatzsystems, ([MÄR])

2.1 Funktionsweise eines Rechners

Die einzelnen Komponenten eines Rechnersystems müssen miteinander in ihrem zeitlichen Ablauf synchronisisert werden. Die Zentraleinheit verfügt mit Hilfe eines Taktgebers über einen **Grundtakt (Grundzyklus, Prozessorzyklus)** in Form einer Rechteckschwingung mit vorgegebener Frequenz. Typische Taktfrequenzen im PC-Bereich sind mehrere Hundert Megahertz. Befehlsausführungszeiten, Speicherzugriffe usw. werden in Takten gemessen, wobei i.a. mehrere Takte pro Operation erforderlich sind, da die Durchführung einer Operation in der CPU in mehrere Teilphasen zerlegt wird. Durch Parallelisierung der Abläufe in den Teilkomponenten der CPU werden jedoch effektive Befehlsausführungszeiten im Eintaktbereich erzielt.

Intern sind die einzelnen Komponenten eines Rechners durch logische **Datenpfade** verbunden, über die Daten in den angegebenen Richtungen fließen können. Physisch sind diese Datenpfade durch Leiterbahnen realisiert, die als **Busse** bezeichnet werden. Die einzelnen Komponenten sind über **Steuerungen (Controller**, **Adapter)** an den Bussen angeschlossen. Es sind mindestens ein **Adreßbus** und ein **Datenbus** und **Steuerleitungen** vorgesehen. Der Prozessor spricht mit dem Inhalt des Adreßbusses einen bestimmten Arbeitsspeicherbereich oder ein Peripheriegerät an. Über den Datenbus fließen Daten zum adressierten Ziel. Die **Busbreite** (gemessen in Bits) von Adreß- bzw. Datenbus gibt an, wieviele Bits gleichzeitig, d.h. physisch parallel, über den jeweiligen Bus transportiert werden können. Dadurch bestimmt sie u.a. die Leistungsfähigkeit eines Rechnerssystems. Die Adreßbusbreite liegt außerdem den maximal ansprechbaren Adreßraum fest, der aber hardwaremäßig nicht in voller Ausbaustufe in Form des Arbeitsspeichers implementiert sein muß. Bei einer Adreßbusbreite von n Bits umfaßt der maximale Adreßraum die Adressen 0, ..., $2^n - 1$. Für die im PC-Bereich häufig vertretene INTEL-80x86-Prozessorfamilie gelten beispielsweise folgende Angaben:

Typ	Adreßbus-breite	Datenbus-breite	max. ansprech-barer Adreß-raum	Datenübertra-gungsrate des inter-nen Bus	Takt-frequenz (in MHz)
8086	20 Bits	8 Bits	1 MB[3]	2 MB/sec	5 - 10
80286	24 Bits	16 Bits	16 MB	4 MB/sec	8 - 10
80386	32 Bits	32 Bits	4 GB	20 - 40 MB/sec	bis 33
80486	32 Bits	32 Bits	4 GB	20 - 40 MB/sec	bis 100
Pentium	32 Bits	64 Bits	4 GB	bis 528 MB/sec	> 100

Der **Arbeitsspeicher (Hauptspeicher)** besteht aus fortlaufend numerierten Speicherplätzen. Die Nummer des Speicherplatzes ist seine **(physikalische) Adresse**. Ein Speicherplatz ist die kleinste ansprechbare Einheit, d.h. eine Adresse identifiziert einen Speicherplatz als

[3] Zur Erinnerung: Die in der DV-Welt üblichen Einheiten zur Angabe von Speichergrößen sind Byte, Kilobyte (KB, 1 KB = 1024 = 2^{10} Bytes), Megabyte (MB, 1 MB = 1024 KB = 2^{20} Bytes) bzw. Gigabyte (GB, 1 GB = 1024 MB = 2^{30} Bytes).

Ganzes und nicht etwa Teile davon. Typische **Adreßbreiten** sind je nach Rechnertyp 16, 20, 24, 25, 31, 32 oder 64 Bits. Jeder Speicherplatz hat dieselbe Größe; üblicherweise ist diese Größe ein **Byte** bestehend aus 8 **Bits**. ***Der Arbeitsspeicher enthält während des*** Rechnerbetriebs auszuführende Programme, die in Form von ***Daten und (Maschinen-) Instruktionen (Befehlen) vorliegen, Systemtabellen, die das Betriebssystem für seine Kontrollaufgaben pflegt, Datenbereiche für den Ablauf des Systems usw.*** Typische **(physikalische) Arbeitsspeichergrößen** sind 1 MB bis zu mehreren Hundert MB im PC-Bereich und bis zu mehreren Gigabyte im Großrechner-Bereich.

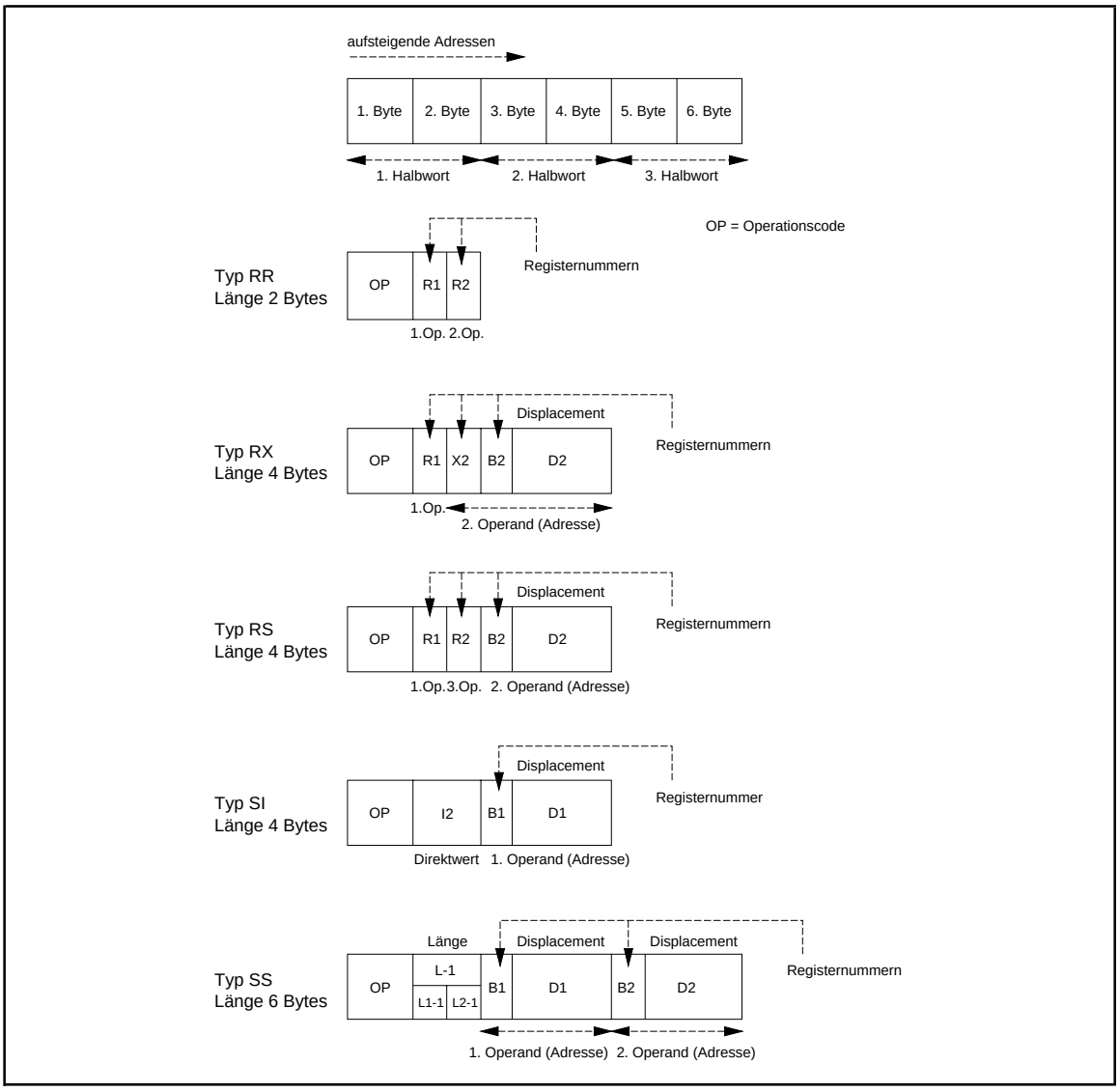


Abbildung 2.1-1: Instruktionsformate (IBM /370)

Eine (Maschinen-) Instruktion belegt im allgemeinen mehrere hintereinanderliegende Speicherplätze. Das Format einer Instruktion sieht einen Operationscode (meist ein Byte), Adreßangaben für die miteinander zu verknüpfenden Operatoren, Benennung von Registern,

deren Inhalte an der Operation beteiligt sind, und gelegentlich weitere Angaben wie Längenattribute der Operatoren oder eine genauere Spezifizierung der Operation vor. Abbildung 2.1-1 zeigt beispielsweise verschiedenen Instruktionsformate, die von den Großrechnern der IBM /370-Architektur bzw. der SNI-Großrechnerarchitektur verwendet werden.

Die Anzahl zusammenhängender Speicherplätze für Daten hängt von deren Typ ab. Die **Interpretation eines Speicherplatzinhalts** als Daten bzw. Instruktionen, also Programmteile, wird vom jeweiligen Programm bzw. der Steuerung des Prozessors bewerkstelligt.

Obwohl der Zugriff auf Arbeitsspeicherinhalte verglichen mit Zugriffen auf die Peripherie relativ schnell erfolgt, werden in modernen Systemen zusätzliche schnelle Zwischenspeicher (**Cache-Speicher**) mit entsprechender Zugriffstechnik zur Beschleunigung des Arbeitsspeicherzugriffs eingesetzt. Aus technischen Gründen ist die Größe eines Cache-Speichers jedoch sehr begrenzt. Bei einem Leseversuch des Arbeitsspeichers stellt die Steuerungskomponente des Cache-Speichers fest, ob sich das gewünschte Datum bereits im Cache-Speicher befindet. Wird so ein "Lese-Treffer" festgestellt, wird das Datum aus dem Cache-Speicher geholt. Bei einem "Nicht-Treffer" wird das Datum aus dem Arbeitsspeicher geladen und gleichzeitig ein Duplikat im Cache-Speicher abgelegt. Bei Schreibzugriffen auf Arbeitsspeicheradressen bzw. auf ihre korrespondierenden Duplikate im Cache-Speicher muß durch entsprechende Techniken Datenkonsistenz zwischen Cache- und Arbeitsspeicher sichergestellt werden:

- Bei der **Write-Through-Methode** werden im Fall, daß sich das zu verändernde Datum im Cache-Speicher als auch im Arbeitsspeicher befindet ("Schreib-Treffer"), beide Exemplare überschrieben. Bei einem Nicht-Treffer, d.h. nur der Arbeitsspeicher enthält das adressierte Datum, wird dieses nur im Arbeitsspeicher aktualisiert. Jeder Schreibzugriff macht sich somit auf dem Datenbus bemerkbar.

- Bei der **Write-Back-Methode** wird im Fall eines Schreib-Treffers zunächst nur das Exemplar im Cache-Speicher verändert und erst zu einem späteren Lese-Zeitpunkt im Arbeitsspeicher aktualisiert. Durch Setzen entsprechender Bits (**Dirty-Bits**) am Datum im Cache-Speicher wird gekennzeichnet, daß der Arbeitsspeicher und der Cache-Speicher hier nicht gleiche Kopien des Datums halten. Bei der späteren Aktualisierung wird das Dirty-Bit des Datums zurückgesetzt und damit Datenkonsistenz angezeigt. Der Nachteil der Write-Back-Methode gegenüber der Write-Through-Methode ist der größere Logik-Aufwand in der Steuerung (Controller), der ja die Dirty-Bits prüfen und verändern muß.

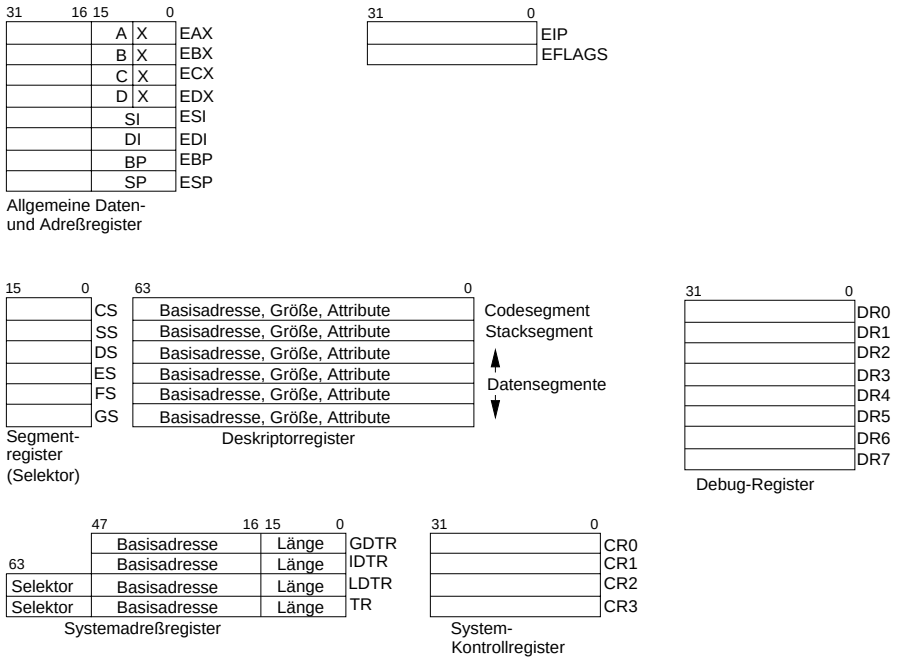
Eine entsprechende Cache-Speichertechnik wird auch bei Peripheriespeicherzugriffen eingesetzt, indem Teile des Arbeitsspeichers oder spezielle zusätzliche Speichereinheiten als Cache für Plattenbereiche reserviert werden, so daß die effektive Datentransferrate und -dauer zwischen Arbeitsspeicher und Peripherie reduziert werden.

Der **Prozessor (CPU, central processing unit)** besteht aus einer Menge von **Rechen-, Adressierungs-, Anzeigen-** und **Spezialregistern**, sowie logischen Schaltungen zur Steuerung des Ablaufs. Seine Aufgabe ist die Ausführung des im Arbeitsspeicher enthaltenen Programms. Abbildung 2.1-2 zeigt exemplarisch den CPU-Registersatz des INTEL-80486-Prozessors (ohne die Register der Gleitpunkt-Verarbeitungseinheit), der im wesentlichen auch für den INTEL-Pentium-Prozessor gilt.

Register, Cache-Speicher, Arbeitsspeicher und Peripheriespeicher bilden bezüglich des Datenaustauschs ein hierarchisch organisiertes Speichersystem, eine **Speicherhierarchie**, mit unterschiedlich langen Zugriffszeiten auf den jeweiligen Ebenen. Während der Datenzugriff auf ein Register und den Cache (bei Treffer) ca. 1 Taktzyklus benötigt, dauert er ein Mehrfaches beim Arbeitsspeicher und liegt um immense Größenordnungen höher beim Peripheriespeicher (vgl. [MÄR]). Es wird daher versucht, den Datenaustausch zwischen den Hierarchiestufen so gering wie möglich zu halten. Er erfolgt meist, wenn sich benötigte Daten nicht in der aktuell adressierten Hierarchieebene befinden und von einer CPU-ferneren Ebene geholt werden müssen, oder wenn Cache-Inhalte aus Kapazitätsgründen durch neue Daten überlagert werden müssen, oder wenn Daten aufgrund von Softwareanforderungen vom Cache in den Arbeits- oder Peripheriespeicher geschrieben werden müssen.

Die **Befehlsausführung** erfolgt in mehreren Teilphasen: Eine Instruktion wird aus dem Arbeitsspeicher gelesen, wobei ihre Adresse aus den Inhalten der Adressierungsregister und eines speziellen Anzeigenregisters, des **Befehlszählerregisters (program counter)**, ermittelt wird. Der Befehl wird interpretiert (dekodiert) und die nächste Instruktion adressiert. Die Operanden des Befehls werden, falls notwendig, in die CPU übertragen, über arithmetische oder logische Operationen miteinander verknüpft und die Resultate in Rechenregistern oder im Arbeitsspeicher abgelegt. Operandenadressen werden ebenfalls mit Hilfe der Adressierungsregister ermittelt.

Damit die Befehlsausführungsphasen der einzelnen Befehle zeitlich überlappt ablaufen können, ist die CPU i.a. in funktionale Teilkomponenten aufgeteilt, deren Arbeitsweise über den Prozessortakt synchronisiert werden kann. Während das Rechenwerk (mit Hilfe der Register) eine Operation ausführt, wird bereits der nächste Befehl gelesen und interpretiert. Diese Technik wird **Pipelining** genannt, siehe Abbildung 2.1-3 für ein fünfstufiges Pipelining. Auf diese Weise reduziert sich die durchschnittliche Instruktionsrate auf typischerweise 1 bis 4 Prozessorzyklen. Durch hardwaremäßig vorgesehene Umstellung der Reihenfolge in der Befehlsausführung wird dabei versucht, eventuell auftretenden Problemen wie **Daten-** und **Sprungabhängigkeit** (data interlock, branch interlock) zu begegnen: Bei Datenabhängigkeit können zwei aufeinanderfolgende Befehle nicht parallel bearbeitet werden, da die Operanden des zweiten Befehls erst zu Verfügung stehen, wenn der erste Befehl vollständig bearbeitet wurde. Eine Sprungabhängigkeit tritt bei Verzweigungsbefehlen auf, deren Sprungziel in der Regel ein entfernt liegender Befehl ist.



Register	Name des Registers	Funktion
EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP	allgemeine Daten- und Adreßregister	Aufnahme von Operanden für arithmetische und logische Operationen, Berechnung des Offsets (Basis- und Indexregister) zur Adreßrechnung
EIP	Befehlszählerregister	Offset des nächsten auszuführenden Befehls im Codesegment
EFLAGS	Flag-Register	Anzeigen (Status-, System- und Kontroll-Flags)
	Segmentregister:	Adreßrechnung: im Protected Mode: Selektor (Index, d.h. in die Deskriptortabelle); im Real Mode: Anfangsadresse des Segments
CS	Codesegment	
SS	Stacksegment	
DS, ES, FS, GS	Datensegmente	
	Deskriptorregister	Aufnahme der aktuellen Deskriptoren für das Code- und Stacksegment und für die Datensegmente
	Systemadreßregister:	
GDTR		globale Deskriptortabelle
IDTR		Interrupt-Deskriptortabelle
LDTR		lokale Deskriptortabelle
TR		aktuelles Task-Statussegment
CR*n*, $0 \le n \le 3$	System-Kontrollregister	Optionen für das Paging, Adressen der Pagingtabellen
DR*n*, $0 \le n \le 7$	Debug-Register	Adressen von Haltepunkten zur Unterstützung von Testhilfeprogrammen

Abbildung 2.1-2: CPU-Registersatz des INTEL 80486-Prozessors (ohne Register der Gleitpunkt-Verarbeitungseinheit FPU)

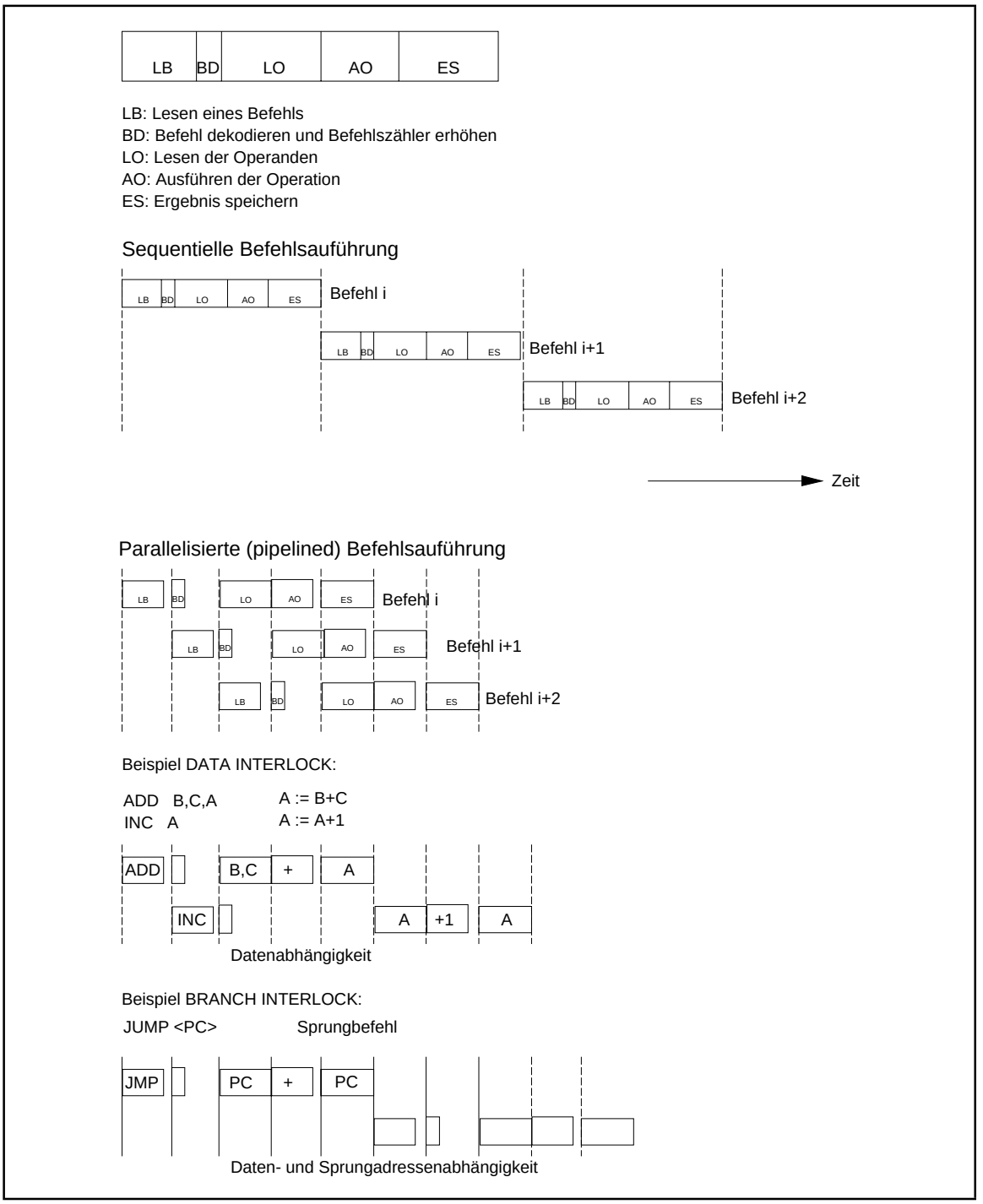


Abbildung 2.1-3: Befehlsausführungsphasen

Exemplarisch für den Ablauf der Befehlsausführung in einer in parallel arbeitende Komponenten aufgeteilten Prozessorarchitektur werden die Vorgänge in der INTEL Pentium-CPU-Architektur (vgl. [MÄR], [MES]) beschrieben. Das Blockdiagramm des Prozessors zeigt Abbildung 2.1-4.

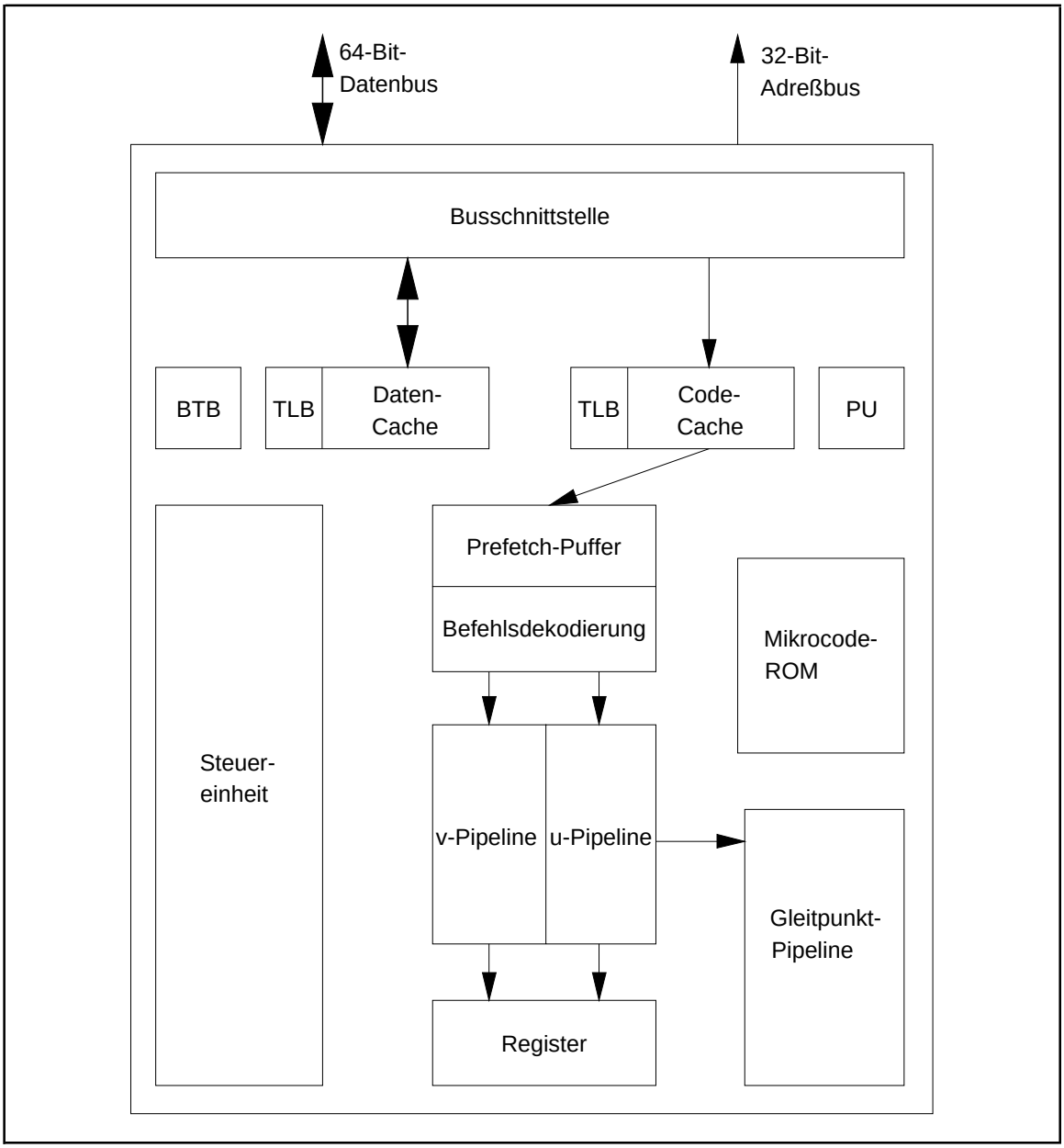


Abbildung 2.1-4: Vereinfachtes Blockdiagramm des INTEL Pentium-Prozessors

Die Busschnittstelle verbindet den Prozessor mit dem 64-Bit-Datenbus und dem 32-Bit-Adreßbus. Die internen Datenpfade haben eine Breite zwischen 128 und 256 Bits, so daß intern eine sehr schnelle Daten- und Codeübertragung stattfindet.

Unmittelbar mit der Busschnittstelle verbunden sind je ein 8 KB großer Cache für Daten und Code. Ein Cache-Eintrag hat die Größe von 32 Bytes. Jedem Cache ist ein eigener **TLB (translation lookaside buffer)** zugeordnet. Dabei handelt es sich um einen sehr schnellen Speicher in Assoziativspeichertechnik, der dazu dient, die Paging Unit (PU) nicht für jeden Speicherzugriff die komplette Umrechnung von virtueller auf physikalische Adresse mit Hilfe des Pagingverfahrens (siehe Kapitel 3.3) durchführen zu lassen: Wenn nämlich aus einer virtuellen Adresse die zugehörige physikalische Adresse bestimmt wird, werden im entsprechenden TLB sowohl der Segment- und Seitennummerteil der virtuellen Adresse als auch die Nummer des zugehörigen physikalischen Seitenrahmens vermerkt.

Dabei werden eventuell vorherige TLB-Einträge überschrieben, weil ja der TLB in seiner Größe beschränkt ist. Liegen Segment- und Seitennummernteil einer virtuellen Adresse und die Nummer des zugehörigen physikalischen Seitenrahmens bereits im TLB (**TLB-Treffer**), erfolgt keine Umrechnung mehr von virtueller Segment- und Seitennummer auf physikalische Seitenrahmennummer; die Werte werden dem TLB entnommen. Also prüft die PU vor jeder Adreßumsetzung zunächst, ob der entsprechende TLB die Informationen für eine virtuelle Adresse enthält. Falls die Überprüfung nicht erfolgreich war (**TLB-Nichttreffer**), wird das übliche Pagingverfahren durchgeführt. Die Caches verwenden physikalische Adressen, der Daten-Cache arbeitet wahlweise mit Write-Through- oder Write-Back-Technik.

Das Herzstück des Pentiums bildet die Steuereinheit, die zwei Befehls-Warteschlangen u und v für Integer-Befehle für ein fünfstufiges Pipelining (u- bzw. v-Pipeline) sowie eine Befehls-Warteschlange für Gleitpunkt-Befehle mit einem achtstufigen Pipelining ansteuert. Integer-Befehle sind hierbei alle Befehle, die keine Gleitpunktoperationen (siehe Kapitel 13) beinhalten, z.B. Integer-Arithmetikbefehle, Vergleichs-, Shift- und Sprungbefehle. Der Prozessor kann dadurch gleichzeitig einen beliebigen Befehl in der u-Pipeline und einen als einfach bezeichneten Befehl in der v-Pipeline ausführen, im Idealfall zwei Integer-Befehle in einem einzigen Takt. Die ersten vier Stufen der Gleitpunkt-Pipeline überlappen mit denen der u-Pipeline, so daß die Integer- und Gleitpunkt-Pipeline nur u.U. parallel arbeiten.

Die beiden Integer-Pipelines werden jeweils durch einen eigenen 32-Byte-Prefetch-Puffer mit Code versorgt. Dabei wird versucht, die Puffer ständig mit Maschinencode, der die nächsten auszuführenden Befehle beinhaltet, gefüllt zu halten. Sobald die Puffer nicht vollständig belegt sind, werden weitere Befehle aus dem Arbeitsspeicher nachgeladen (Befehls-Prefetching). Wenn ein Sprung- oder Unterbrechungsbehandlungsbefehl (siehe unten) ausgeführt werden soll, werden Befehle aus dem Arbeitsspeicher, beginnend an der entsprechenden Zieladresse, bereitgestellt; der Prefetch-Puffer wird vorher gelöscht.

Einen wesentlichen Fortschritt gegenüber dem Vorgängermodell INTEL 80486 bei der Behandlung von Programmverzweigungen bildet die Branch-Prediction-Logik. Sie besteht aus einer Steuereinheit und dem BTB (branch target buffer). Er speichert die Zieladressen der Verzeigungen und zusätzlich Informationen über die Häufigkeit, mit der die jeweilige Verzeigung ausgeführt bzw. nicht ausgeführt wurde. Dadurch kann die Branch-Prediction-Logik Verzeigungen ziemlich zuverlässig vorhersagen und die wahrscheinlichste Zieladresse für das Befehls-Prefetching verwenden. Pipeline-Hemmungen sowie Pipeline-Leerungen sowie das nachfolgende explizite Befehls-Fetching werden deutlich reduziert und die Programmausführung beschleunigt.

Die Adreßumsetzung von einer virtuellen Adresse in eine physikalische Adresse wird von der PU (paging unit) bewerkstelligt.

Da der Pentium-Prozessor auch die z. T. sehr komplexen Befehlssätze seiner Vorgängermodelle unterstützen muß, enthält der Pentium-Prozessor in einer Supporteinheit speziellen Mikrocode zur Ausführung dieser Funktionen. Einfachere Funktionen, z.B. alle Arithmetik- und Shiftoperationen werden entsprechend dem RISC-Prinzip (siehe unten) durch eine festverdrahtete Logik ausgeführt.

Insgesamt setzt der Pentium-Prozessor die CISC-Tradition (siehe unten) seiner Vorgänger INTEL 80x86 fort, verwendet aber zusätzlich viele Techniken von RISC-Rechnern.

Die meisten Hardwarearchitekturen erlauben die Ausführung von Programmen in mindestens zwei unterschiedlichen Betriebsmodi: Im privilegierten **Systemmodus (kernel mode, supervisor mode)** ist die Ausführung aller Maschinenbefehle der Hardware erlaubt, und ein Programm im Systemmodus kann auf alle Systemdaten und Hardwareteile zugreifen und dabei entscheidend und teilweise nicht-rekonstruierbar den aktuellen Zustand der Maschine verändern. Häufig sind Ein- und Ausgabebefehle nur im Systemmodus erlaubt. Je nach Hardwarearchitektur sind gelegentlich weitere Strukturierungen definiert (das Beispiel des Privilegierungskonzepts der INTEL 80x86-Architektur wird weiter unten erläutert). Im nichtprivilegierten **Anwendermodus (Benutzermodus, user mode)** ist die Durchführung eines eingeschränkten Satzes an Instruktionen (nichtprivilegierte Befehle) erlaubt; außerdem besteht nur eine eingeschränkte Zugriffsberechtigung auf Systemdaten und Hardwarekomponenten. Die Unterscheidung von System- und Anwendermodus wird hardwaremäßig durch Setzen entsprechender Anzeigenregister realisiert, und der Wechsel vom Anwender- in den Systemmodus erfordert selbst die Durchführung eines privilegierten Befehls.

Die **Anzeigenregister** enthalten Informationen über die Ausführung einer Operation und den aktuellen Zustand des Prozessors. Beispielsweise wird das Ergebnis des wertmäßigen Vergleichs zweier Operanden durch Setzen einer definierten Bitkombination in einem dieser Register angezeigt. Diese **Anzeigen** können vom Programm ausgewertet werden, und es kann je nach Anzeige verschieden reagieren. Einige typische Anzeigen führen zu einem Programmabbruch durch den Prozessor. Sie beschreiben dann einen "irregulären" Zustand des Prozessors, der aufgrund der letzten Befehlsausführung entstand. Bei einigen dieser Situationen ist es möglich, den automatischen Programmabbruch zu verhindern, indem diese Absicht in weiteren Anzeigenregistern per Programm durch sogenannte **Masken** (das sind spezielle fest definierte Zeichenkombinationen) festgehalten wird. Eine Maske gibt an, auf welche Anzeige sofort, erst später oder gar nicht reagiert werden soll.

Zusammenfassend geben die Anzeigen Auskunft über das Ergebnis des wertmäßigen Vergleichs von Operanden, das Ergebnis von arithmetischen und logischen Operationen (z.B. ob das Resultat einer Addition kleiner, größer oder gleich Null ist), arithmetische Zahlenbereichsüberläufe als Ergebnis einer Operation, Datenformatfehler bei arithmetischen Operationen, Divisionsfehler (z.B. Division durch Null) während einer Operation, den Versuch, eine dem Rechner nicht bekannte (nicht-entschlüsselbare) oder dem Anwender zur Zeit nicht-erlaubte (privilegierte) Operation auszuführen, ungültige, aber trotzdem verwendete Operandenadressen usw.

Mit Hilfe der **Spezialregister**, über die die meisten Prozessorarchitekturen verfügen, können Arithmetikunterstützung, verschiedene Zeitgeber, Testhilfeoptionen, Unterstützung für die Mehrfachbenutzung der CPU, Speicherschutzmechanismen, Eingabe- und Ausgabeoperationen usw. realisiert sein.

Viele Prozessorarchitekturen verfügen über hardwaremäßig implementierte oder durch die Hardware unterstützte **Schutzmechanismen** des Arbeitsspeichers vor fehlerhaftem Zugriff, insbesondere Schutz des Betriebssystems und Schutz vor Zugriffe auf Programm- und Datenteile anderer fremder Anwender. Diese Schutzmechanismen sind häufig in entsprechende **Privilegierungskonzepte** eingebunden.

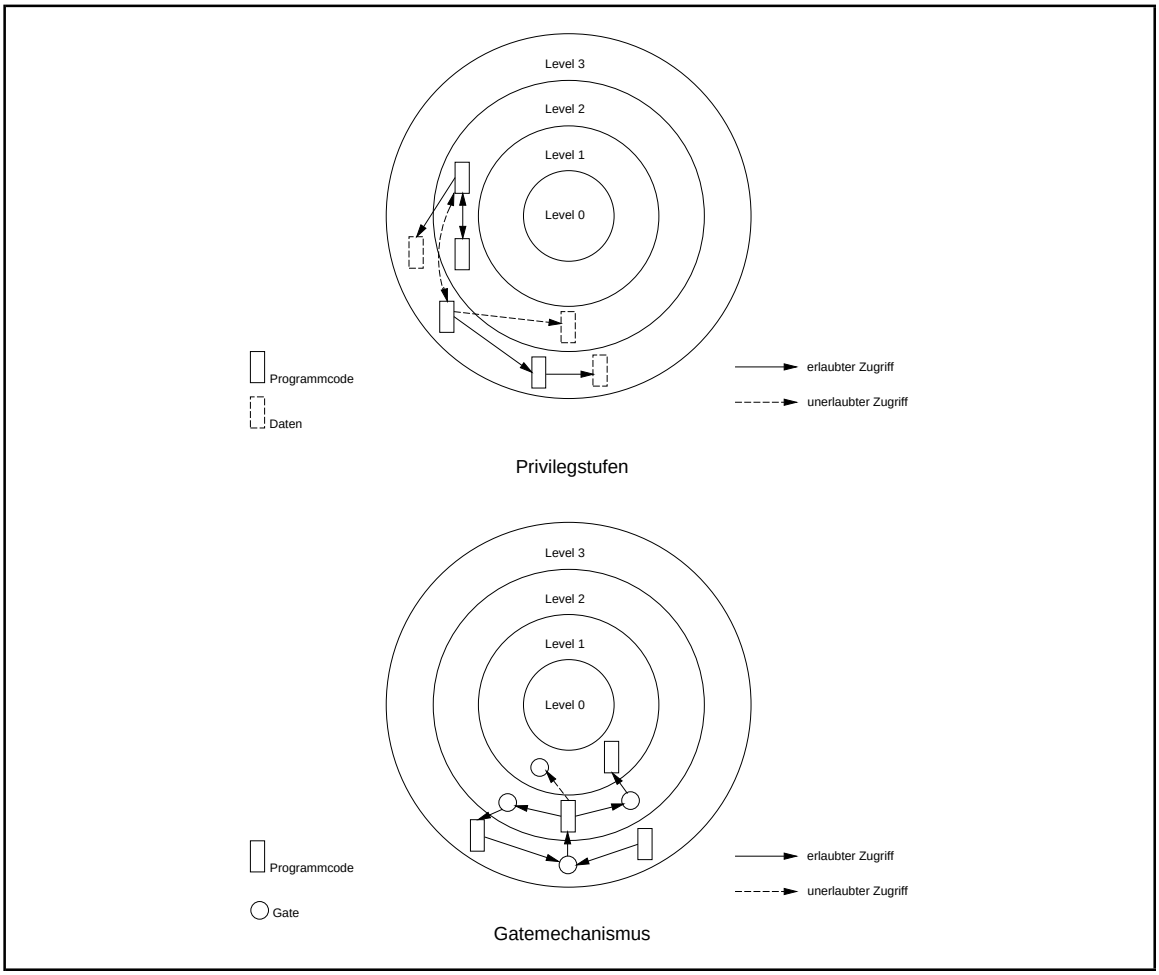


Abbildung 2.1-4: Privilegierungskonzept INTEL 80386/80486

Ein Beispiel ist das hierarchische Privilegierungskonzept der INTEL 80386/80486-Architektur (vgl. [NEL]):

Jeder Programmteil (Codesegment) und jeder Datenteil (Datensegment) gehört einem von vier Privilegstufen an. Die Privilegstufe mit den meisten Privilegien ist die Stufe 0; die Stufe 3 verfügt über die wenigsten Privilegien. Der Zugriff auf Datensegmente kann nur von derselben oder von einer höheren Privilegstufe, d.h. mit einer kleineren Nummer, verglichen mit der aktuellen Privilegstufe erfolgen. Auf Codesegmente kann nur innerhalb derselben Stufe verzweigt werden. Außerdem gibt es eine Reihe von Maschineninstruktionen, deren Ausführung nur auf Privilegstufe 0 erlaubt ist. Insbesondere Betriebssysteme setzen dieses Priveligierungskonzept ein, wobei sie nicht immer alle Privilegstufen nutzen. UNIX oder WINDOWS 95 verwenden beispielsweise nur die Stufen 0 und 3; OS/2 verwendet drei Stufen: Der OS/2-Code läuft auf Stufe 0, Anwendungsprogramme auf Stufe 3 und spezielle Routinen mit Zugriff auf die Eingabe- und Ausgabeeinheiten auf Stufe 2.

Der Übergang von einer Privilegstufe auf eine höher privilegierte Stufe erfolgt über Gates. Dabei handelt es sich um definierte Zugangspunkte, deren Beschreibungen in Systemtabellen gehalten werden. Der Gebrauch eines Gates ermöglicht ein vom Prozessor kontrolliertes Umschalten der Privilegstufe, wodurch z.B. verhindert wird, daß ein weniger privilegiertes Programm auf höher privilegierte Systemprozeduren und Systemdaten direkt zugreift.

Der **Anschluß der Peripherie** (Eingabeeinheit, Ausgabeeinheit, externe Massenspeicher) erfolgt meist über genormte Schnittstellen. Bei der Großrechnerarchitektur wird dieser Anschluß über sogenannte **Kanäle** oder **E/A-Systeme** (mit verschiedenen Charakteristika bzgl. der Operationsgeschwindigkeit und der Anzahl gleichzeitig übertragener Daten), die selbständige Rechner darstellen und simultan (parallel) zur Zentraleinheit operieren. Bei der Busarchitektur (Abbildung 2-2) werden die peripheren Geräte z.T. mit Hilfe von speziellen Einsteckkarten ***direkt an den internen Bus*** angeschlossen.

Gelegentlich erfolgen auch Datentransfers direkt zwischen Arbeitspeicher und Peripherie **(DMA, direct memory access)**.

Die gegenseitige physikalische Anpassung eines Geräts an die standardisierte Schnittstelle (des Busses) übernehmen gerätespezifische **Controller**. Das DV-System bedient eine Gerät, indem der entsprechende Controller programmiert wird; er kann als eigenständiges Teilsystem mit einem eigenen Registersatz, lokalem Speicher zur Pufferung von Daten und der Möglichkeit zur Erzeugung von Interrupts (siehe Kapitel 2.2) angesehen werden.

Die Ansteuerung eines Geräts erfolgt vom Betriebssystem her durch ein geräte(typ)spezifisches Programm, den **(Geräte-) Treiber**. Betriebssysteme besitzen **fest eingebaute Treiber** entsprechend den standardmäßig angeschlossenen Geräten oder **installierbare Treiber**, die spezielle Geräte ansteuern und bei Systeminitialisierung meist resident in den Arbeitsspeicher geladen werden. Weitere Details zum Ein/Ausgabekonzept beschreibt Kapitel 2.3.

Im Laufe der Zeit wurden Rechnersysteme entwickelt, deren CPU eine Vielzahl unterschiedlicher Maschineninstruktionen ausführen kann, die z.T. sehr spezielle Aufgaben lösen (z.B. Umladen mehrerer Register mit einer Maschineninstruktion für Multiprozessing-Systeme, umfangreiche Stringoperationen) bzw. ein komplexes Befehlsformat aufweisen. Dadurch wird die Steuerung in der CPU aufwendig und kostenintensiv. Die so entstandene Rechnerarchitektur wird als **CISC-Architektur (CISC, complex instruction set computer)** bezeichnet. In einem typischen CISC-Rechner sind 150 bis 250 verschiedene Maschineninstruktionen definiert. Beispiele sind die INTEL 80x86- und Motorola 680x0-Rechner oder die Rechnerfamilien der IBM- und SNI-Großrechner.

Die Beobachtung, daß 80% aller Programme nur 20% aller verfügbaren Maschineninstruktionen einsetzen[4], führte zu der Überlegung, eine Rechnerarchitektur zu entwerfen, die nur sehr einfache, dafür aber sehr schnell ausführbare Maschinenbefehle verarbeitet. Komplexere Operationen werden durch mehrere Maschinenbefehle nachgebildet. Die so entworfene Architektur wird mit **RISC-Architektur (RISC, reduced instruction set computer)** bezeichnet. Ein RISC-Rechner enthält meist weniger als 100 verschiedene Maschineninstruktionen. Beispiele hier sind IBM RS6000-Prozessoren, PowerPC, DEC

[4] Folkloristische Regel.

Alpha-Chip INTEL Pentium (der aus Kompatibilitätsgründen zu seinen Vorgängern noch zu den CISC-Rechnern zählt, jedoch viele charakteristische Eigenschaften der RISC-Architektur aufweist) u.a. Weitere typische Eigenschaften einer RISC-Architektur sind:

- Es gibt nur wenige und stark vereinfachte Befehlsausführungsphasen, die parallel ausgeführt werden
- Es gibt nur sehr einfache und wenige Typen von Maschinenbefehlen, deren Bearbeitung meist in einem Prozessorzyklus erfolgt und die nicht mikroprogrammiert, sondern "fest verdrahtet" sind
- Es erfolgt eine einfache Adressierung der Operanden in den Maschinenbefehlen: Arbeitsspeicherzugriffe finden nur mittels LOAD- und STORE-Operationen statt, alle anderen Operationen werden nur auf Registeroperanden durchgeführt
- Ein einfaches einheitliches Anweisungsformat sorgt dafür, daß die Befehlsdekodierung sehr schnell (Registerzugriff und Befehlsdekodierung simultan) erfolgen kann
- Es wird eine Konfliktvermeidungstrategie zur schnellen Ausführung von Sprungbefehlen verfolgt
- Die Hardwarekomplexität wird durch Arbeitsverlagerung in Compiler und Delegation von Steuerungsaufgaben an optimierende Compiler verringert
- Ein RISC-Rechner enthält sehr viele Register.

2.2 Eine Hardware/Software-Schnittstelle: Der Interruptmechanismus

Eine der wichtigsten Hardware/Software-Schnittstellen ist der **Interruptmechanismus** eines Rechners. Man unterscheidet Rechner-abhängig Hardware- und Software-Interrupts unterschiedlicher Typen; eine grobe Klassifizierung zeigt Abbildung 2.2-1.

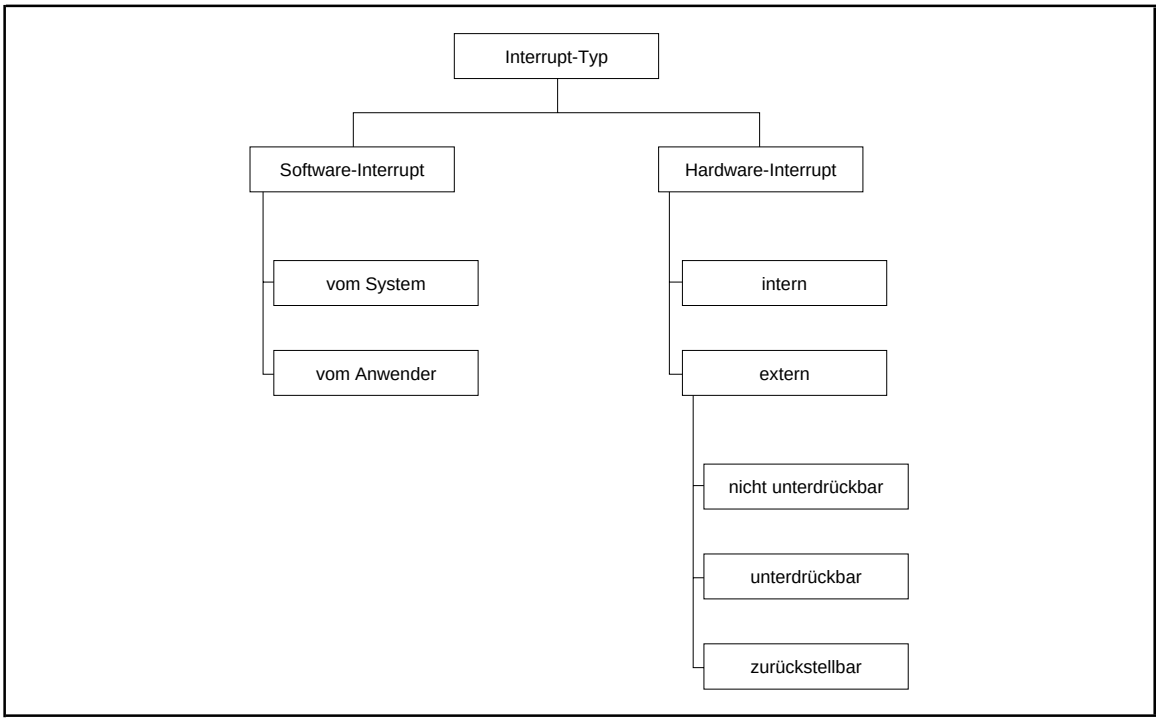


Abbildung 2.2-1: Interrupttypen eines Rechners

Ein **Interrupt** ist ein definiertes Signal, das die gerade laufende Aktion der CPU unterbricht und damit Gelegenheit gibt, auf dieses Signal sofort zu reagieren. Der Zeitpunkt, an dem ein Interrupt auftritt, ist i.a. nicht vorhersehbar, d.h. ein Interrupt stellt ein **asynchrones Ereignis** dar. **Externe Hardware-Interrupts** kommen von angeschlossenen Geräten der Peripherie (z.B. Steuereinheiten der Tastatur oder des Diskettenlaufwerks beim PC), **interne Hardware-Interrupts** kommen aus der CPU (z.B. Zeitgeber, arithmetische Fehlersituationen, Versuch der Ausführung eines privilegierten Befehls im Anwendermodus). Durch geeignete Hardware-Mechanismen (Masken) sind einige Interrupts **unterdrückbar**, d.h. sie werden überhaupt nicht behandelt, oder **zurückstellbar**, d.h. der Interrupt wird erst später behandelt. Auch Anwendungs- und Systemprogramme können sich des Interruptmechanismus des Rechners bedienen, sofern der Maschineninstruktionssatz des Rechners einen entsprechenden Befehl zur Generierung eines **Software-Interrupts** bereitstellt.

Jeder Interrupt ist mit einer fest definierten **Interruptnummer** versehen. Die Zusammenstellung in Abbildung 2.2-2 zeigt exemplarisch die im sogenannten "Real Mode" definierten 256 Interrupts der INTEL 80x86-Prozessortypen.

Int.nummer	Belegung	Int.nummer	Belegung
00	CPU: Division durch Null	25	DOS: Diskette/Festplatte lesen
01	CPU: Einzelschritt	26	DOS: Diskette/Festplatte schreiben
02	CPU: NMI (Fehler im Arbeitsspeicher-Baustein)	27	DOS: Programm beenden, resident bleiben
03	CPU: Breakpoint erreicht	28	DOS: Idle-Interrupt
04	CPU: numerischer Überlauf	29 bis 2D	(reserv. für nicht dokumentierte DOS-Funkt.)
05	Hardcopy	2E	Aufruf von COMMAND.COM
06	unbekannter Befehl	2F	Multiplexer-Interrupt
07	(reserviert)	30	(reserv. für nicht dokumentierte DOS-Funkt.)
08	H/W-Interrupt IRQ0: Timer (18,2 mal/sec)	31	DPMI-Funktionsaufrufe (DPMI-Server)
09	H/W-Interrupt IRQ1: Tastatur	32	(reserv. für nicht dokumentierte DOS-Funkt.)
0A	H/W-Interrupt IRQ2: zweiter 8259-Baustein	33	Maustreiber
0B	H/W-Interrupt IRQ3: serielle Schnittstelle COM2	34 bis 3E	Borland-Floatingpoint-Emulator
0C	H/W-Interrupt IRQ4: serielle Schnittstelle COM1	3F	Overlay-Manager
0D	H/W-Interrupt IRQ5: Festplatte	40	Floppy-Server
0E	H/W-Interrupt IRQ6: Diskette	41 bis 42	Adresse der Festplattenparametertabelle
0F	H/W-Interrupt IRQ7: parall. Schnittstelle LPT1:	43	EGA/VGA-Graphikzeichensatz
10	BIOS: Videofunktionen	44	EGA-Interrupt
11	BIOS: Konfiguration ermitteln	45 bis 49	(können beliebig belegt werden)
12	BIOS: Arbeitsspeichergröße ermitteln	4A und 50	BIOS: Alarmzeit erreicht
13	BIOS: Zugriff auf Diskette/Festplatte	4B bis 66	(können beliebig belegt werden)
14	BIOS: Zugriff auf serielle Schnittstelle	67	EMS-Manager
15	BIOS: Kassettenzugriff/erweiterte Funktionen	68 bis 6B	(können beliebig belegt werden)
16	BIOS: Tastaturabfrage	6C	Real-Time-Clock
17	BIOS: Zugriff auf parallele Druckerschnittstelle	6D	VGA-Interrupt
18	Aufruf des ROM-BASIC	6E bis 6F	(können beliebig belegt werden)
19	BIOS: System booten (<Alt>+<Ctrl>+<Del>)	70	H/W-Interrupt IRQ08: Echtzeituhr
1A	BIOS: Real-Time-Clock: Zeit/Datum abfragen	71	H/W-Interrupt IRQ09
1B	<Crtl>-<Break>-Taste gedrückt	72	H/W-Interrupt IRQ10
1C	wird nach jedem INT 08 aufgerufen	73	H/W-Interrupt IRQ11
1D	BIOS: Adresse der Video-Parameter-Tabelle	74	H/W-Interrupt IRQ12
1E	Adresse der Disketten-Parameter-Tabelle	75	H/W-Interrupt IRQ13: math. Coprozessor 80287
1F	Adr. d. Graphikzeichensatzes (ASCII 128..255)	76	H/W-Interrupt IRQ14: Festplatte
20	DOS: Programm beenden	77	H/W-Interrupt IRQ15
21	DOS: DOS-Funktion aufrufen	78 bis 7F	(können beliebig belegt werden)
22	Adresse der DOS-Programm-Ende-Funktion	80 bis F0	(werden innerhalb von GWBASIC genutzt)
23	Adresse der DOS <Ctrl>-C-Funktion	F1 bis FF	(stehen zur freien Verfügung)
24	Adresse der DOS-Fehlerfunktion		

Abbildung 2.2-2: Standardmäßige Interruptbelegung (Interruptnummern sedezimal) beim INTEL 80x86-Prozessor im "Real Mode"

Die **Interruptbehandlung** läuft nach folgendem Prinzip ab (Abbildung 2.2-3): Über die Interruptnummer wird vom Rechner in einer im Arbeitsspeicher an fester Stelle installierten Tabelle (**Interruptvektortabelle**) die Anfangsadresse einer speziellen **Interruptbehandlungsroutine** gefunden, in die wie bei einem Unterprogrammsprung verzweigt wird. Die Interruptbehandlungsroutinen sind als Teil des Betriebssystems festgelegt oder werden vom Benutzer bereitgestellt (im laufenden Betrieb oder bei Systemstart). Sie gehören nicht zum normalen Programmablauf. Nach Behandlung des Interrupts wird das unterbrochene Programm meist an der Unterbrechungsstelle fortgesetzt. Wenn während der Interruptbehandlung weitere Interrupts eintreffen, werden diese ebenfalls sofort behandelt, wenn sie nicht als zurückgestellte oder als abzuweisende Interrupts klassifiziert sind, so daß i.a. eine ineinandergeschachtelte Behandlungsreihenfolge entsteht.

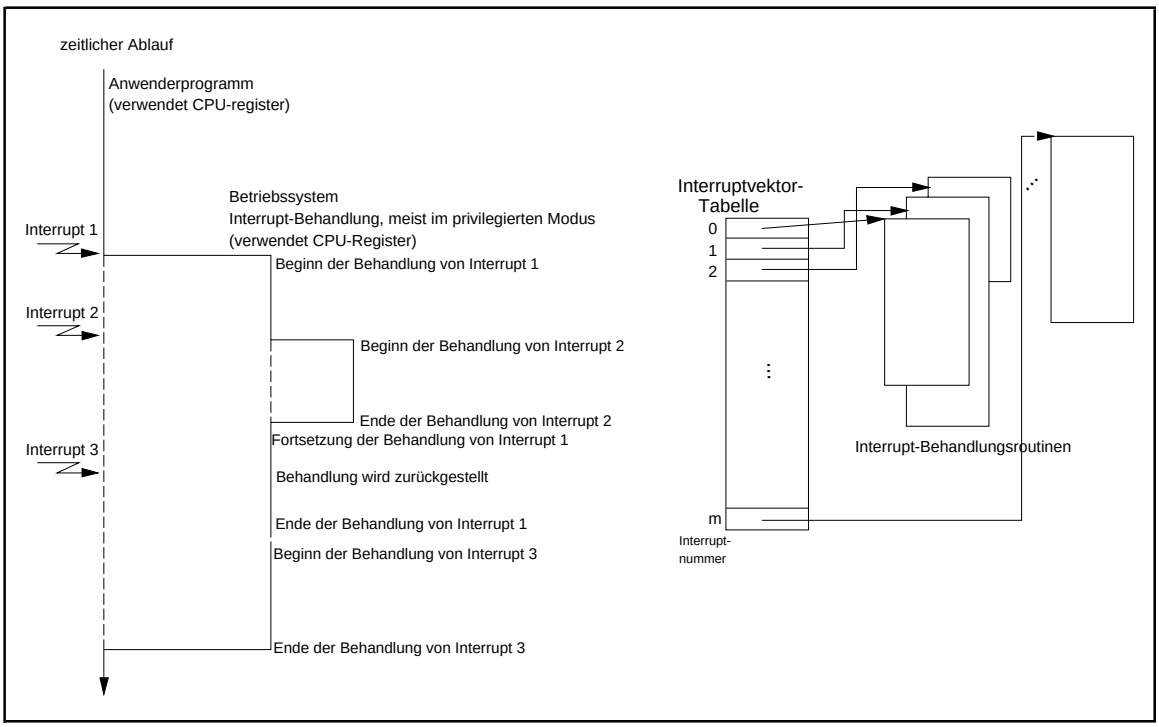


Abbildung 2.2-3: Interrupt-Behandlung

Die Erzeugung eines Software-Interrupts aus einem PASCAL-Programm heraus erfolgt durch Aufruf der (System-) Routine `Intr` mit dem Aufrufformat

```
PROCEDURE Intr (IntNo : BYTE; VAR Regs : Registers);
```

Der Parameter `IntNo` gibt die entsprechende Interruptnummer an (Abbildung 2.2-2); der (strukturierte) Parameter `Regs` übergibt Registerinhalte an die `Intr`-Prozedur bzw. gibt in der `Intr`-Prozedur gesetzte Registerinhalte zurück.

Zur Definition einer eigenen Interruptbehandlung, etwa zum Interrupt `$1B` (<Crtl>-<Break>-Taste), bedient man sich der (System-) Routinen `GetIntVec` und `SetIntVec` gemäß folgendem Prinzip (Abbildung 2.2-4): Zunächst muß eine eigene Interruptbehandlungsroutine, etwa mit dem Bezeichner `Int1BHandler`, definiert werden,

die durch das Schlüsselwort `INTERRUPT` als Interruptbehandlungsroutine gekennzeichnet wird (der Compiler erzeugt für derartige Routinen speziellen Start- und Beendigungscode, z.B. werden Registerinhalte gesichert). Der Interruptvektor der bisherigen Routine zur Behandlung des Interrupts `$1B` wird ermittelt und lokal gesichert (mit `GetIntVec`) und die Adresse (der Interruptvektor) der neuen Interruptbehandlungsroutine in die Interruptvektortabelle eingebaut (mit `SetIntVec`). Bei jedem Interrupt mit Nummer `$1B` läuft nun die neue Interruptbehandlungsroutine `Int1BHandler` ab. In folgendem Beispiel wird später die bisherige Interruptbehandlungsroutine wieder in Kraft gesetzt.

```
PROGRAM eigener_interrupt;

VAR int1Bsave_ptr : Pointer;   { zu Sicherstellung des bisherigen
                                 Interruptvektors zum Interrupt $1B }

...

PROCEDURE Int1BHandler (Flags, CS, IP, AX, BX, CX, DX,
                        SI, DI, DS, ES, BP            : WORD);
                       INTERRUPT;
BEGIN { Int1BHandler }
  ...
END   { Int1BHandler };

...

BEGIN { eigener_interrupt }
  ...
                         { bisherigen Interruptvektor sichern: }
  GetIntVec ($1B, int1Bsave_ptr);
                         { neuen Interruptvektor einsetzen:    }
  SetIntVec ($1B, @Int1BHandler);
  ...
                         { bisherige Interruptbehandlungs-
                           routine reaktivieren:               }
  SetIntVec ($1B, int1Bsave_ptr);
  ...
END   { eigener_interrupt }.
```

Abbildung 2.2-4: Definition einer eigenen Interruptbehandlung

2.3 Aspekte der Gerätesteuerung und des Ein/Ausgabekonzepts

Eine weitere Hardware/Software-Schnittstelle stellt der Datenverkehr zwischen der "Umwelt" eines Rechners und den Programmen im Arbeitsspeicher dar. Die dabei stattfindenden sehr geräteabhängigen Aktionen bedürfen der intensiven Steuerung durch das Betriebssystem des Rechners. In höheren Programmiersprachen sind aufgrund dieser Maschinen- und Betriebssystemabhängigkeit der Vorgänge häufig nur wenige allgemeine Ein/Ausgabefunktionen vorgesehen. In diesem Kapitel werden einige hardwarenahe Aspekte der Gerätesteuerung und des Ein/Ausgabekonzepts (vgl. [TAN]) behandelt.

Ein Gerät zusammen mit dem zugehörigen Treiber kann als eigenständiger, in das übrige DV-System ***eingebetteter Rechner*** angesehen werden, der über einen ***speziellen Registersatz und Puffer für den Datentransfer*** zur Entkoppelung unterschiedlicher Operationsgeschwindigkeiten von Gerät, Bus und CPU und zur Überbrückung von Wartezeiten, falls der

Bus gerade mit anderen Datentransferaktionen belegt ist. Außerdem werden gerätespezifische Fehlersituationen behandelt. Der **typische Instruktionssatz** umfaßt Kommandos zur Steuerung des Geräts, zur Ermittlung von Informationen über den gegenwärtigen Gerätezustand und die eigentlichen Datentransferinstruktionen.

Meist besitzt ein DV-System unterschiedliche **Typen von Geräten und Gerätetreibern**:

- **blockorientierte Geräte**: Informationen werden in Blöcken fester Größe übertragen, die das Gerät jeweils über eine einzige Adresse identifiziert. Typische Blockgrößen liegen im Kilobytebereich. Beispiele für blockorientierte Geräte sind Disketten- oder Festplattenlaufwerke

- **zeichenorientierte Geräte:** Es werden Zeichenströme übertragen, die keine weitere Strukturierung aufweisen. Einzelne Zeichen sind nicht adressierbar, sondern werden sequentiell bearbeitet. Beispiele für zeichenorientierte Geräte sind Terminals, Drucker, Maus, Kommunikationsschnittstellen (Netzwerkschnittstellen).

Einige Geräte wie Timer, die lediglich Interrupts erzeugen und keine Daten übertragen, oder Bildschirme, die Pufferinhalte im Arbeitsspeicher direkt interpretieren, lassen sich nicht eindeutig in dieses Schema einordnen. Trotzdem ist dieses Klassifizierungsschema allgemein genug, um damit geräteunabhängige Ein/Ausgabeschnittstellen zu definieren ([TAN]).

Exemplarisch wird der Ablauf bei der Datenübertragung zwischen Peripherie (Datei) und Arbeitsspeicher (Datenbereich eines Programms) in Abbildung 2.3-1 gezeigt. Eine Anwendung möchte Daten aus einem ihrer Datenbereiche in eine Datei, die durch einen Namen identifiziert wird, übertragen und gibt einen entsprechenden Auftrag an das Betriebssystem. Die Dateiverwaltung bestimmt über den Dateinamen die physikalische Position, an der das Ziel der Übertragung liegt und gibt einen entsprechenden Auftrag an das Ein/Ausgabemanagement des Betriebssystems. Dieses generiert hieraus einen Auftrag an den Controller, der für das Speichermedium zuständig ist, die spezifizierte Datenmenge vom Arbeitsspeicher an das Gerät zu transferieren. In diesem Modell werden **Aufträge** über verschiedene Stufen im System weitergereicht und modifiziert. Wesentlich dabei ist, daß eine zeitliche Entkoppelung der einzelnen Aufträge erfolgen kann, d.h. an einer Stufe, z.B. einem Treiber bzw. zugehörigen Gerätecontroller, ankommende Aufträge können dort in eine Auftragswarteschlangen eingereiht werden, falls das Gerät gerade noch mit der Abwicklung eines anderen Auftrags beschäftigt ist. Die Synchronisation zwischen den einzelnen Stufen erfolgt über Interrupts.

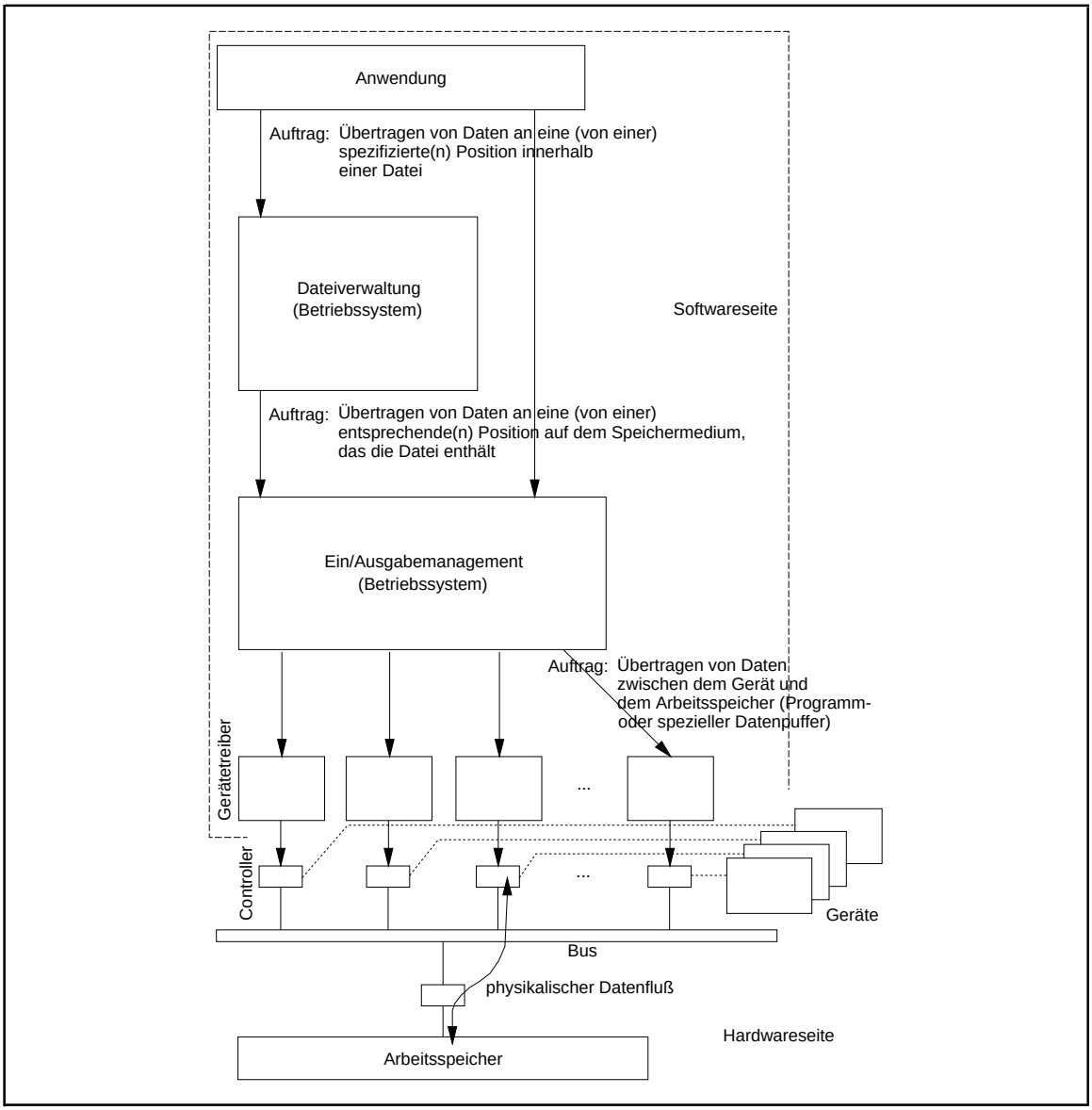


Abbildung 2.3-1: Prinzip der Datenübertragung im Rechner

Man unterscheidet zwei Arbeitsweisen eines Ein/Ausgabegeräts:

- Bei der **synchronen Arbeitsweise** wird ein Auftrag über das Ein/Ausgabemanagement an ein Gerät gegeben, und die Anwendung wartet solange, bis das Ein/Ausgabemanagement die Erledigung des Auftrags beispielsweise durch die Übergabe eines Returncodes anzeigt. Die synchrone Arbeitsweise entspricht einem normalen Unterprogrammaufruf, dessen Ziel in einem anderen Teilsystem, hier dem Ein/Ausgabemanagement, liegt.

- Bei der **asynchronen Arbeitsweise** wird ein Auftrag an das Ein/Ausgabemanagement übergeben. Dieses signalisiert dem Auftraggeber die Entgegennahme des Auftrags, wenn dieser vom Treiber entgegengenommen wurde, und der Auftraggeber kann seinen Ablauf mit anderen Aktionen fortsetzen. Wenn der Auftrag an das Ein/Ausgabemanagement

abgearbeitet ist, wird dem Auftraggeber dieses Ereignis - evtl. unter Übergabe eines Returncodes - mitgeteilt. Auf diese Weise können Wartezeiten, die durch eine langsame Operationsdauer eines Geräts entstünden, für weitere Tätigkeiten genutzt werden. Die Implementierung einer asynchronen Ein/Ausgabe erfordert i.a. Synchronisationsaktivitäten, insbesondere dann, wenn nacheinander mehrere Aufträge bezüglich desselben Geräts an das Ein/Ausgabemanagement übergeben werden oder die Resultate der einzelnen Aufträge voneinander abhängig sind.

Die im folgenden angegebenen Attribute Adressierungsweise, Gerätesteuerung und Datentransfer beschreiben neben der Klassifizierung nach Gerätetyp (block- bzw. zeichenorientiert) die **charakteristischen Eigenschaften eines Geräts** und sind beliebig kombinierbar (ausgelassen wird dabei die Fehlerkontrolle, die entweder Fehler durch entsprechende Protokolle behebt oder zumindest Fehler-Returncodes dem Initiator eines Auftrags übermittelt):

- **Adressierungsweise**: Sie beschreibt, wie das Gerät durch die CPU adressiert wird, und hängt vom Gerätetyp und von der Art des Anschlusses des Controllers am Bus ab. Im PC-Bereich findet man üblicherweise speicherabgebildete (memory-mapped) und I/O-abgebildete (I/O mapped) Geräte:

  -- bei **speicherabgebildeten (memory-mapped) Geräten** sind die Register und/oder Puffer des Controllers in den Arbeitsspeicheradreßraum des Prozessors abgebildet, so daß ein Zugriff auf diese Arbeitsspeicheradressen einem Zugriff auf die Register bzw. Puffer des Controllers entspricht. Dieser Zugriff erfolgt mit den üblichen Befehlen zur Datenübertragung zwischen Arbeitsspeicheradressen. Beispielsweise liegt der Bildschirmpuffer in der INTEL 80x86-Architektur (Real Mode, Textmodus) auf den festdefinierten Adressen $A0000_{16}$ bis $BFFFF_{16}$; jedem Punkt des Bildschirms sind bestimmte mehrere Bytes fest zugeordnet, die Inhalt und Darstellungsform der Information auf dem Bildschirm festlegen. Eine Änderung an den jeweiligen Adressen wirkt sich sofort auf den Inhalt des zugeordneten Punktes auf dem Bildschirm aus.

  -- **I/O-abgebildete (I/O mapped) Geräte** werden Teile des Arbeitsspeichers als Ein/Ausgabe-Zugangspunkte (**Ports**) definiert. Ein Port ist fest oder generierbar mit einem Gerät verbunden, und **spezielle Ein/Ausgabebefehlsfolgen** transferieren Daten zwischen den CPU-Registern und diesen Zugangspunkte, d.h. zwischen dem Rechner und dem am Port angeschlossenen Gerät. Abbildung 2.3-4 zeigt beispielsweise die Belegung der Portadressen im INTEL 80x86.

Portadresse	Nutzung	Portadresse	Nutzung
0000..000F	1. DMA-Controller	02E0..02EF	GPIB-Board
0010..001F	2. DMA-Controller	02F8..02FF	COM2
0020..003F	1. Interruptcontroller	0300..031F	Prototypkarte
0040..005F	Timer	0320..032F	Harddisk-Controller
0060..006F	Tastatur	0360..036F	Netzwerkkarten
0070..007F	CMOS-Uhrenbaustein	0378..037F	LPT2
0080..009F	1. DMA-Controller	0380..038F	2. Bisynchron-Adapter
00A0..00BF	2. Interruptcontroller	03A0..03AF	1. Bisynchron-Adapter
00C0..00DF	2. DMA-Controller	03B0..03BB	Monochromer Videoadapter
00F0..00FF	Coprozessor	03BC..03BF	LPT1
0100..01EF	?	03C0..03CF	EGA/VGA-Karten
01F0..01F8	Harddisk-Controller	03D0..03DF	Farbbildschirm-Adapter
0200..020F	Gameport	03F0..03F7	Disketten-Controller
0278..027F	LPT3	03F8..03FF	COM1
02B0..02DF	2. EGA-Adapter		

Abbildung 2.3-4: Portadressen im INTEL 80x86

- **Gerätesteuerung**: Es wird bestimmt, wie der Prozessor den gegenwärtigen Status des Geräts ermittelt und eine Aktion des Geräts anstößt und steuert. Typische Gerätesteuerungen sind:

  -- **Polling**: Das System fragt periodisch an, ob das Gerät bedient werden möchte, z.B. ob ein Datentransfer in den Arbeitsspeicher aktuell ansteht

  -- **interruptgesteuerte Verarbeitung**: durch einen Interrupt wird (asynchron) dem Prozessor mitgeteilt, daß eine Aktion des Geräts ansteht, die dann bedient wird.

- **Datentransfer**: Strategien, wie Daten physikalisch zwischen Gerät und Arbeitsspeicher übertragen werden, beinhalten:

  -- **programmierte Ein/Ausgabe**: der Datentransfer zwischen Arbeitsspeicher und Gerät erfolgt gesteuert durch den Prozessor, indem er eine Folge von Ein/Ausgabeinstruktionen (z.B. eine wiederholte Durchführung von Ein/Ausgabebefehlen bezüglich eines Ports) ausführt. Während der Abarbeitung dieses Ein/Ausgabeprogramms ist der Prozessor für andere Aktionen nicht verfügbar.

  -- **direkter Speicherzugriff (DMA, direct memory access)** ist eine Technik, in der ein spezieller Controller (DMA-Controller) programmiert wird, um Daten zwischen Gerät und Arbeitsspeicher zu übertragen, ohne den Prozessor des Rechners in Anspruch zu nehmen. Der DMA-Controller wird angestoßen, indem ihm die Arbeitsspeicheradresse, an der die zu transferierenden Daten liegen bzw. an die sie geschrieben werden sollen, und die Menge zu übertragender Daten mitgeteilt werden. Während der DMA-Controller Daten transferiert, kann der Prozessor andere Aktivitäten ausführen. Das Ende der Datenübertragung teilt der DMA-Controller durch einen Interrupt mit.

Ein Problem könnte sich hierbei ergeben, daß sowohl der Datenstrom durch den DMA-Controller als auch der Instruktionsstrom für den Prozessor über den internen Bus laufen und dabei Konkurrenzsituationen bezüglich der Busbenutzung entstehen. Da jedoch im Prozessor üblicherweise bereits mehrere Instruktionen im Vorgriff in einer Instruktionswarteschlange stehen und auf Abarbeitung warten, ist der Prozessor auch dann mit Instruktionen "versorgt", wenn der DMA-Controller gerade Daten über den Bus überträgt[5].

Das Prinzip der Ein/Ausgabe in Großrechnern über autonome Ein/Ausgabekanäle ordnet sich ebenfalls hier ein: Es wird im Arbeitsspeicher eine Folge von Ein-/Ausgabeinstruktionen (**Kanalprogramm**) bereitgestellt, deren Startadresse dem Kanal mitgeteilt wird, das dann vom Kanal zeitlich (echt) parallel zum Ablauf des Prozessors abgearbeitet wird.

[5] Man sagt, der DMA-Controller "stiehlt" Buszyklen vom Prozessor.

3 Aspekte der Rechnerprogrammierung

Die **Maschinensprache** eines Rechners besteht aus den binär kodierten Maschineninstruktionen, die von der Steuerung des Prozessors direkt interpretiert werden können. Folglich ist die Maschinensprache rechnerspezifisch. Programmierung in Maschinensprache war höchstens in den Anfangstagen der Rechnerentwicklung üblich.

Unter dem Begriff **Assembler-Sprache** versteht man eine mnemotechnische, symbolische Schreibweise der Maschinensprache eines Rechners. Diese Sprache ist daher wie die Maschinensprache hochgradig rechnerabhängig. Ein in der Assembler-Sprache eines Rechners formuliertes Programm muß vor seinem Ablauf in Maschinensprache übersetzt werden; der dazu notwendige Sprachübersetzer wird wie die Sprache selbst mit **Assembler** bezeichnet. Hierbei entsteht aus einem Programm in der Assembler-Sprache ein Objektmodul. Die Übersetzung erfolgt bei den meisten Assembleranweisungen 1:1, d.h. der Assembler generiert aus einer Anweisung in der Assembler-Sprache (höchstens) eine Maschineninstruktion; häufig dienen einige Anweisungen in einem Assembler-Programm der Übersetzungssteuerung.

Ausschnitt aus einem ausführbaren Programm in Maschinensprache (die 1. Spalte zeigt sedezimal in vereinfachter Form den Abstand zum Segmentanfang, d.h. die Adresse innerhalb des Moduls)	(Programmstart an Adresse 0063) 0060 : 9A 00 00 DC 0F 9A 0D 00 7A 0F 55 89 E5 0070 : 31 C0 9A CD 02 DC 0F 9A CC 01 7A 0F BF 70 01 1E 0080 : 57 BF 00 00 0E 57 31 C0 50 9A 70 06 DC 0F B8 0C 0090 : 00 31 D2 52 50 31 C0 50 9A F8 06 DC 0F BF 1D 00 00A0 : 0E 57 31 C0 50 9A 70 06 DC 0F 9A FE 05 DC 0F 9A 00B0 : 91 02 DC 0F BF 70 00 1E 57 9A 9C 06 DC 0F A3 52 00C0 : 00 9A 9D 05 DC 0F 9A 91 02 DC 0F 83 3E 52 00 00 00D0 : 7E 07 83 3E 52 00 0C 7E 58 BF 70 01 1E 57 9A DD 00E0 : 05 DC 0F 9A 91 02 DC 0F BF 70 01 1E 57 BF 21 00 00F0 : 0E 57 31 C0 50 9A 70 06 DC 0F A1 52 00 99 52 50 0100 : 31 C0 50 9A F8 06 DC 0F BF 2A 00 0E 57 31 C0 50 0110 : 9A 70 06 DC 0F 9A DD 05 DC 0F 9A 91 02 DC 0F BF ...
Das Anfangsstück dieses Programmabschnitts in INTEL 80x86-Assembler (2. und 3. Spalte)	9A0000DC0F CALL 0FDC:0000 9A0D007A0F CALL 0F7A:000D 55 PUSH BP 89E5 MOV BP,SP 31C0 XOR AX,AX 9ACD02DC0F CALL 0FDC:02CD 9ACC017A0F CALL 0F7A:01CC BF7001 MOV DI,0170 1E PUSH DS 57 PUSH DI BF0000 MOV DI,0000 0E PUSH CS 57 PUSH DI 31C0 XOR AX,AX 50 PUSH AX 9A7006DC0F CALL 0FDC:0670 ...

Abbildung 3-1: Maschinensprache und Assembler-Programm

Einen Programmausschnitt in Maschinensprache und das entsprechende Anfangsstück in INTEL 80x86-Assembler zeigt Abbildung 3-1. Üblicherweise werden die Werte nicht in Binär- sondern in Sedezimaldarstellung angegeben. Auf die Bedeutung einer Assembler-Sprache in der Systemprogrammierung wird in Kapitel 3.1 eingegangen.

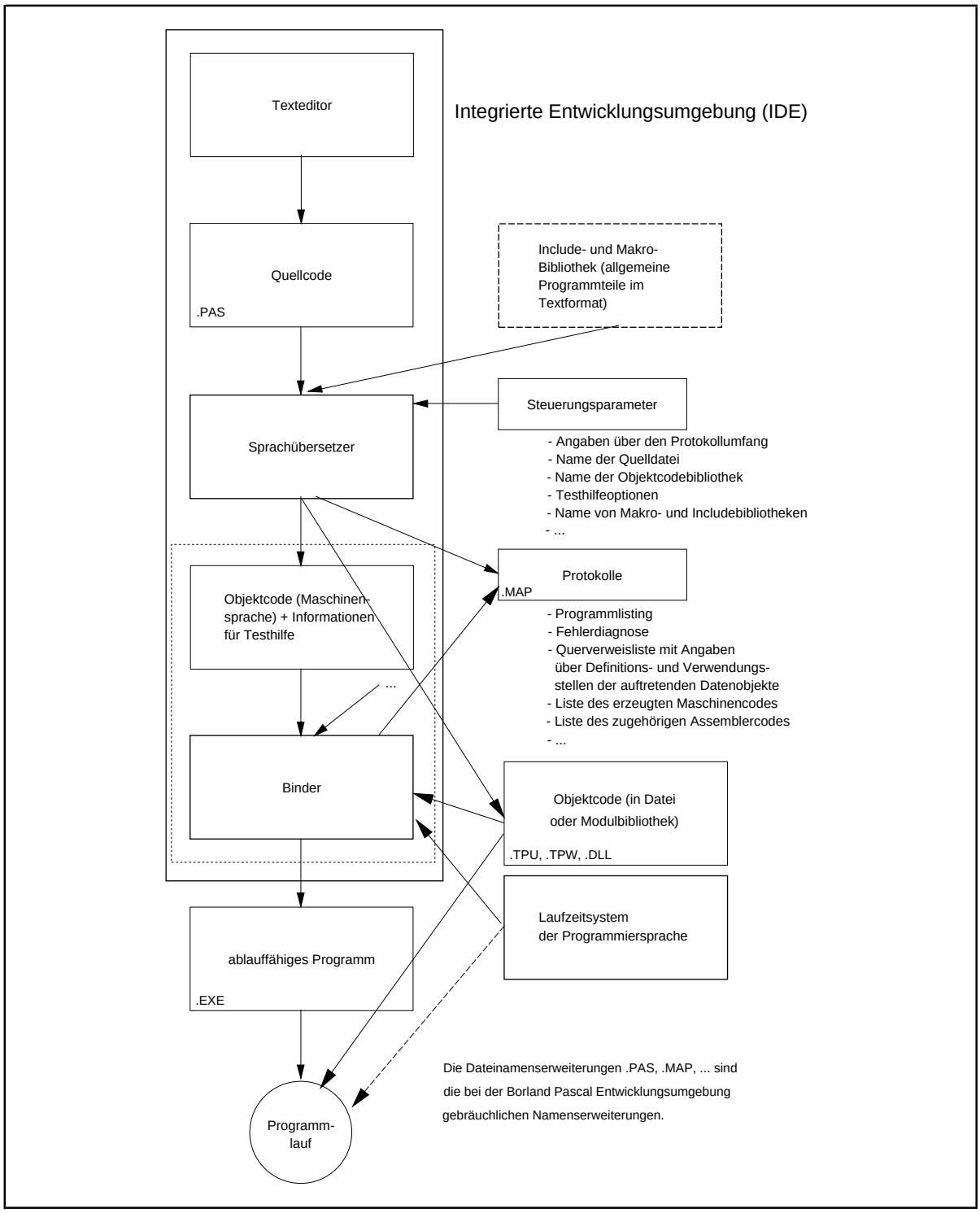


Abbildung 3-2: Programmerstellung

Bei der Programmerstellung in einer **höheren Programmiersprache (high order language, HOL)** werden im allgemeinen die im folgenden beschriebenen Schritte durchlaufen (vgl. Abbildung 3-2). Zu den höheren Programmiersprachen zählen z.B. Pascal, C++, Ada, Modula-2, Oberon, FORTRAN, COBOL und alle Sprachen der 4. Generation (4GL), nicht aber die Assemblersprachen der unterschiedlichen Rechnerfamilien. Die Sprache C nimmt eine Mittelstellung zwischen den Assemblersprachen und den höheren Programmiersprachen ein. Die Sprache Java, die ebenfalls zu den höheren Programmiersprachen zählt, wird aufgrund der Anforderung, plattformunabhängige Programmierung zu ermöglichen, anders behandelt; am Ende des Kapitels wird darauf eingegangen.

Ein Programm wird im lesbaren Textformat mit Hilfe eines (Text-) Editors als **Quellcode (Quelltext)** erzeugt. Dabei ist neben der korrekten Umsetzung der Programmiervorgaben und der Programmlogik selbstverständlich die **Syntax** der verwendeten Programmiersprache zu beachten. Die Syntax der Programmiersprache gibt die Regeln vor, nach denen eine Programmieraufgabe in dieser Sprache formuliert werden. Gelegentlich werden Textteile, die auf gleiche Weise auch in anderen Programmen vorkommen, nur benannt und dann während der folgenden Übersetzungsphase automatisch, gesteuert durch entsprechende Steuerungsparameter, in den Quelltext eingefügt. Ein derartiger Quelltextteil heißt **Include** bzw. **Makro (-expansion)**.

Der **Sprachübersetzer (Compiler)** der zugehörigen Programmiersprache übersetzt den Quellcode in **Objektcode**, der in der Maschinensprache des eingesetzten Rechners formuliert ist. Auch dieser Vorgang wird durch Parameter gesteuert, die beispielsweise angeben, welchen Zusatzinformationen über den Übersetzungsvorgang in Protokollen festgehalten werden sollen. Meist muß die Übersetzungsphase mehrere Male durchgeführt werden, insbesondere dann, wenn der Quellcode die Syntaxregeln verletzt. Die Übersetzung erfolgt im allgemeinen 1:*m*, d.h. aus einer Anweisung des Quellcodes in der höheren Programmiersprache werden mehrere Instruktionen in Maschinensprache erzeugt. Abbildung 3-3 zeigt, welcher Maschinencode aus einer Anweisung eines COBOL-Programms (für einen Rechner der SNI 7·500-Systemfamilie mit dem entsprechenden COBOL-Compiler) erzeugt wird. In diesem Fall entstehen aus einer einzigen COBOL-Anweisung zur Übertragung von Variablenwerten sechs Instruktionen der Maschinensprache, die neben der reinen Datenübertragung von einer in die andere Variable Datenformatierungen der beteiligten Variableninhalte vornehmen.

Bei den "klassischen" prozeduralen Sprachen wie Pascal, C, C++, COBOL, FORTRAN oder Ada kann man im Quellcode grundsätzlich zwischen Daten und Instruktionen unterscheiden. Entsprechend findet man auch in den erzeugten Objektcodemoduln Bereiche mit Daten und Bereiche mit Instruktionen. Je nach Programmiersprache und Implementierung für einen Rechner(-typ) ist diese logische Trennung von Code und Daten nach der Übersetzung auch mehr oder weniger physikalisch ausgeprägt. Unter Umständen werden aus einem Programm im Quellcode sogar mehrere Objektcodemoduln bzw. -dateien generiert.

COBOL-Anweisung:	erzeugte Instruktionen in Maschinensprache (in Klammern: maschinensprachlicher Operationscode in der zugehörigen Assembler-Sprache):

`MOVE EINK-WERT TO L-EINK-WERT.`
 	
(EINK-WERT und L-EINK-WERT sind unterschiedlich formatierte Daten der WORKING-STORAGE SECTION):
`EINK-WERT PIC 9(4)V99.`
`L-EINK-WERT PIC ZZZ9.9(2).`
	`F235C2A0403A (PACK)`
	`F835C2B0C2A0 (ZAP)`
	`960FC2B3 (OI)`
	`D208C460212E (MVC)`
	`DF08C460C2B0 (ED)`
	`D20630DBC462 (MVC)`

Abbildung 3-3: Übersetzung einer COBOL-Anweisung

Der aus dem syntaktisch korrekten Quellcode schließlich erzeugte Objektcode wird entweder in einer **Objektcodebibliothek (Modulbibliothek, Bindebibliothek, link library)** bzw. einer eigenen Objektcodedatei abgelegt oder vom **Binder (linker)** direkt zu einem auf dem Rechner ablauffähigen Programm gebunden. Bei diesem Vorgang fügt der Binder vorübersetzte Programmteile im Objektcodeformat, die er in entsprechenden Dateien oder Bibliotheken findet (auch hierzu werden Steuerungsparameter angegeben), an das zu erzeugende Programm an. Eine der wichtigsten Bibliotheken beinhaltet das **Laufzeitsystem** der Programmiersprache. Dabei handelt es sich um übersetzte (externe) Routinen, die in vielen Programmen vorkommen (z.B. Bildschirmsteuerung des Zielrechners), um Spracherweiterungen der Programmiersprache über den allgemeinen Standard hinaus oder um Programmteile, die den Zugriff auf Betriebssystemfunktionen und Rechnerkomponenten erst ermöglichen (z.B. Ein/Ausgaberoutinen).

Dieser Vorgang wird als **statisches Binden** bezeichnet, da vor der Laufzeit alle Adreßverweise auf externe Programmteile aufgelöst und der jeweilige Code angebunden wurde, auch wenn später während der Laufzeit dieser Code eventuell gar nicht durchlaufen wird. Daneben gibt es die Möglichkeit des **dynamischen Bindens**, d.h. des Anbindens externer Programmteile erst während der Laufzeit, wenn sie auch wirklich benötig werden. Auf die verschiedenen Bindemethoden wird später noch genauer eingegangen.

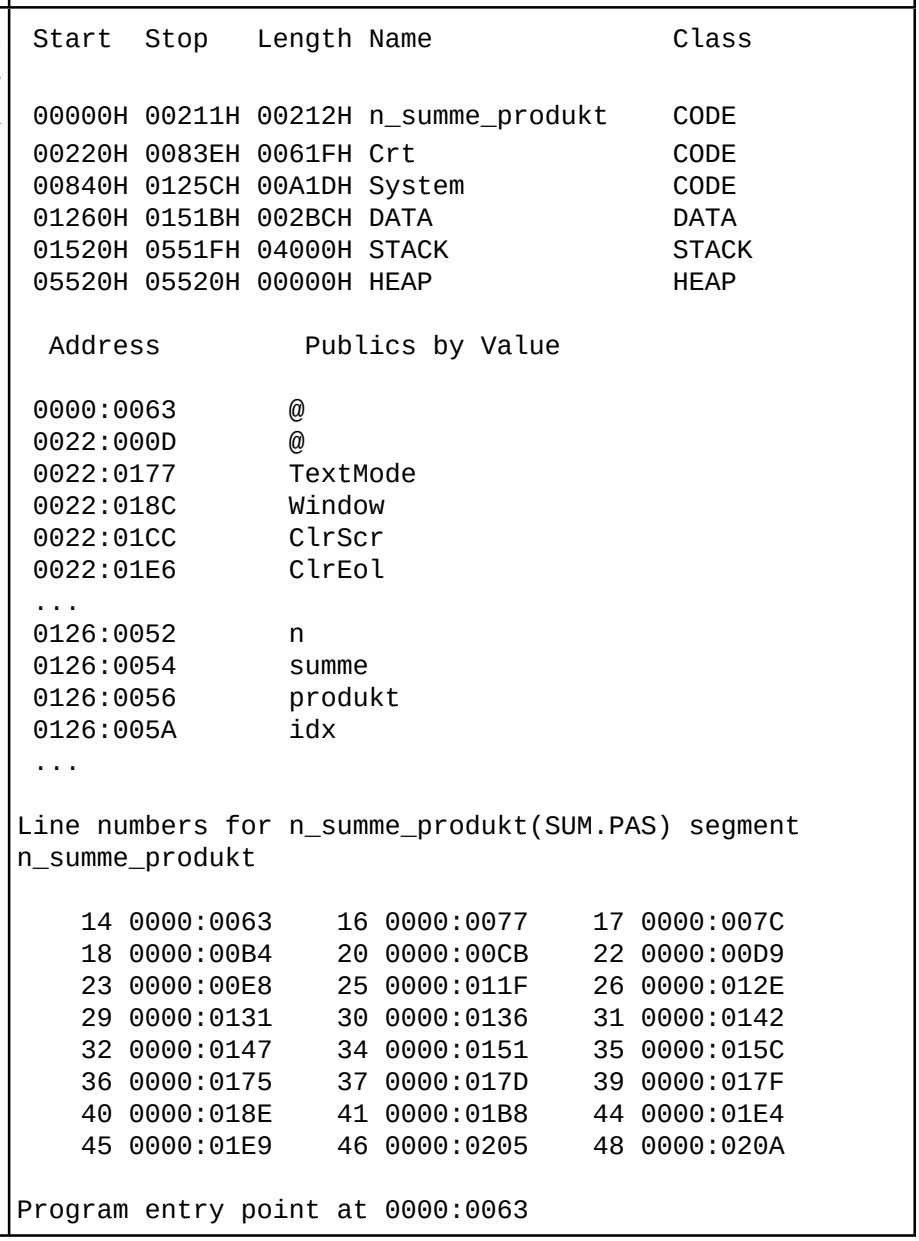
Das **ablauffähige Programm** wird in einer Datei abgelegt, deren Inhalt dann zum Programmstart vom Betriebssystem in den Arbeitsspeicher des Rechners geladen wird. Diese Datei enthält neben dem Programmcode und Datenbereichen rechnerspezifische Zusatzinformationen über die Strukturierung des Programms, eventuell vorkommende und während des Ladevorgangs noch anzupassende Adreßverweise, Angaben, an welche Stelle das Programm zu laden ist usw.

Abbildung 3-4 zeigt ausschnittsweise die Ergebnisse der Programmerstellung bei einem sehr einfachen PASCAL-Programm.

Quellcode (in Datei SUM.PAS)

Zeilen-nummer

```
 1  PROGRAM n_summe_produkt;
 2
 3  USES Crt;
 4
 5  CONST n_max = 12;
 6
 7  TYPE  bereich = 1..n_max;
 8
 9  VAR n       : INTEGER;
10      summe   : INTEGER;
11      produkt : LONGINT;
12      idx     : bereich;
13
14  BEGIN
15    { Einlesen }
16    ClrScr;
17    Write (’Bitte n eingeben (1 <= n <= ’, n_max, ’): ’);
18    Readln (n);
19
20    IF (n <= 0) OR (n > n_max)
21    THEN BEGIN
22           Writeln;
23           Writeln
24             (’FEHLER: ’, n, ’ ist nicht zulässigen’);
25           Writeln
26         END
27    ELSE BEGIN
28           { Initialisierung }
29           summe   := 0;
30           produkt := 1;
31           idx     := 1;
32           WHILE idx <= n DO
33           BEGIN
34             summe   := summe + idx;
35             produkt := produkt * idx;
36             idx     := idx + 1
37           END;
38           { Ausgabe }
39           Writeln;
40           Writeln (’Summe   = ’, summe);
41           Writeln (’Produkt = ’, produkt)
42          END;
43
44  HighVideo;
45  Writeln (’Programmende’);
46  Normvideo
47
48  END.
```

Ausschnitt aus dem ausführbaren Programm in Maschinensprache (in Datei SUM.EXE)	siehe Abbildung 3-1.
Das Anfangsstück dieses Maschinencodes in 80x86-Assembler	siehe Abbildung 3-1.
Ausschnitt aus dem vom Binder erzeugten Protokoll (in Datei SUM.MAP)	

```
 Start  Stop   Length Name               Class

 00000H 00211H 00212H n_summe_produkt    CODE
 00220H 0083EH 0061FH Crt                CODE
 00840H 0125CH 00A1DH System             CODE
 01260H 0151BH 002BCH DATA               DATA
 01520H 0551FH 04000H STACK              STACK
 05520H 05520H 00000H HEAP               HEAP

  Address         Publics by Value

 0000:0063       @
 0022:000D       @
 0022:0177       TextMode
 0022:018C       Window
 0022:01CC       ClrScr
 0022:01E6       ClrEol
 ...
 0126:0052       n
 0126:0054       summe
 0126:0056       produkt
 0126:005A       idx
 ...

Line numbers for n_summe_produkt(SUM.PAS) segment
n_summe_produkt

    14 0000:0063    16 0000:0077    17 0000:007C
    18 0000:00B4    20 0000:00CB    22 0000:00D9
    23 0000:00E8    25 0000:011F    26 0000:012E
    29 0000:0131    30 0000:0136    31 0000:0142
    32 0000:0147    34 0000:0151    35 0000:015C
    36 0000:0175    37 0000:017D    39 0000:017F
    40 0000:018E    41 0000:01B8    44 0000:01E4
    45 0000:01E9    46 0000:0205    48 0000:020A

Program entry point at 0000:0063
```

Abbildung 3-4: Ergebnisse bei der Programmerstellung in PASCAL (Beispiel)

Besonders in der PC-Welt sind heute komfortable **integrierte Entwicklungsumgebungen (IDE, integrated development environment)** für die verschiedenen Programmiersprachen gebräuchlich, die Komponenten für alle Einzelschritte des Programmerstellungsprozesses in ein Standardsoftwarepaket zusammenfassen und auf die spezifischen Belange der Programmentwicklung abgestimmt sind. Die Fehlersuche wird durch leistungsfähige Testhilfen sehr gut unterstützt. Zudem erzeugen die Sprachübersetzer durch den Einsatz von Optimierungsverfahren sehr kompakten Code. Ein Beispiel ist das System Delphi der Firma Inprise (früher Borland), vgl. [WAR].

In den häufigsten Anwendungen erfolgt die Interaktion des Benutzers mit einem Programm über weitgehend standardisierte **Dialogelemente** wie Auswahlmenüs, -listen und -knöpfe (mit mehreren anzukreuzenden Alternativen oder genau einer Alternative), einfache Texteingabezeilen usw. Die Dialogelemente sind aus Gründen der räumlichen Begrenzung mit verschiedenen Arten von Rollbalken versehen und werden mit der Maus angesteuert. Typischerweise werden sie in mehr oder weniger überlappenden Fenstern angeordnet, deren Größe und Position auf dem Bildschirm zur Laufzeit veränderbar sind. Bei Auswahl eines Dialogelements mit der Maus wird eine definierte Aktion ausgelöst, meist als **Ereignis** (**event**) bezeichnet, das an einen Ereignisbehandlungsmechanismus weitergereicht wird, der selbständig den Programmteil findet, der für die Behandlung des Ereignisses zuständig ist.

Seit Einführung objektorientierten Programmierung ist es üblich, daß Programmiersysteme derartige Dialogelemente und Ereignisbehandlugsmechanismen in Form von vordefinierten Objektklassen dem Programmierer zur Verfügung stellen. Er hat dann nur noch die Anordnung der Dialogelemente auf einem Dialogformular, die einzelnen Ereignisse und die Aktionen z.B. in Form von Prozeduren und das individuelle Layout der Dialogelemente (Beschriftungen, Texte von Auswahlalternativen, spezielles Aussehen usw.) zu definieren. Die Grundmechanismen, über die jedes Dialogelement verfügt, sind in den Methoden der entsprechenden Objektklasse implementiert.

Neuere Entwicklungsumgebungen wie Delphi unterstützen den Anwender weitgehend bei der Erstellung von Dialoganwendungen: Beim Entwurf werden die einzelnen Dialogelemente interaktiv *im Dialog mit der Entwicklungsumgebung* in ein anfangs leeres Formular eingefügt. Dabei stehen Paletten von unterschiedlichen Dialogelementen zur Verfügung, aus denen mit der Maus das jeweilige Element ausgewählt und an die gewünschte Position auf dem Formular gezogen wird. Beschriftungen und individuelles Layout werden ebenfalls so definiert. Die Entwicklungsumgebung generiert selbstätig die entsprechenden (Object Pascal-) Programmanweisungen. Lediglich die Aktionen, die durch Aktivierung eines Dialogelements ausgelöst werden sollen, müssen noch in den generierten Code eingefügt werden. Auch hierbei gibt es eine weitgehende Unterstützung durch die Entwicklungsumgebung.

Neben den Dialogelementen stellen heutige Entwicklungsumgebungen weitere vordefinierte und z.T. vorübersetzte Programmteile in Form von (Klassen-) Bibliotheken bereit, die von häufig benötigten Datenstrukturen über Komponenten, die den Zugriff auf alle Rechnerteile unterstützen, Fenstersysteme, Kommunikationsmechanismen, Multitasking usw. bis hin zu kompletten Anwendungskomponenten für spezifische Applikationen beinhalten.

Der in Abbildung 3-2 beschriebene Prozeß der Programmerstellung erzeugt aus eventuell weitgehend plattformunabhängigem Quellcode hochgradig maschinenabhängigen Objektcode. Dieser Vorgang ist natürlich für Sprachen, die plattformunabhängige Programmierung erlauben sollen wie beispielsweise **Java** (vgl. [HEN]) so nicht durchführbar. Abbildung 3-5 zeigt vereinfacht die Vorgehensweise bei der Programmerstellung mit Java.

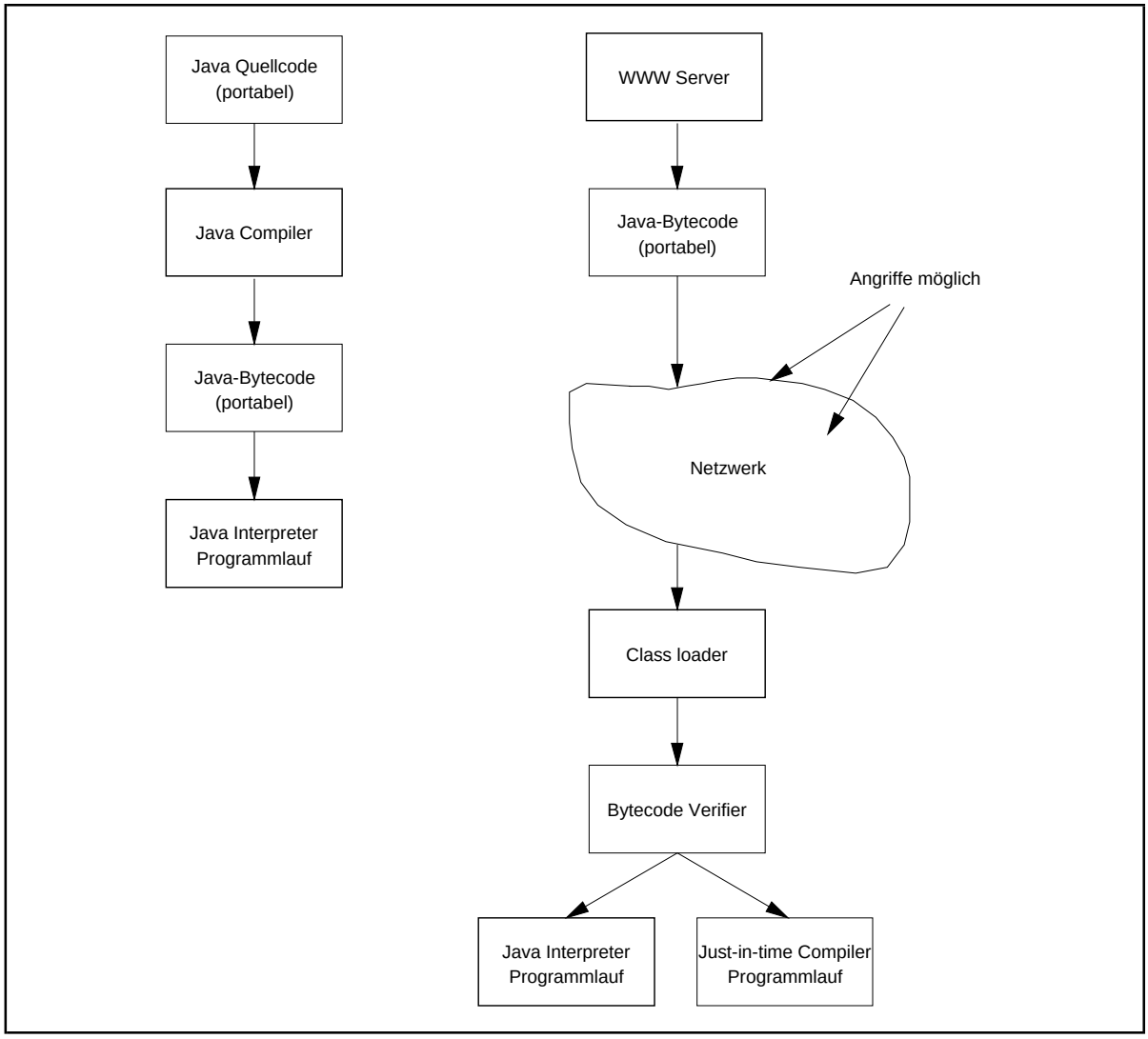


Abbildung 3-5: Programmerstellung mit Java

Der portable (plattformunabhängige) Java-Quellcode wird von einem Java-Compiler in portablen (plattformunabhängigen) **Java-Bytecode** übersetzt. Dabei handelt es sich um den Maschinencode für einen abstrakten Prozessor, die sogenannte **Java virtual machine**[6]. Um den Java Bytecode auf einem Rechner ausführen zu können, wird dort ein Java-Interpreter gestartet,

[6] Die Idee, eine abstrakte Zielmaschine für das Ergebnis der Compilation zu wählen, wurde bereits vor Jahrzehnten für einige Pascal-Compiler (Standard-Pascal) zur Erzeugung plattformunabhängiger Programme erfolgreich umgesetzt.

der den Java-Bytecode der Java virtual machine in Maschinensprache für den Rechner übersetzt. Damit können Java-Programme auf allen Rechnern ablaufen, für die ein Java-Interpreter zur Verfügung steht.

In der Regel werden Java-Programme, die z.B. in HTML-Seiten integriert sind (Java Applets), während der Ausführungszeit über ein unsicheres Netzwerk wie das Internet aus einem entsprechenden Server nachgeladen. Um das Einschleusen von Viren, trojanischen Pferden und ähnlichen Softwareteilen über den Nachladevorgang zu verhindern, verfügt Java über ein ausgefeiltes Schutzkonzept. Beispielsweise ist der Zugriff auf das lokale Dateisystem des Rechners durch derartige nachgeladene Programme zunächst verboten und kann nur auf Wunsch des Benutzers ausgeführt werden. Sprachkonzepte von Java wie der Verzicht auf Zeiger, die automatische Speicherverwaltung und das Objektkonzept stellen weitere Sicherheitsmaßnahmen dar. Ein über ein Netz geladener Java-Bytecode wird zunächst im **Java-Bytecode Verifier** auf Viren und mögliche falsche Objektzugriffe untersucht, bevor er dem Java-Interpreter oder einem online aufgerufenen Compiler (just-in-time Compiler) übergeben wird.

3.1 Höhere Programmiersprachen und Assembler-Sprachen in der Systemprogrammierung

In der Systemprogrammierung, hauptsächlich bei der Entwicklung von Betriebssystemen und anderen hardwarenahen Programmen, benötigt man zur Steuerung der Hardware Zugriff auf alle Rechnerkomponenten. Außerdem müssen der Programmcode schnell ausführbar sein und die Eigenschaften der Hardware optimal genutzt werden. Diese Punkte sprechen für den Einsatz der Assembler-Sprache eines Rechners. Demgegenüber stehen der hohe Codierungs- und Testaufwand von Assemblerprogrammen, ihre schlechte Wartbarkeit und die geringe Portabilität, gelegentlich selbst innerhalb derselben Rechnerfamilie.

Programme in einer höheren Programmiersprache werden heutzutage mit den optimierenden Compilern der Sprache in Maschineninstruktionen übersetzt, deren Laufzeitverhalten entsprechenden Assemblercode häufig noch übertrifft. Die übrigen Nachteile der Assemblerprogrammierung treten beim Einsatz einer höheren Programmiersprache meist ebenfalls nicht auf. Jedoch fehlen (standardmäßig) Sprachmittel zum direkten Ansteuern der Hardware. Die Forderung nach Portabilität der Programme steht dem entgegen. Das Setzen von Registern, die Veränderung von Arbeitsspeicherinhalten oder das Senden von Daten an einen Kommunikationsanschluß ist rechnerabhängig und auf andere Rechner (-typen) in der Regel nicht übertragbar. Um diesem Nachteil für die Systemprogrammierung abzuhelfen, könnte man geeignete Sprachmittel in die Programmiersprache einbauen. Man muß dann ein allgemeingültiges Rechnermodell unterstellen, das im konkreten Fall auf den jeweiligen Rechnertyp abgebildet wird, um wenigstens einen Rest an Portabilität zu wahren. Die Programmiersprache enthielte dann Anweisungen zur direkten Programmierung der CPU-Register, des Arbeitsspeichers, der Geräte usw. Die Vorteile einer höheren Programmiersprache werden aber gerade durch den Verzicht auf einen derartigen Ansatz ermöglicht.

Ein Ansatz, der keine Sprachspezialisierung auf einen Rechnertyp benötigt, sieht den Anschluß externer Assemblerprogramme durch Unterprogrammsprünge vor. Das bedeutet aber neben dem durch die Unterprogrammtechnik auferlegten Laufzeitverlust, daß die Programmteile, die den direkten Hardwarezugriff erfordern, jeweils zusammengefaßt und in Assemblerroutinen darstellbar sind. Einen Schritt weiter gehen Programmiersprachendialekte, die durch geeignete Erweiterungen das Einfügen von Assembler- und Maschinencodesequenzen in die Programme auf Sprachebene ermöglichen; natürlich sollte man hiervon nur äußerst sparsamen Gebrauch machen, da so die Portabilität der Programme verhindert wird. Diese Sprachdialekte sind dann natürlich wieder nur für einen speziellen Rechnertyp geeignet (z.B. INTEL 80x86).

Beispielsweise werden in Borland Pascal-Programmen 80x86-Assembleranweisungen in die Schlüsselwörter **`ASM`** und **`END`** eingeschlossen. Bei der Übersetzung werden die Anweisungen durch den integrierten Assembler in Maschinencode übersetzt. Auch das direkte Einfügen von Maschinencode ist möglich, und zwar mit der **`INLINE-Anweisung`**. Auf beide Möglichkeiten soll hier nicht weiter eingegangen werden, da dem Aufruf externer Prozeduren trotz des Laufzeitverlusts durch den Unterprogrammsprung der Vorrang gegeben wird.

Eine weitere Möglichkeit, die Verwendbarkeit einer höheren Programmiersprache für die Systemprogrammierung zu erhöhen, besteht in der (vorsichtigen) Spracherweiterung bezüglich der Datentypen. Dadurch können Bitoperationen oder Operationen, die die beteiligten Datentypen weitgehend ignorieren, so wie in der Assemblerprogrammierung, zugelassen werden. Und schließlich ist ein wesentlicher Punkt das Vorhandensein eines Pointerkonzepts, das eine Adreßarithmetik zuläßt, die die "Sicherheitsmechanismen" der Programmiersprache wie Kompatibilitätsüberprüfungen der beteiligten Datentypen schon während der Übersetzungszeit jedoch nicht außer Kraft setzen sollte[7].

Insgesamt kann durch diese Sprachkonzepte und Spracherweiterungen die Notwendigkeit zum Einsatz der Assemblersprache eines Rechners in der Systemprogrammierung reduziert werden.

3.2 Adreßräume und das Prozeßkonzept

Die einzelnen Speicherzellen des in einem Rechner installierten Arbeitsspeichers werden rechnerintern im Ausführungszeitpunkts eines Programms über **physikalische Arbeitsspeicheradressen** angesprochen, wobei der Arbeitsspeichernumerierung **linear** von 0 bis zu einer Obergrenze geht, die durch seinen Ausbau bestimmt ist. Die Adreßbreite und die im Einzelfall variierende Größe des Arbeitsspeichers bestimmt den **physikalischen Adreßraum** des einzelnen Rechners. In der Systemprogrammierung nehmen auch Programme häufig Bezug auf physikalische Adressen, insbesondere bei der Ansteuerung von speicherabgebildeten Geräten (z.B. Bildschirmspeicher), von I/O-Ports usw.

---

[7] Gerade in diesem Punkt zeichnet sich PASCAL gegenüber Programmiersprachen wie C besonders aus.

Normalerweise erzeugt ein Compiler für ein Programm keine direkten physikalischen Adressen. Das würde nämlich bedeuten, daß ein übersetztes Programm nur an einer fest definierten Stelle im Arbeitsspeicher laufen kann und seine Datenobjekte immer an festen Stellen liegen, die bereits zur Übersetzungszeit des Programms bekannt sein müssen. Man würde also einen Teil der Arbeitsspeicheradressen fest für das jeweilige Programm vorsehen. Eine flexible und ökonomische Systemnutzung erscheint so unmöglich.

Jeder Rechner (-typ) verfügt vielmehr über eine "Sicht" seines Arbeitsspeichers, die wesentlich angibt, wie dieser adressiert wird. Die so definierte **Adressierungsmethode**, die das **Speichermodell des Rechners** festgelegt, bestimmt, wie innerhalb von Maschineninstruktionen Adressen zu bilden sind. Meist wird eine derartige Adresse nicht als physikalische Adresse eingesetzt, sondern sie besteht aus mehreren Teilen, z.B. aus einem sogenannten **Basiswert** und einem **Offset**. Der Compiler erzeugt aus Quellcode Objektmoduln gemäß dem vereinbarten Speichermodells des Zielrechners. Man erreicht dadurch, daß zur Übersetzungszeit noch nicht genau bestimmt werden muß, an welcher Stelle im Arbeitsspeicher das Programm später liegen wird; erst zur Laufzeit (nach dem Laden des Programms) und mit weiterer Unterstützung von Hardwareregistern (Adressierungsregister) und eventuell durch das Betriebssystem werden die Programmadressen in physikalische Arbeitsspeicheradressen umgesetzt. Beispielsweise besitzt der INTEL 80386-Prozessor (und nachfolgende Rechner dieser Rechnerfamilie) sogar mehrere, wenig miteinander kompatible Speichermodelle, die mit Real Mode, Protected Mode und Flat Mode bezeichnet werden und noch weitere Varianten aufweisen (siehe auch Kapitel 3.3).

In Multiprocessingsystemen, in denen mehrere unabhängige und um die Rechnerressourcen konkurrierende Prozesse unter Kontrolle des Betriebssystems einen Rechner parallel (genauer: zeitlich verzahnt und damit pseudo-parallel) nutzen, wird von der real verfügbaren Hardware noch weiter abstrahiert, und entsprechend ist die von einem Compiler generierte Adressierung noch anders zu interpretieren. Im **Prozeßkonzept eines Betriebssystems** (vgl. z.B. [TAN], [BRA]), das eine ***Virtualisierung der Hardware*** liefert, sieht jeder Prozeß (Abbildung 3.2-1)

- einen eigenen Satz von virtuellen Registern

- einen eigenen virtuellen Adreßraum

- eine virtuelle Prozeßumgebung.

Der virtuelle Registersatz eines Prozesses und sein virtueller Adreßraum sind von den virtuellen Registern und den virtuellen Adreßräumen anderer Prozesse dabei vollständig getrennt.

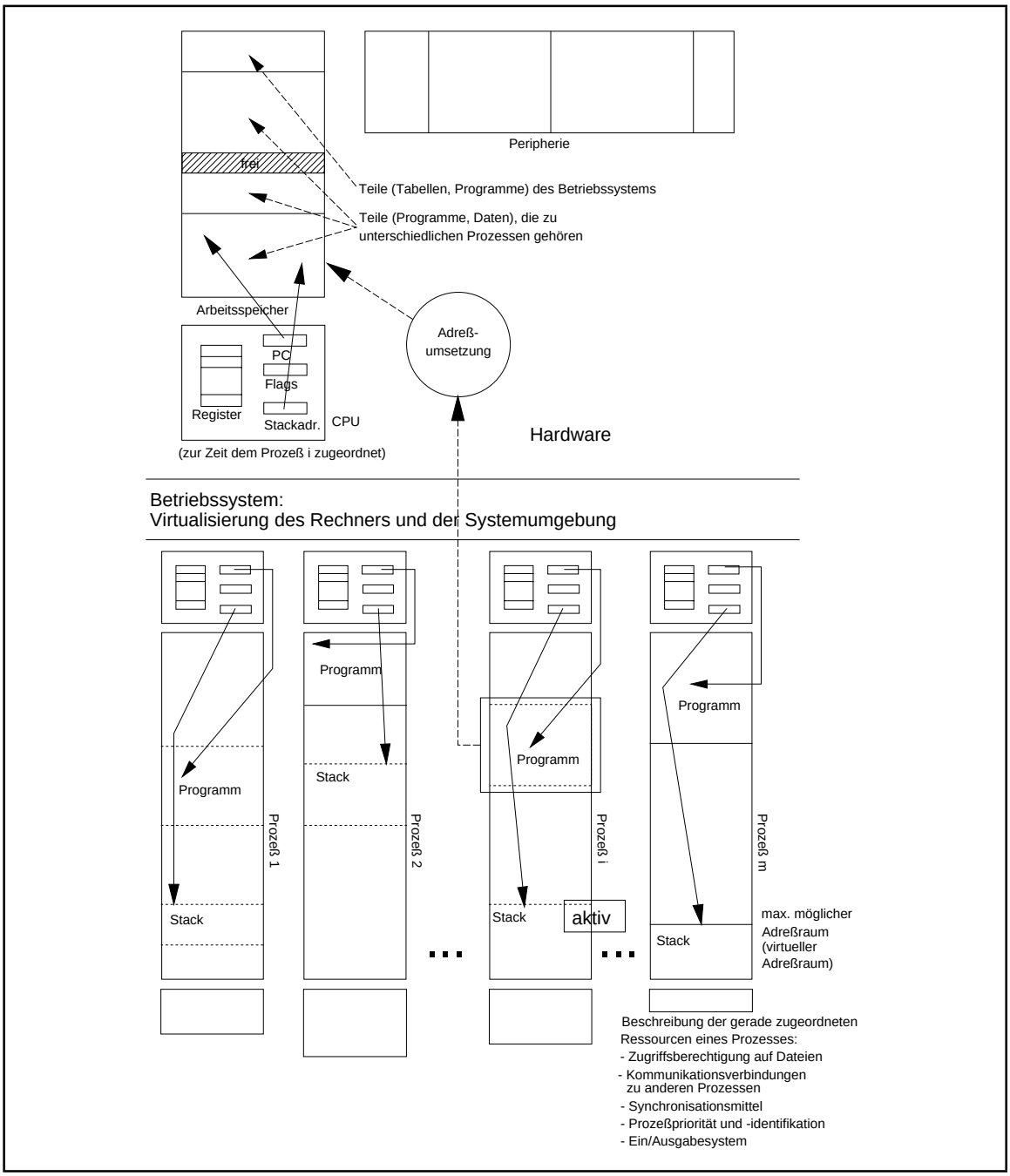


Abbildung 3.2-1: Prozeßkonzept

Die **virtuellen Register** sind ein Bild des CPU-Registersatzes, so wie sie von einem Prozeß gesehen werden, wenn er allein über die Register verfügen würde. Insbesondere kennt ein Prozeß "seine" Inhalte der Rechen- und Anzeigenregister und den Inhalt seines (virtuellen) Befehlszählerregisters, der bezogen auf seinen virtuellen Adreßraum die Adresse des nächsten auszuführenden Befehls enthält. Zusätzlich adressiert ein virtuelles Register den prozeßeigenen Stack (siehe unten).

Der **virtuelle Adreßraum** eines Prozesses ist das "logische" Bild des bezüglich der möglichen Adressen maximal verfügbaren Arbeitsspeichers, und zwar so, als würde der Prozeß allein über den Arbeitsspeicher verfügen. Die Programmteile (Code, Daten, Stack), die ein Prozeß durchläuft bzw. verwendet, nehmen Bezug auf **virtuelle Adressen**. Die Größe des virtuellen Adreßraums wird allein durch die Adressierungsbreite des Rechners bestimmt. Für jeden Prozeß in einem Betriebssystem ist sein virtueller Adreßraum prinzipiell wie für alle anderen Prozesse strukturiert (die Inhalte der virtuellen Adreßräume der einzelnen Prozesse unterscheiden sich natürlich). Diese Strukturierung nennt man das **Speichermodell des Betriebssystems**. Das einfachste Speichermodell ist das **flache Speichermodell**, in dem die virtuellen Adressen von 0 bis $2^n - 1$ (bei einer Adreßbreite von n Bits) durchnumeriert sind. Andere Speichermodelle sind das **segmentierte Speichermodell**, das **Speichermodell mit fester Adreßzuordnungen** usw. Bei flachem Speichermodell und den heute in der kommerziellen Informationsverarbeitung üblichen Adressierungsbreiten ergeben sich Größen der virtuellen Adreßräume, wie sie in folgender Tabelle gezeigt werden.

Adressierungsbreite der Hardware	virtueller Adreβbereich
20 Bits	0, ..., 1.048.576
24 Bits	0, ..., 16.777.215
25 Bits	0, ..., 33.554.431
31 Bits	0, ..., 2.147.483.647
32 Bits	0, ..., 4.294.967.295
64 Bits	0, ..., $> 1.8 \cdot 10^{19}$
n Bits	0, ..., 2^n-1

Durchläuft ein Prozeß ein Programm, so kann man die in dem Programm vorkommenden Adressen als Adressen interpretieren, die sich auf den virtuellen Adreßraum dieses Prozesses beziehen. Der Compiler des Programms hat natürlich in diesem Fall virtuelle Adressen generiert. Das Programm wird also vor seiner Ausführung so gesehen, als sei es in den virtuellen Adreßraum des Prozesses geladen, der das Programm durchläuft, bzw. man kann sagen, der Prozeß durchlaufe ein Programm in seinem virtuellen Adreßraum. Zu diesem Zeitpunkt braucht das Programm noch nicht physisch in den Arbeitsspeicher geladen zu sein, sondern es ist dem virtuellen Prozeßraum und damit dem Prozeß "zugeordnet". Ein Programm kann so in unterschiedlichen virtuellen Adreßräumen liegen, ohne daß die beteiligten Prozesse wegen der strikten Trennung der Prozesse davon Kenntnis haben.

Zur Realisierung der Unterprogrammtechnik und der Interruptbehandlung verfügt jeder Prozeß über einen als **Stack** bezeichneten (virtuellen) Speicherbereich, der über ein spezielles Stackregister adressiert wird.

Betriebssystemabhängig ist der virtuelle Adreßraum eines Prozesses gelegentlich in zwei Teile unterteilt: in den (virtuellen) **Benutzeradreβraum** und den (virtuellen) **Systemadreβraum**. Der Inhalt des Benutzeradreßraum ist prozeßspezifisch belegt und enthält diejenigen Programme und Daten eines Prozesses, die er im Benutzermodus

verwendet (einschließlich seines Benutzerstacks). Führt ein Prozeß gerade ein Programm des Betriebssystems im Systemmodus aus, so läuft dieses logisch gesehen im virtuellen Adreβraum dieses Prozesses, und zwar im Systemadreβraum.

Zur **virtuellen Prozeßumgebung** eines Prozesses gehört i.a. ein **virtuelles Terminal** bzw. ein **virtueller Bildschirm** mit **virtueller Eingabetastatur**, um die Verbindung zu einem menschlichen Anwender zu ermöglichen. Weiterhin wird in der virtuellen Prozeßumgebung beschrieben, über welche **Systemressourcen** (Arbeitsspeicherbereiche, Zugriffsberechtigungen auf Dateien, Zeitgeber, Hardwarekomponenten), **Interruptbehandlungsroutinen**, **Kommunikationsverbindungen zu anderen Prozessen**, **Synchronisationsstruktur mit anderen Prozessen** ein Prozeß zu dem jeweiligen Zeitpunkt verfügt.

Damit die Aktionen eines Prozesses real ablaufen können (der Prozeß ist dann **aktiv**, d.h. die Aktionen in der CPU beziehen sich auf diesen Prozeß), müssen seine "virtuellen Bestandteile" auf die Hardware des Rechners abgebildet werden, d.h. er muß über die CPU des Rechners, Teile des Arbeitsspeichers und seiner virtuellen Systemumgebung entsprechende Hardwarekomponenten verfügen. Insbesondere werden auch virtuelle Ein-/Ausgabegeräte (Terminal, Bildschirm oder Tastatur) auf reale Geräte (beim virtuellen Bildschirm eventuell auf ein einzelnes Bildschirmfenster) abgebildet. Bei einer einzigen CPU kann immer nur höchstens ein Prozeß aktiv sein, d.h. über die "realen" CPU-Register verfügen. Bei Multiprocessing mit *m* CPUs können bis zu *m* Prozesse gleichzeitig aktiv sein. Das Umschalten von einem aktiven Prozeß auf einen anderen bedeutet, daß derjenige Prozeß, der bisher über die realen CPU-Register verfügen konnte, nun die Registerverwendung an einen anderen Prozeß abgibt. Dieser Vorgang heißt **Prozeßwechsel** und findet in den gängigen Systemen viele Male pro Sekunde statt und bedarf daher einer intensiven Hardwareunterstützung in Form entsprechender Maschineninstruktionen. Details dieser betriebssysteminternen Abläufe werden exemplarisch und vereinfacht in Kapitel 11 behandelt.

Das Betriebssystem regelt die Abbildung des virtuellen Adreßraums eines Prozesses auf die reale Hardware. Wird ein Prozeß aktiv, d.h. werden ihm nun die CPU-Register zugeordnet, muß auch der Teil seines virtuellen Adreßraums, der die nächste auszuführende Anweisung und die zugehörigen Datenbereiche enthält, in den Arbeitsspeicher geladen worden sein (Speicherverwaltung des Betriebssystems). In der Regel werden dabei zusammenliegende Bereiche des virtuellen Adreßraums nicht auf genauso zusammenliegende Bereiche des Arbeitsspeichers abgebildet. Zusätzlich wird während der Lebensdauer eines Prozesses dieser Abbildungsvorgang wiederholt durchgeführt (ein Prozeß kann immer wieder deaktiviert und der von ihm gerade belegte Arbeitsspeicherbereich anderen Prozessen zur Verfügung gestellt werden), wobei einem virtuellen Adreßbereich dann nicht notwendigerweise derselbe physikalische Arbeitsspeicherbereich zugeordnet wird. Daher muß vor jedem Zugriff eines gerade aktiven (in der CPU laufenden) Prozesses auf eine virtuelle Adresse, die ja in der jeweiligen Maschineninstruktion benutzt wird, eine **Adreßumsetzung von der virtuellen Adresse auf eine reale Arbeitsspeicheradresse** erfolgen. ***In der Regel erfolgt die Adreßumsetzung einer virtuellen in eine reale Arbeitsspeicheradresse mit Hardwareunterstützung und gesteuert von prozeßspezifischen Adreßumsetzungstabellen***, auf die nur das Betriebssystem zugreifen darf. Die

Hardwareunterstützung erwartet dabei, daß diese Tabellen für jeden Prozeß existieren und korrekt gepflegt werden, eine Aufgabe, die der Speicherverwaltung des Betriebssystems zufällt. Bei einem Prozeßwechsel schaltet dann der Prozeßwechselmechanismus von den Tabellen des bisher aktiven Prozesses auf die Adreßumsetzungstabellen des zu aktivierenden Prozesses um.

In Erweiterung dieses Konzepts liegt es nahe, auch mehrere ***(pseudo-) parallel ablaufende Programmläufe innerhalb eines Prozesses*** zuzulassen. Jeder einzelne Programmlauf eines Prozesses verwendet ein Programm, das in dessen virtuellem Adreßraum liegt, und zwar belegen verschiedene Programme eines Prozesses unterschiedliche Adreßbereiche des virtuellen Adreßraums. Innerhalb eines Prozesses ist immer ***genau ein Programmlauf aktiv***. Die (Pseudo-) Parallelität entsteht dadurch, daß einzelne Programmläufe begonnen, aber noch nicht beendet wurden, weil zwischenzeitlich Instruktionen anderer Programmläufe des Prozesses ausgeführt werden. In diesem Fall werden diese parallelen Programmläufe eines Prozesses auch **schwach nebenläufige Prozesse (Threads, Leichtgewichtsprozesse)** genannt. Zu einem Prozeß gehört also mindestens ein Thread.

Unterschiedliche Threads eines Prozesses können durchaus durch dasselbe Programm laufen und dort jeweils bis zu unterschiedlichen Anweisungen gekommen sein. Natürlich muß dabei gewährleistet sein, daß jeder Thread für das Programm über einen eigenen Satz lokaler Daten verfügt (mit lokalen Daten sind hier Datenobjekte gemeint, die nur in diesem Programm deklariert sind und daher nur hier verwendet werden). Die in Kapitel 7 beschriebene Realisierung des Prozedurkonzepts stellt dieses sicher, allerdings nur bezogen auf lokale Datenobjekte einer Prozedur (eines Programmteils). Im Prozeß globale Datenobjekte sind für alle Threads sichtbar und zugreifbar und müssen gegebenenfalls durch geeignete Synchronisationsmechanismen vor konkurrierendem Zugriff der Threads geschützt werden (siehe Kapitel 11).

Die Threads eines Prozesses haben gemeinsam:

- den virtuellen Adreßraum des Prozesses, einschließlich der Tabellen zur Umsetzung von virtuellen Adressen auf physikalische Arbeitsspeicheradressen

- die virtuelle Prozeßumgebung, d.h.

    -- die gegenwärtige Ressourcenbelegung des Prozesses (geöffnete Dateien, Gerätebelegung, Zugriffsberechtigungen, Schutzattribute usw.)
    -- die Kommunikationsverbindungen zu anderen Prozessen, insbesondere Warteschlange für den Empfang von Nachrichten usw.
    -- die Synchronisationsmittel des Prozesses, einschließlich der Tabellen zur Bearbeitung von definierten Ereignissen für den Prozeß
    -- die Interruptbehandlungsroutinen
    -- Prozeßidentifikation und Prozeßpriorität, von der eventuell eine eigene Priorität für den Programmlauf abgeleitet wird.

Jeder Thread eines Prozesses besitzt:

- ein Programm im virtuellen Adreßraum des Prozesses (unterschiedliche Threads können durch dasselbe Programm laufen)

- einen eigenen Satz virtueller Register, insbesondere ein eigenes virtuelles Befehlszählerregister, so daß verschiedene Programmläufen eines Prozesses zeitlich verzahnt ablaufen können

- einen eigenen Stack

- eventuell ein eigenes virtuelles Terminal bzw. ein Ausgabefenster innerhalb des virtuellen Terminals des zugehörigen Prozesses

Threads innerhalb eines Prozesses könnnen Daten und Nachrichten über den gemeinsamen virtuellen Adreßraum des Prozesses austauschen. Dagegen können ***unterschiedliche Prozesse Nachrichten nur mit Hilfe von Betriebssystemdiensten austauschen***, da die virtuellen Adreßräume verschiedener Prozesse vollständig voneinander getrennt sind und nur das Betriebssystem die Gesamtheit aller virtuellen Adreßräume kennt. Das Betriebssystem liefert **Dienste zur Prozeßsynchronisation zwischen Prozessen** und - je nach Betriebssystem - auch zwischen Threads einzelner Prozesse. Der Wechsel des Aktivzustands zwischen Threads eines Prozesses ist weitaus weniger zeitaufwendig als der Wechsel zwischen verschiedenen Prozessen, da bei einem Threadwechsel die gesamte Prozeßumgebung unverändert bleibt.

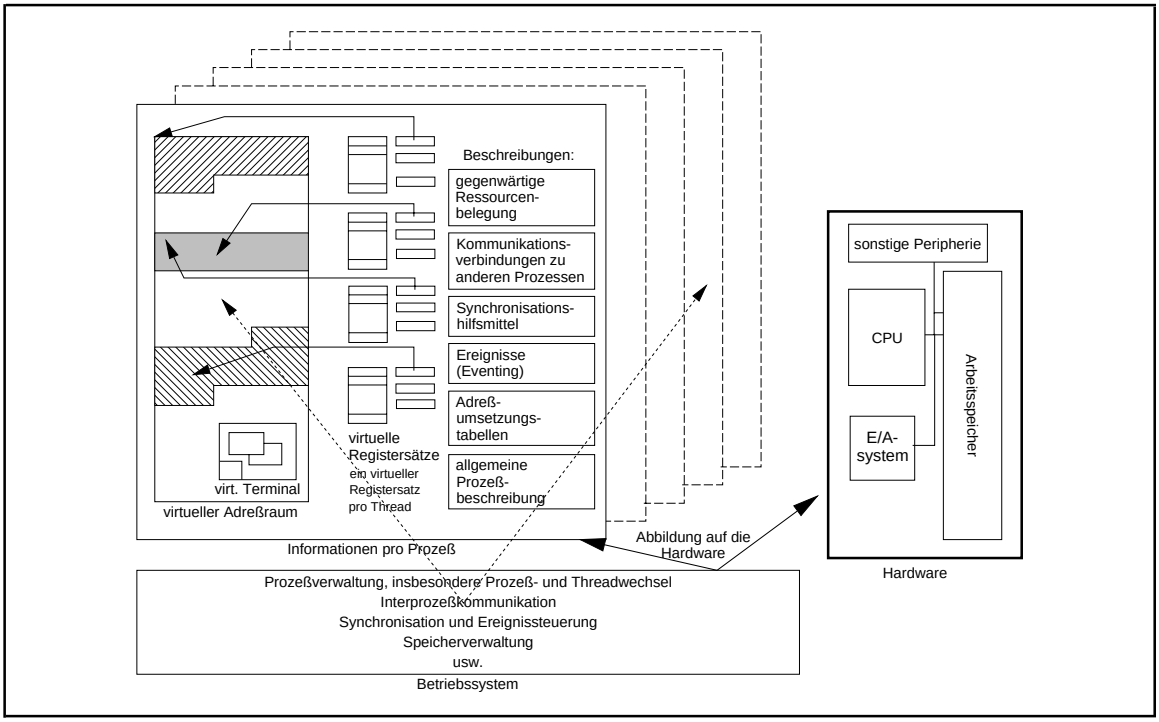


Abbildung 3.2-2: Das Prozeßmodell

Abbildung 3.2-2 stellt die wichtigsten Informationen bezüglich eines Prozesses im hier beschriebenen Prozeßmodell dar (es handelt sich hierbei um die "logischen" Bestandteile des Prozeßmodells; physisch sind diese an unterschiedlichen Stellen im System realisiert, z.B. in den Stacks des Prozesses, im Task Control Block, in verschiedenen Warteschlangen usw; siehe auch Kapitel 11). Die Pfeile aus den virtuellen Registersätzen in den virtuellen Adreßraum deuten die Inhalte der Befehlszählerregister für den jeweiligen Thread an. Die übrigen isolierten Kästchen innerhalb der virtuellen Registersätze sind die jeweiligen virtuellen Anzeigen- und Stackregister. Unter der allgemeinen Prozeßbeschreibung sollen Prozeßidentifikation, -priorität und weitere Einträge wie Benutzergruppen, Abrechnungsdaten usw. verstanden werden. Der gepunktete Pfeil zwischen dem virtuellen Adreßraum, der Interprozeßkommunikation des Betriebssystems und einer weiteren Prozeßbeschreibung (gestrichelt gezeichnet) symbolisiert den Informationsfluß des Datenaustauschs zwischen verschiedenen Prozessen.

Es wird hier zwischen den Begriffen Speichermodell eines Betriebssystems und Speichermodell eines Rechners logisch getrennt. Zum Speichermodell des Betriebssystems gehört der virtuelle Adreßraum eines Prozesses mit seinen Adreßumsetzungsmechanismen. Das Speichermodell des Rechners gibt an, wie der Arbeitsspeicher eines Rechners zu adressieren ist. Bei einem Betriebssystem, das speziell für einen Rechnertyp entworfen ist, z.B. MS-DOS, SNI-BS2000, MVS, VMS, WINDOWS 95, WINDOWS-NT usw., findet sich das Speichermodell des Rechners direkt im virtuellen Adreßraum wieder, jedoch wird meist von der realen, d.h. im konkreten Fall installierten Arbeitsspeichergröße abstrahiert. Betriebssysteme wie UNIX, die als Zielmaschinen viele unterschiedliche Rechnertypen vorsehen, haben ein eigenes virtuelles Speichermodell, das dann im jeweiligen Fall auf das Speichermodell der Zielmaschine umgesetzt werden muß.

Zusammenfassend ergibt sich für die **Art der Adressen in einem ablauffähigen Programm, wie sie von einem Compiler erzeugt werden** (Abbildung 3.2-3):

- Ist das Programm für den Einsatz unter Kontrolle eines Betriebssystems mit virtueller Adressierung bestimmt, sind die Adressen in einem übersetzten und gebundenen Programmodul in der Regel virtuelle Adressen. Nur in wenigen Systemprogrammen, die dann meist in der Assemblersprache des Zielrechners geschrieben sind, werden physikalische Arbeitsspeicheradressen direkt eingesetzt. Die Speicherverwaltung des Betriebssystems setzt die virtuellen Adressen prozeßspezifisch und tabellengesteuert während des Programmlaufs in physikalische Arbeitsspeicheradressen um

- In einer Systemumgebung, in der keine virtuellen Adressierungsmodelle verwendet werden, berücksichtigen die Programmadressen in der Regel immer noch das Speichermodell des Rechners. In diesem Fall kann man den virtuellen Adreßraum mit dem Speichermodell des Rechners gleichsetzen, so daß man auch hier von virtuellen Adressen in einem übersetzten und gebundenen Programmodul sprechen kann, die der Compiler erzeugt. Während der Ausführungsphase eines Programms findet dann eventuell noch eine hardwaremäßig unterstützte sehr einfache Adreßumsetzungen der Programmadressen auf physikalische Arbeitsspeicheradressen (ohne die Notwendigkeit von Adreßumsetzungstabellen) statt.

Selbst unter MS-DOS beispielsweise, das eine Speichersicht hat, die fast dem real installierten Arbeitsspeicher des Rechners entspricht, erfolgen noch während der Ausführungszeit Adreßumsetzungen von (implizit) zweiteiligen 16-Bit-Adressen, wie sie im Programm verwendet werden, auf physikalische 20-Bit-Arbeitsspeicheradressen.

Insgesamt kann man also alle Adressen in einem von einem Compiler einer höheren Programmiersprache erzeugten ausführbaren Programmodul als virtuelle Adressen ansehen, zumindest in ihrer Behandlungsweise virtuellen Adressen gleichbedeutend, die noch in physikalische Adressen umgesetzt werden müssen.

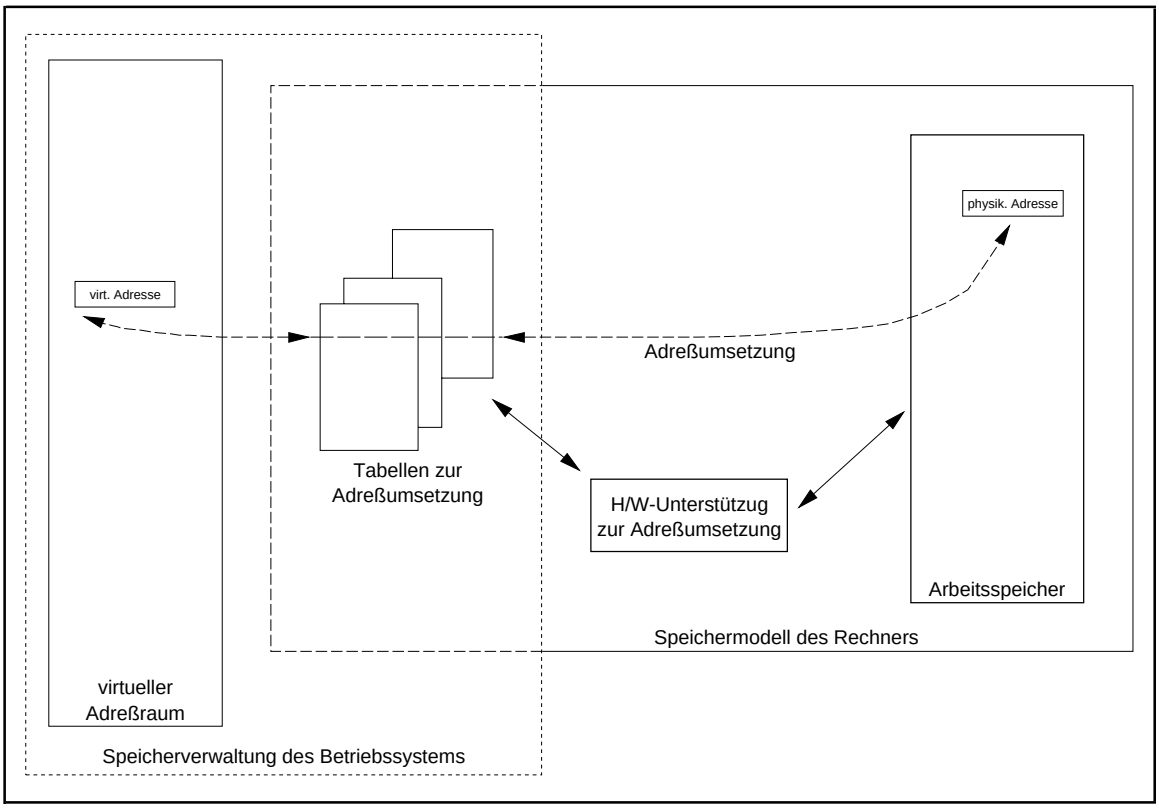


Abbildung 3.2-3: Speichermodell und Adreßumsetzung

3.3 Beispiele von Speichermodellen

Dieses Kapitel behandelt überblicksmäßig Beispiele von Speichermodellen, und zwar den Real Mode und den Protected Mode der INTEL 80x86-Prozessoren als Beispiele segmentierter Speichermodelle, und ein allgemeines Speichermodell, das in vielen Betriebssystemen verwendet wird: die virtuelle Adressierung mittels Paging. Letztere Methode wird in ähnlicher Form von den INTEL-Prozessoren unterstützt (Flat Mode) und findet in Betriebssystemen wie WINDOWS-NT, WINDOWS 95, OS/2, UNIX und vielen Großrechner-Betriebssytemen mit Multitasking-Konzept Anwendung und wird daher hier vom Rechnertyp unabhängig beschrieben.

3.3.1 Beispiel: Der Real Mode des INTEL 80x86

Auf das mit **Real Mode** bezeichnete Speichermodell des INTEL 80x86-Rechners stützt sich das Betriebssystem MS-DOS ([THI][8]).

Der Real Mode sieht einen Arbeitsspeicher mit 1 MB vor. Physikalische Arbeitsspeicheradressen sind also 20 Bits breit. In diesem physikalischen Adreßraum liegen (physikalisch ungetrennt) sowohl Teile des Betriebssystems als auch Anwenderprogramme. Ein Anwenderprogramm hat Zugriff auf alle Hauptspeicheradressen, einschließlich der Betriebssystemkomponenten (z.B. Interruptvektoren, Interrupt-Behandlungsroutinen, Gerätetreiber, Hardwareschnittstellen). Ein Speicherschutzmechanismus ist nicht vorgesehen. Ein wesentlicher Nachteil des Real Modes ist die Beschränkung des adressierbaren Arbeitsspeicherbereichs auf 1 Megabyte (1 MB), wobei Anwenderprogramme davon maximal 640 KB nutzen können, da die Adreßbereiche oberhalb der 640-KB-Grenze im wesentlichen zur Ansteuerung der Hardwarekomponenten fest vergeben sind. Das Betriebssystems MS-DOS unterstützt jedoch die Nutzung von mehr als 1 MB real vorhandenen Arbeitsspeicher (bis 16 MB) durch den Einsatz der Extended-Memory-Technik oder den LIM-Standard (eine Entwicklung der Firmen LOTUS, INTEL und Microsoft, auch EMS, expanded memory specification, genannt, siehe unten).

Das Arbeitsspeicherlayout in MS-DOS zur Laufzeit zeigt Abbildung 3.3.1-1.

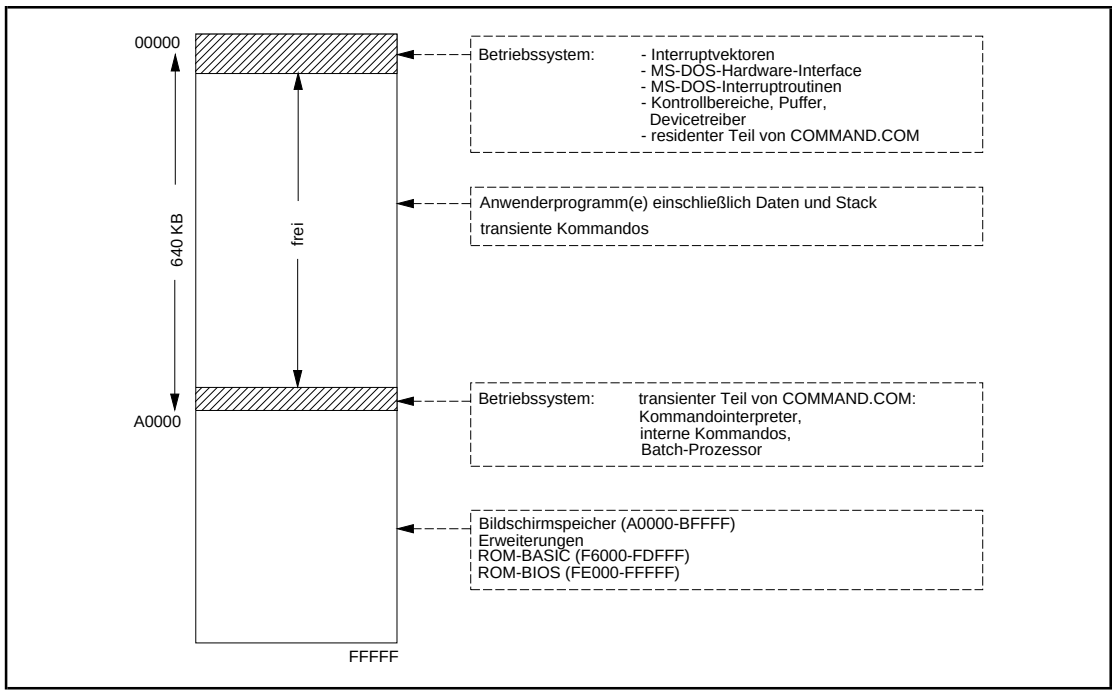


Abbildung 3.3.1-1: Arbeitsspeicherlayout in MS-DOS

[8] Vgl. auch Microsoft MS-DOS Operating System Programmer's Reference Manual, Markt&Technik, 1986.

Der Real Mode kennt zwei Typen von Lademoduln eines Programms: die COM-Dateien und die EXE-Dateien, die durch entsprechende Kennungen in den ersten beiden Bytes unterschieden werden. Eine **COM-Datei** enthält ein Programm einschließlich lokaler Daten, dessen sämtliche Adressen innerhalb von 64 KB liegen, und wird im folgenden nicht weiter betrachtet.

Eine **EXE-Datei** enthält ein Programm, das größer als 64 KB und im Arbeitsspeicher frei verschiebbar ist.

Ein Programm, das im Real Mode läuft, muß vollständig in den Arbeitsspeicher passen. Innerhalb eines von einem Compiler erzeugten Lademoduls sind Programmcode und Daten eines Programms logisch getrennt und in sogenannte **Segmente** fester Länge von jeweils 64 KB aufgeteilt. Man spricht von **Codesegmenten** und **Datensegmenten**. Ein Programm besteht meist aus mehreren Code- bzw. Datensegmenten. Zusätzlich werden zur Realisierung der Unterprogrammtechnik während der Laufzeit im Arbeitsspeicher **Stacksegmente** zugewiesen.

Jedes Objekt (Daten, Maschineninstruktion) eines Programms wird durch seine **Segmentnummer** relativ zum Lademodulanfang und seinen Abstand zum Anfang des Segments, den **Offset**, lokalisiert, d.h. eine komplette virtuelle Adresse eines Objekts besteht aus einer (16 Bits langen) Segmentnummer und einem (16 Bits langen) Offset. Im Programm wird eine Referenz auf ein Objekt entweder durch die Angabe beider Komponenten notiert (im folgenden wird dafür die Schreibweise [Segmentnummer, Offset] verwendet) oder nur durch die Angabe der Offset-Komponente (im folgenden durch [.,Offset] beschrieben). Im ersten Fall spricht man von einem **FAR-Pointer**, im zweiten Fall von einem **NEAR-Pointer**. Die einzelnen Maschinenbefehle verwenden zur Adressierung ihrer Operanden je nach Instruktionstyp FAR- oder NEAR-Pointer. Ein NEAR-Pointer kann nicht über eine Segmentgrenze hinweg verweisen. In Abbildung 3.3.1-2 ist das mit P bezeichnete Objekt in Segment i ein NEAR-Pointer und das mit Q bezeichnete Objekt ein FAR-Pointer in das Segment $j \neq i$. Einige Maschinenbefehle verwenden NEAR-Pointer, obwohl das referenzierte Objekt in einem anderen Segment, z.B. in einem Datensegment, liegt; in diesem Fall wird implizit vorausgesetzt, daß die zugehörige Segmentnummer zur Laufzeit vor Erreichen des Maschinenbefehls im Programmlauf bereits in ein definiertes Register (Segmentregister, siehe unten) geladen wurde.

Der Lademodul eines Programms (die EXE-Datei) enthält neben den Programm- und Datensegmenten eine variabel lange **Verschiebungstabelle (relocation table)**, in die die vollständige Adresse jedes FAR-Pointers innerhalb der Segmente (durch die Angabe von dessen Segmentnummer und Offset) eingetragen ist. Außerdem enthält die EXE-Datei am Anfang einen sogenannten Header von 28 Bytes, in dem Verwaltungsinformationen stehen, die während des Ladevorgangs des Programms benötigt werden. Beim Laden des Programms wird im Arbeitsspeicher ein freier Bereich auf möglichst kleiner durch 16 teilbaren Adresse ermittelt[9], der das Programm aufnehmen kann (die Programmlänge wird aus entsprechenden Headereinträgen errechnet), und die Segmente des Programms (ohne Verschiebungstabelle und Header) werden hintereinander, wie sie in der EXE-Datei liegen, in den Arbeitsspeicher geladen. Durch den Anfang dieses Bereichs ist die Nummer des **(physikalischen) Startsegments** des geladenen Programms im Arbeitsspeicher festgelegt.

[9] Jede durch 16 teilbare Arbeitsspeicheradresse ist eine potentielle Adresse für den Anfang eines Startsegments eines geladenen Programms.

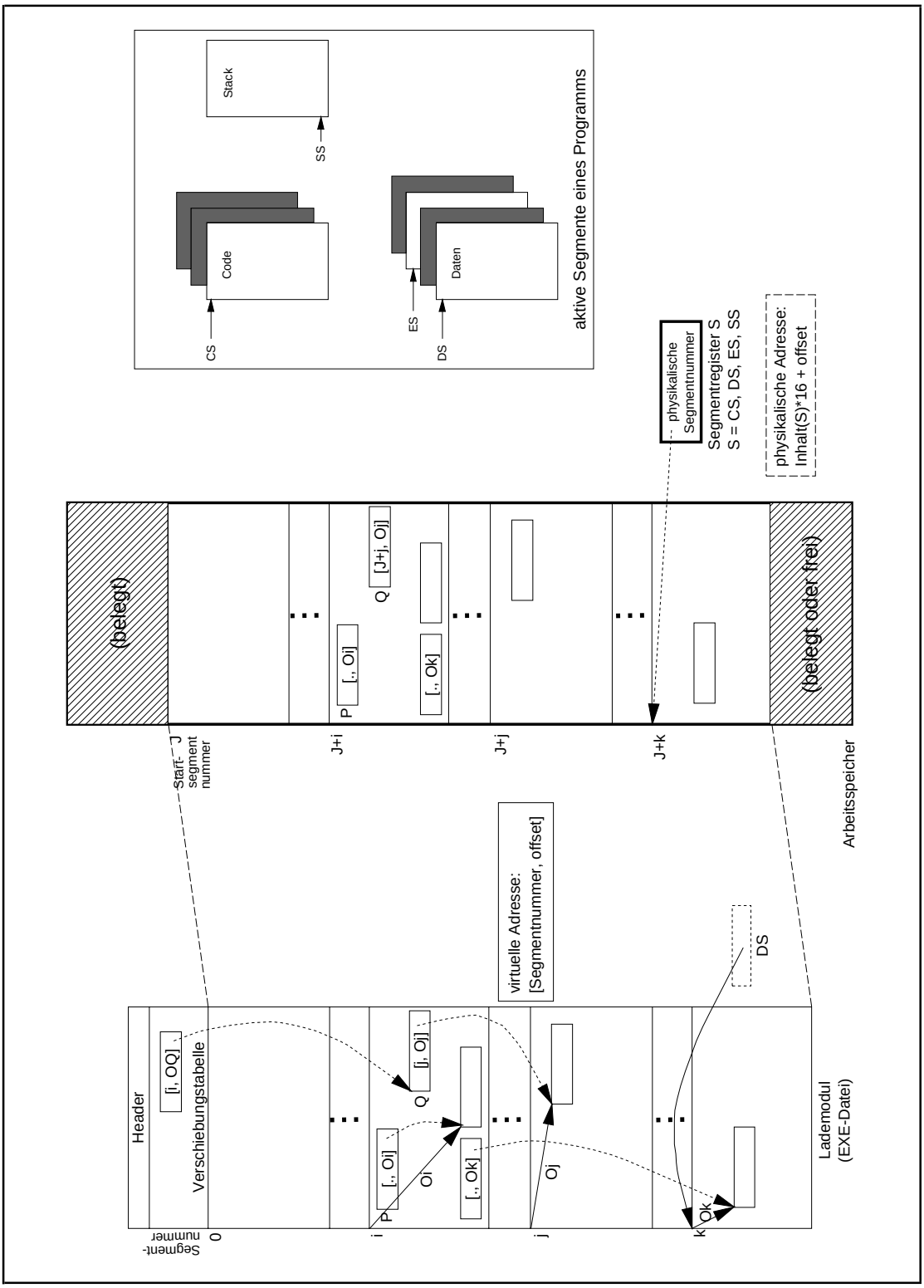


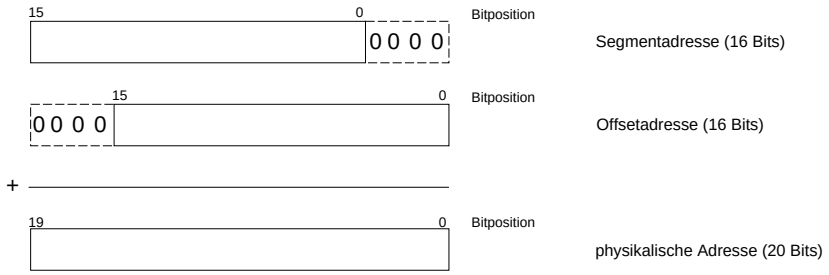
Abbildung 3.3.1-2: Adressierung im INTEL 80x86 Real Mode

Im Beispiel der Abbildung 3.3.1-2 ist die Startsegmentnummer die Nummer *J*. Jeder Eintrag [*seg*, *off*] der Verschiebungstabelle verweist innerhalb der EXE-Datei auf eine zu modifizierende Adresse: im geladenen Programm im Arbeitsspeicher steht das zu modifizierende Programm an der Stelle [*seg+J*, *off*]. Hat dieses Objekt bisher die Form [*j*, *Oj*], so wird es durch [*j+J*, *Oj*] ersetzt

(vgl. Abbildung 3.3.1-2). Auf diese Weise beziehen sich nun die modifizierten Segmentnummern im geladenen Programm nicht mehr auf die Anfangssegmentnummer 0 im Lademodul des Programms in der EXE-Datei, sondern auf die Segmentnummer 0 im Arbeitsspeicher. Ein geladenes Programm (Code und Daten) kann im Arbeitsspeicher nicht mehr verschoben werden, da nach dem Ladevorgang die Verschiebungstabelle, die die zu modifizierenden Objekte lokalisiert, nicht verfügbar ist.

Auch wenn ein Programm eventuell aus vielen Code- und Datensegmenten besteht, sind während der Laufzeit gleichzeitig nur vier Segmente aktiv, die durch die Inhalte der vier Segmentregister CS, DS, ES und SS der CPU (Abbildung 2.1-1) identifiziert werden:

CS Der Inhalt des CS-Registers (code segment) repräsentiert das gerade aktive **Codesegment**. Enthält das CS-Register einen Wert *ccs*, so ist *ccs*16* die Basisadresse (physikalische Anfangsadresse) des aktiven Codesegments. Der Inhalt des Register IP (instruction pointer) ist der offset des nächsten ausführbaren Befehls, der also in der CPU wie folgt ermittelt wird:



Das Ziel eines Sprungbefehls/Unterprogrammsprungs, das sich innerhalb desselben Codesegments befindet, wird entweder durch einen NEAR-Pointer identifiziert (NEAR-Call); in diesem Fall wird lediglich IP verändert. Oder der Sprungbefehl/Unterprogrammsprung verweist mittels eines FAR-Pointers über Segmentgrenzen hinweg (FAR-Call); dann werden CS und IP verändert.

DS Das DS-Register (data segment) repräsentiert ein gerade aktives **Datensegment**. Das DS-Register enthält eine Adresse *cds*; *cds*16* stellt die Basisadresse (physikalische Anfangsadresse) des Datensegments dar. Das Datensegment enthält die vom Programm zu manipulierenden Datenbereiche. Die Register SI und DI (und je nach Befehl auch das Register BX) bestimmen Offsets von Datenbereichen innerhalb des aktiven Datensegments.

ES Das ES-Register (extra segment) repräsentiert ebenfalls ein Datensegment. Es wird hauptsächlich dann eingesetzt, wenn Speicherzelleninhalte über einen Bereich hinweg kopiert werden sollen, der größer als 64 KB ist, und sich dadurch in unterschiedlichen Segmenten befinden. Vor allem die Stringbefehle verwenden das ES-Register.

SS Das SS-Register (stack segment) enthält eine Adresse *css*, die über die Formel *css*16* auf die Basisadresse (physikalische Anfangsadresse) des **Stacksegments** zeigt. Der Stack wird zur Speicherung der Rücksprungadresse bei Unterprogrammaufrufen, zum Zwischenspeichern von Daten und zur Parameterübergabe an Unterprogramme verwendet. Bei der Interrupt-Behandlung wird der Stack ebenfalls zur Zwischenspeicherung des aktuellen CS-

und IP-Werts (Rücksprungadresse) und des Anzeigenregisters FLAGS genutzt[10]. Der aktuelle Offset zum Stackanfang steht im Register SP; das Register BP kann ebenfalls einen Offsetwert bezüglich des Stacks enthalten.

Unter MS-DOS ist es wegen der Adressierungsbreite von 20 Bits nicht möglich, direkt mehr als 1 MB Arbeitsspeicher anzusprechen. Da zudem die meisten Adressen oberhalb von 640 KB reserviert sind, bleibt für Anwenderprogramme weniger als 640 KB (ca. 100 KB wird von MS-DOS belegt). Zwei Techniken erlauben es, Arbeitsspeicher jenseits der 1-MB-Grenze anzusprechen: Extended Memory und Expanded Memory Technik.

Mit **Extended Memory** wird die Fortführung des Arbeitsspeichers über die 1-MB-Grenze hinweg bezeichnet. Das BIOS (Teil des MS-DOS) stellt zwei Funktionen zur Verfügung, um mit dem Extended Memory zu arbeiten: INT $15 Funktion $88 stellt die Größe des installierten bzw. verfügbaren Extended Memories fest, INT $15 Funktion $87 kopiert Speicherbereiche einer Größe bis zu 64 KB über die 1-MB-Grenze hinweg. Um Adressen oberhalb von 1 MB anzusprechen, muß der Rechner kurzfristig vom Real Mode in den Protected Mode umschalten, in dem höhere Speicherbereiche (bis zu 16 MB beim 80286 bzw. bis 2 GB ab 80386) adressierbar sind. Bei Anwendung der Extended-Memory-Technik muß der Anwender selbst die notwendigen internen Systemtabellen (Deskriptortabellen) bereitstellen, da beide BIOS-Funktionsaufrufe den Protected Mode voraussetzen. Die beschriebenen Funktionsaufrufe sind zeitkritisch, da das Zurückschalten vom Protected in den Real Mode vergleichsweise lange dauert. Außerdem setzen sie mindestens einen INTEL 80286-Prozessor voraus.

Einige Hilfsprogramme, z.B. das Cache-Programm SMARTDRV oder virtuelle Plattensysteme, verwenden Extended Memory, so daß dann für den Anwender nicht die volle Speichererweiterung zur Verfügung steht.

Bei der **Expanded Memory Technik (EMS)** wird der Erweiterungsspeicher in Seiten (pages) der Länge 16 KB eingeteilt. Ein 64 KB großer Bereich unterhalb des 1-MB-Grenze dient als "Fenster", indem er 4 Seiten (die physikalisch im Erweiterungsspeicher nicht notwendig hintereinanderliegen) aufnimmt. Die Manipulation der im Fenster befindlichen Seiten beziehen sich dann logisch auf den Erweiterungsspeicher. Ein Treiber namens EMM (Expanded Memory Manager), der über den CONFIG.SYS-File geladen werden muß, sorgt für die Abbildung der Erweiterungsseiten auf das 64-KB-Fenster (z.B. Laden und Zurückschreiben der im Fenster befindlichen Seiten). Der EMM wird über INT $67 und verschiedene Funktionsnummern angesprochen.

3.3.2 Beispiel: Der Protected Mode des INTEL 80x86

Ein weiteres Speichermodell des INTEL 80x86-Prozessors, der **Protected Mode**, überwindet die 1 MB-Grenze bei der Adressierung und läßt Multiprocessing und Multiprogramming zu. Der Protected Mode stellt die "klassische" Variante der Adressierung durch Segmentierung dar ([TAN]).

[10] Das ist notwendig, da das CS- und das IP-Register mit der Anfangsadresse (Interruptvektor) der entsprechenden Interrupt-Behandlungsroutine geladen werden. Nach Abarbeitung der Interrupt-Behandlungsroutine werden die gesicherten Werte für CS, IP und FLAGS aus dem Stack wieder rekonstruiert (Befehl IRET).

Programme und Datenbereiche eines Prozesses sind im Speichermodell des Protected Modes wieder in Blöcke, dieses Mal von variabler Länge entsprechend der Programmstruktur (Programme, Prozeduren, Datenbereiche usw.), eingeteilt. Die Blöcke heißen wieder **Segmente**. Ein Objekt in einem Segment wird durch eine zweiteilige virtuelle Adresse angesprochen, die sich wieder aus einem als **Selektor** (16 Bits) und einem als **Offset** (32 Bits) bezeichneten Teil zusammensetzt. Der Selektor wird jetzt aber nicht als Basisadresse (-teil) interpretiert, sondern stellt einen *numerischen Index* in eine Tabelle dar, eine **Deskriptortabelle**, die vom Betriebssystem gepflegt werden muß und Beschreibungen von Code- und Datenbereichen enthält. Eine Deskriptortabelle kann auch als **Segmentbeschreibungstabelle** bezeichnet werden. Es gibt eine systemweite **globale Deskriptortabelle**, deren Einträge Segmente beschreiben, die pro Prozeß einen Datenbereich darstellen, der bei Prozeß- und Threadwechsel Sicherstellungsbereiche für Register ("virtuelle Register") bereitstellt (in Abbildung 3.3.2-1 mit TSS bezeichnet), bzw. in denen eventuell Code- und Datenteile liegen, die von allen Prozessen gemeinsam verwendet werden. Für jeden Prozeß gibt es weiterhin eine eigene **lokale Deskritortabelle**, die die Beschreibungen für Segmente aller Programme (Threads) dieses Prozesses enthält. Weitere Deskritortabellen gibt es für die Interruptroutinen. Abbildung 3.3.2-1 zeigt das Prinzip bei mehreren Prozessen mit lokalen Code-, Daten-, und Stacksegmenten und global verfügbaren Programmteilen; der Interruptteil ist nicht dargestellt.

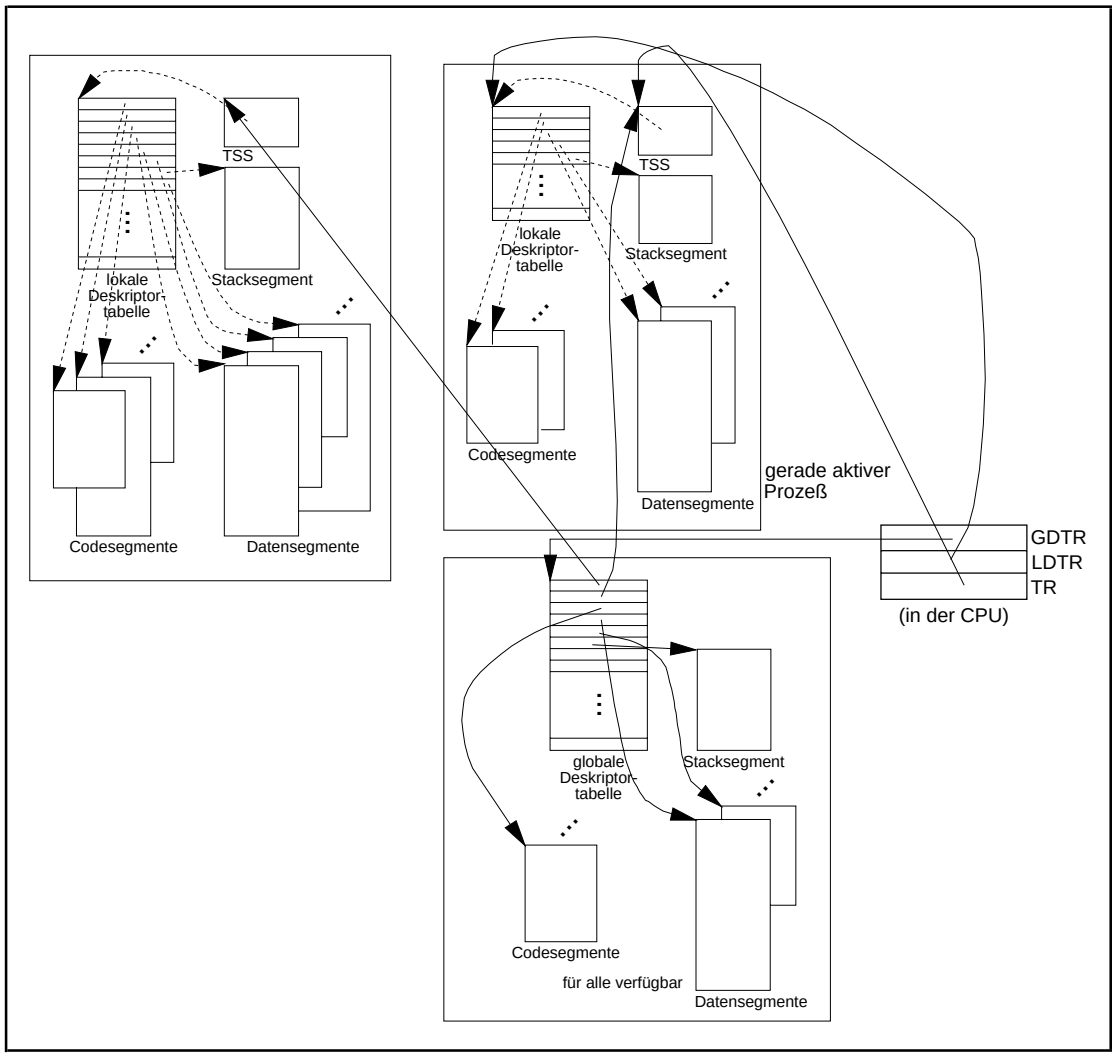


Abbildung 3.3.2-1: Segmente im INTEL Protected Mode

Pro Prozeß gibt es eine prozeßeigene Datenstruktur **Task Status Segment (TSS)**, in der der einzelne Prozeß beschrieben wird. Über die globale Deskriptortabelle ist jeder TSS erreichbar. Der TSS enthält u.a. eine Sicherstellungsbereich für die CPU-Registerinhalte bei Prozeßwechsel. Um ein zusätzliches Threadwechselmodell zu implementieren, sind prinzipiell zusätzliche Sicherstellungsbereiche, nämlich ein Bereich für jeden Thread erforderlich. Um hier Speicherplatz zu sparen und um den (häufig vorkommenden) Threadwechsel zu beschleunigen, lassen sich diese Sicherstellungsbereiche auch auf dem prozeßeigenen Stack (Kernel Stack im Systemmodus) einrichten, eine Technik, die beispielsweise das Betriebssystem OS/2 einsetzt.

Ein Eintrag in einer Deskriptortabelle heißt **Deskriptor** und enthält folgende Informationen:

- die **Basisadresse** (32 Bits), an der das Segment im Arbeitsspeicher geladen ist (falls es überhaupt geladen ist)
- die **Längenangabe** (20 Bits) des Segments
- **Attribute** des Segments (**Segmenttyp**), die Merkmale des virtuellen Schutzmechanismus beschreiben; so werden u.a. Datensegmente (mit nur lesbaren bzw. veränderbaren Daten) oder Programmsegmente (nur ausführbarer Code bzw. ausführbarer Code und lesbare Daten, z.B. Konstanten) unterschieden
- **Anzeigen**, die angeben, ob das Segment gerade im Arbeitsspeicher geladen ist oder nicht, und über die die Zugriffshäufigkeit auf das Segment vom Betriebssystem periodisch getestet werden kann
- weitere **Flags**, die über Privilegierungsstufen und über weitere Details der physikalischen Adressierung[11] Auskunft geben

Wird ein Segment in den Arbeitsspeicher geladen, so wird die Startadresse (Basisadresse, 32 Bits) mit den übrigen Einträgen im entsprechenden Deskriptor abgelegt. Wird dann ein Objekt in einem Segment über eine virtuelle Adresse angesprochen, so wird die Basisadresse des Segments mit Hilfe des Selektors über den zugehörigen Deskriptor ermittelt und der Offset des Objekts dazuaddiert (Abbildung 3.3.2-2). Falls bei dieser Adreßumsetzung festgestellt wird, daß sich ein benötigtes Segment nicht im Arbeitsspeicher befindet, wird ein Interrupt (**segment fault**) erzeugt, der das Betriebssystem zum Nachladen des angesprochenen Segments veranlaßt.

Die Adresse(n) der Deskriptortabelle(n) für den aktiven Prozeß ist in einem Hardwareregister, dem (den) **Segmentregister**(n) festgehalten. Aus Performancegründen bei der Adreßumsetzung von virtueller in reale Adresse werden außerdem die aktuellen Deskriptoren in der CPU in dafür vorgesehene Register, **Deskriptorregister**, abgelegt (siehe Abbildung 2.1-1); bei einem Thread- oder Prozeßwechsel müssen diese Register dann umgeladen werden.

Insgesamt ist es bei diesem Konzept möglich, Segmente erst dann in den Arbeitsspeicher zu laden, wenn auf sie auch wirklich zugegriffen werden soll. Außerdem können Segmente im Arbeitsspeicher verschoben oder auf Plattenspeicher ausgelagert werden. Allerdings sind in diesen Fällen komplexe Operationen zur Pflege der Deskriptortabellen erforderlich, auch wenn die Hardware in Form spezieller Maschinenbefehle Unterstützung bietet. Insgesamt ist dieses Verfahren stark an eine Hardwarearchitektur angelehnt.

---

[11] Es wird beispielsweise festgehalten (Granulität), ob die Längenangabe in Bytes oder in Seiten mit jeweils 4KB zu interpretieren ist. Bei der Maßeinheit Byte kann die Längenangabe bei den zur Verfügung stehenden 20 Bits zwischen 1 Byte und 2^{20} = 1 MB liegen; bei der Maßeinheit Seite sind bis zu $2^{20} \cdot 2^{12} = 2^{32}$ = 4 GB möglich.

Als Besonderheit des Protected Modes in der INTEL 80x86-Architektur kann die Möglichkeit angesehen werden, der hier beschriebenen Segmentierung das im Kapitel 3.3.3 beschriebene Paging-Verfahren zusätzlich nachzuschalten. Das bedeutet, daß die ermittelte 32 Bits lange Adresse nicht als reale Arbeitsspeicheradresse, sondern weiterhin als virtuelle Adresse interpretiert wird. Diese Adresse wird nun dem Paging unterworfen und dabei erst eine reale Arbeitsspeicheradresse bestimmt.

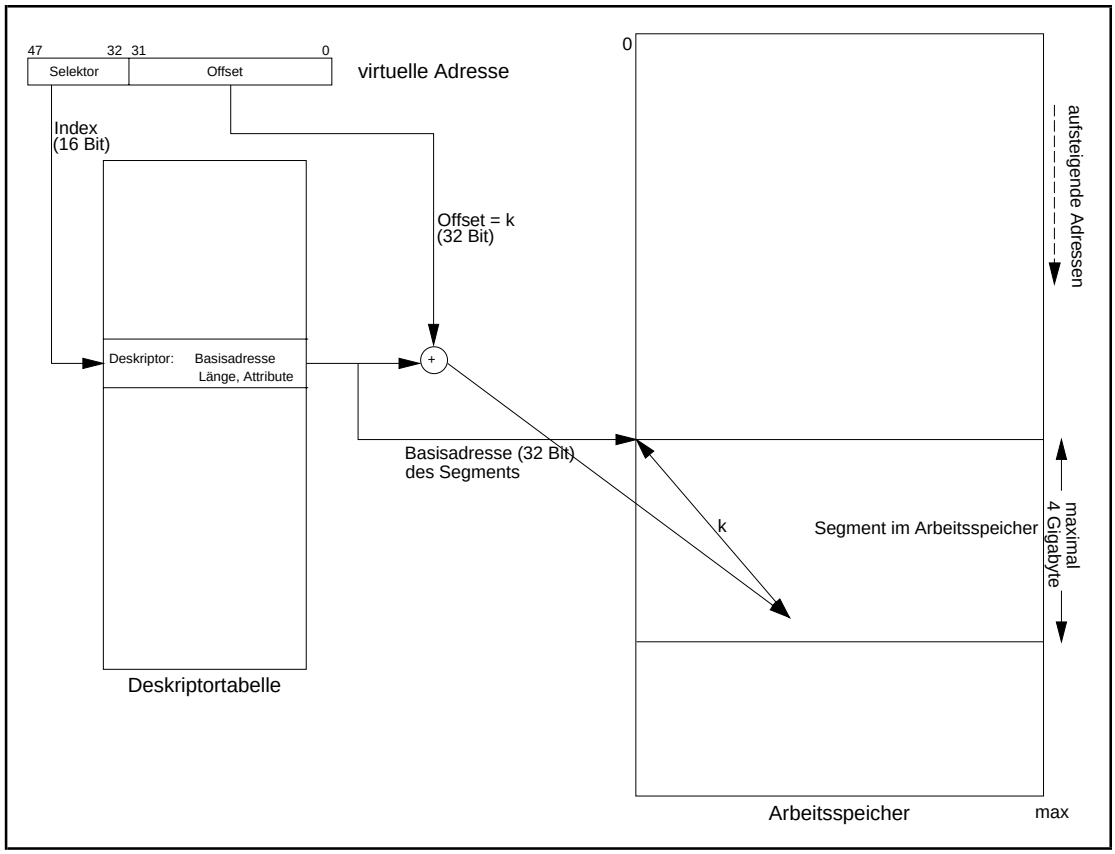


Abbildung 3.3.2-2: Adressierung im Protected Mode INTEL 80x86

3.3.3 Beispiel: Virtuelle Adressierung mittels Paging

Der virtuelle Adreßraum jedes Prozesses wird in **Seiten (pages)** gleicher fester Länge eingeteilt. Ebenso wird der Arbeitsspeicher in Blöcke derselben Länge in **Seitenrahmen** (**page frames**) eingeteilt. ***Diese Aufteilung des virtuellen Adreßraums in Seiten berücksichtigt nicht die Programmstruktur der im virtuellen Adreßraum liegenden Programme*** und ist dadurch programmunabhängig. Bei Numerierung der Bytes innerhalb einer Seite und der Seiten innerhalb gesamten virtuellen Adreßraums jeweils bei 0 beginnend kann man sich eine virtuelle Adresse in zwei Teile aufgeteilt denken: die Seitennummer (innerhalb des virtuellen Adreßraums) und den Offset (innerhalb der Seite).

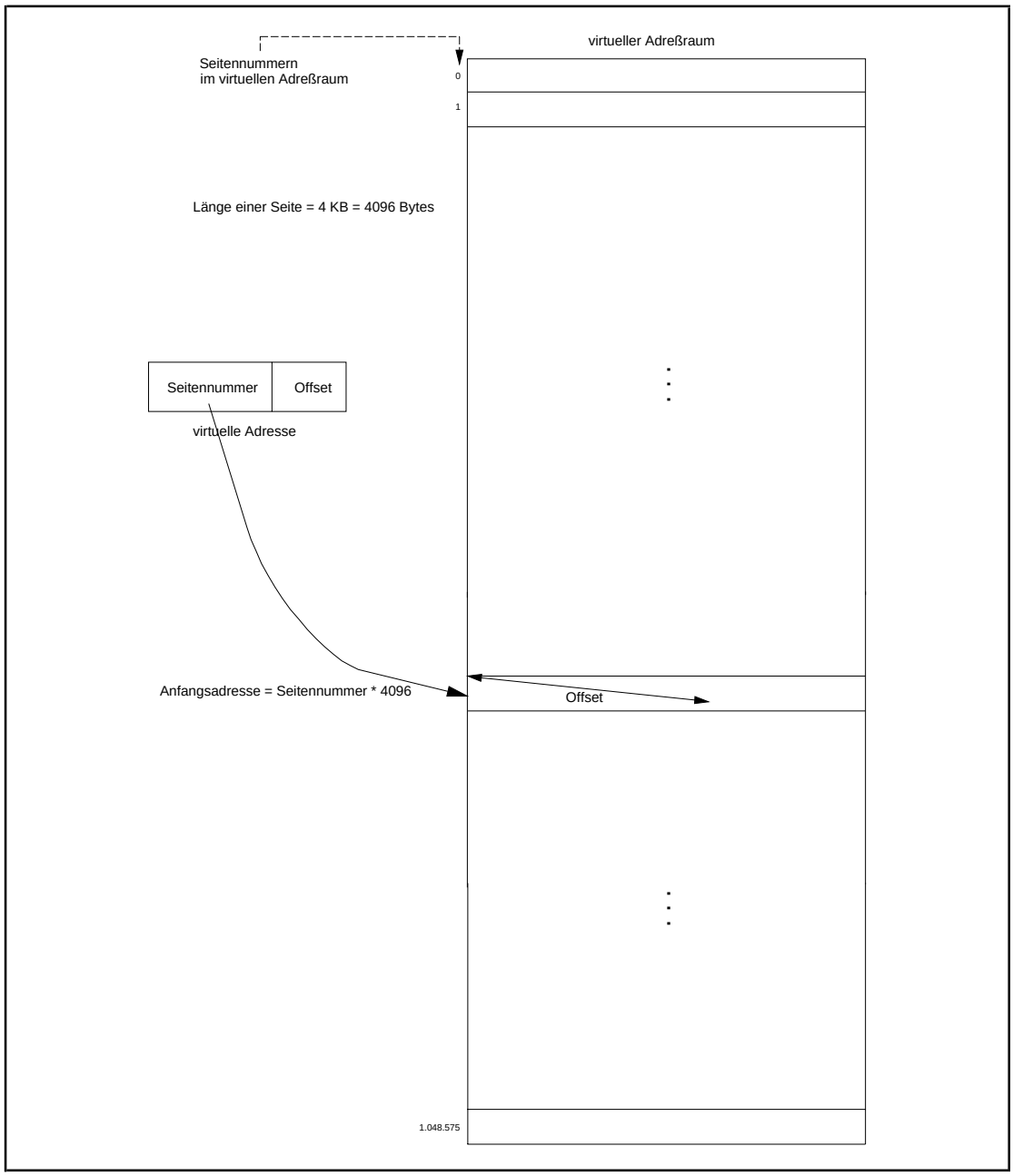


Abbildung 3.3.3-1: Virtuelle Adressierung mittels Paging (Zahlenangaben exemplarisch)

Eine Seite des virtuellen Adreßraums eines Prozesses wird dann im Arbeitsspeicher benötigt, wenn ein auszuführender Maschinenbefehl eine (virtuelle) Adresse innerhalb der Seite anspricht oder wenn ein weiterer Befehl für den Prozeß ausgeführt werden soll. Befindet sich die entsprechende Seite noch nicht im Arbeitsspeicher, so wird sie in einen freien Seitenrahmen geladen, indem folgende Schritte durchgeführt werden (**demand paging**):

- Es wird ein Interrupt (**page fault, missing item fault**) erzeugt, der das laufende Programm unterbricht und die Kontrolle an das Betriebssystem übergibt.

- Die Speicherverwaltung lädt die Seite, die die angesprochene Referenz enthält, in einen freien Seitenrahmen des Arbeitsspeichers. Falls alle Seitenrahmen belegt sind, wird nach einer festgelegten **Seitenersetzungsstrategie** eine Seite aus einem Seitenrahmen ausgelagert (verdrängt).

Jeder virtuellen Adresse einer Seite ist jeweils eine reale Adresse zugeordnet, sobald die Seite in einen Seitenrahmen geladen wurde. Im Laufe der Prozeßlebensdauer kann eine Seite mehrmals in einen Seitenrahmen geladen und auch wieder aus einem Seitenrahmen verdrängt werden. Beim Nachladen einer Seite wird i.a. nicht derselbe Seitenrahmen wie beim vorherigen Laden verwendet, sondern irgendein freier Seitenrahmen. Die Umrechnung von virtueller auf reale Adresse eines Objekts in einer Seite, die in einem Seitenrahmen geladenen ist, erfolgt dann beim Zugriff auf das Objekt. Eine geladene Seite kann, wenn sie nicht mehr benötigt wird, durch eine Seite z.B. eines anderen Anwenders ersetzt werden. Auf diese Weise brauchen jeweils nur die Seiten in Seitenrahmen geladen zu werden, die man gerade benötigt. Im allgemeinen liegen virtuell zusammenhängende Seiten nicht in zusammenhängenden Seitenrahmen, und auch der gesamte virtuelle Adreßraum eines Anwenders ist nicht ständig komplett auf den Arbeitsspeicher abgebildet.

Jeder Zugriff auf eine angesprochene Referenz erfordert die Umsetzung der virtuellen Adresse in eine reale Arbeitsspeicheradresse. Dazu werden für jeden Prozeß vom Betriebssystem **Seitentabellen** verwaltet, die u.a. anzeigen, in welchen Seitenrahmen des Arbeitsspeichers eine Seite des virtuellen Adreßraums geladen ist. Außerdem werden in den Seitentabellen Seitenattribute des (virtuellen) Speicherschutzes, aktuelle Verfügbarkeit der Seite, Zugriffs- und Änderungsanzeigen bzgl. der Seite usw. festgehalten. Zur hinreichend schnellen Umsetzung einer virtuellen in eine reale Adresse muß der Prozessor das Verfahren unterstützen.

In den meisten Implementierungen des Pagings, wie z.B. beim Flat Mode der INTEL 80x86-Architektur, umfaßt eine Seite 2^{12} Bytes = 4 KB. Bei einem angenommenen virtuellen Adreßraum der Größe 2^{32} Bytes = 4 GB enthält der virtuelle Adreßraum eines Prozesses also 2^{20} = 1 MB viele Seiten, d.h. die Darstellung einer Seitennummer erfordert 20 Bits. Um die virtuelle *Anfangsadresse einer Seite* zu ermitteln, braucht man die Seitennummer nur mit der Seitenlänge, nämlich 2^{12}, zu multiplizieren bzw. die Bits in der Binärdarstellung der Seitennummer um 12 Stellen nach links zu verschieben. Die Anzahl an Bits, die benötigt werden, um in einem Eintrag der Seitentabelle eine Seitenrahmennummer festzuhalten, hängt eigentlich von der Größe des real vorhandenen Arbeitsspeichers (genauer: der Anzahl der vorhandenen Seitenrahmen) ab. Es wird aber unabhängig vom jeweiligen Rechner für die Seitenrahmennummer eine feste Anzahl an Bits vorgesehen. Geht man von 32 Bits einschließlich der Bits für die Seitenattribute aus, so würde eine einzige Seitentabelle pro Prozeß allein einen 4 MB großen zusammenhängenden resident im Arbeitsspeicher liegenden Speicherblock belegen[12].

[12] 2^{20} Einträge der Größe 32 Bits (= 4 Bytes) ergibt 2^{22} Bytes = 4 MB.

Diesem Kapazitätsproblem wird durch ein zweistufiges Pagingverfahren begegnet: Jeweils eine feste Anzahl hintereinanderliegender Seiten im virtuellen Adreßraum eines Prozesses werden zu einer "Verwaltungseinheit" zusammengefaßt. Entsprechend wird die Seitentabelle in soviele kleinere Seitentabellen aufgeteilt, wie es Verwaltungseinheiten gibt. Die Bits zur Darstellung einer Seitennummer werden in zwei Teile gegliedert: der erste Teil identifiziert eine so definierte Verwaltungseinheit, der zweite Teil identifiziert eine Seite innerhalb der Verwaltungseinheit. Es wird ein **Seitenverzeichnis (page directory)** angelegt, das soviele Einträge enthält, wie es Verwaltungseinheiten gibt. Ein Eintrag im Seitenverzeichnis identifiziert diejenige Seitentabelle, die zu den Seiten der Verwaltungseinheit gehören.

In obigem Beispiel werden jeweils $2^{10} = 1024$ im virtuellen Adreßraum eines Prozesses hintereinanderliegende Seiten zu einer Verwaltungseinheit zusammengefaßt, so daß es 2^{10} Verwaltungseinheiten gibt. Die führenden 10 Bits der Seitennummer werden zur Numerierung der Verwaltungseinheiten verwendet, die anderen 10 Bits für die Seitennummer innerhalb der Verwaltungseinheit.

Ein Eintrag im Seitenverzeichnis zeigt auf die Basisadresse der Seitentabelle, die die Seiten dieser Verwaltungseinheit beschreibt, und umfaßt 32 Bits = 4 Bytes. Da alle Seitentabellen auf einer 4-KB-Grenze beginnen, brauchen für die Anfangsadresse wieder nur die höherwertigen 20 Bits festgehalten zu werden (die unteren 12 Bits sind gleich 0_2), so daß Platz für die Speicherung von Attributen bleibt, die sich auf die gesamte Verwaltungseinheit beziehen. Das Seitenverzeichnis belegt also $2^{10} \cdot 4 = 2^{12} = 4$ KB. Ein Eintrag in einer Seitentabelle belegt ebenfalls 32 Bits, so daß bei 2^{10} Seiten pro Verwaltungseinheit für jede Seitentabelle 4 KB benötigt werden. Während das Seitenverzeichnis eines Prozesses resident im Arbeitsspeicher liegt, werden die Seitentabellen selbst dem Paging unterworfen.

Jeder Prozeß besitzt also ein Seitenverzeichnis (resident im Arbeitsspeicher) und eine Menge von Seitentabellen, die ein- und ausgelagert werden. Die Anfangsadresse des Seitenverzeichnisses für den gerade aktiven Prozesses wird in einem dafür vorgesehenen CPU-Register festgehalten (bei der INTEL 80x86-Architektur ist es das Kontrollregister CR3). Den Umsetzmechanismus von einer virtuellen Adresse in eine Arbeitsspeicheradresse zeigt Abbildung 3.3.3-2.

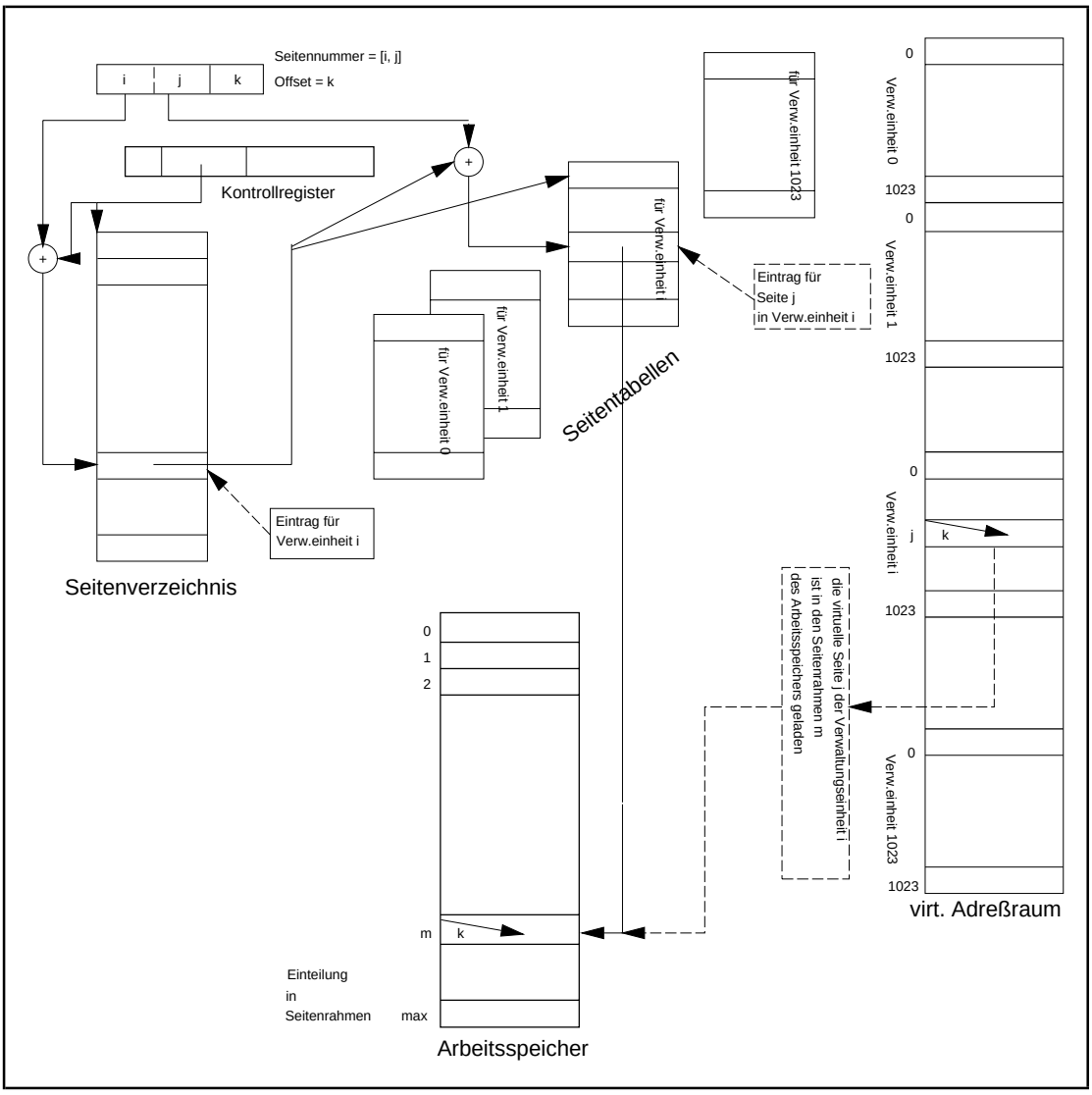


Abbildung 3.3.3-2: Implementierungsprinzip des Pagings (Zahlenangaben exemplarisch)

3.4 Programme in einer höheren Programmiersprache und das Laufzeitlayout

Ein Programm setzt sich auf der HOL-Ebene im allgemeinen aus einem **Deklarationsteil (Definitionsteil)** und einem **Anweisungsteil (prozeduraler Teil, Befehlsteil)** zusammen (Abbildung 3.4-1).

Im Deklarationsteil werden alle Datentypen und Variablen (Datenobjekte) definiert, die das Programm verwendet, d.h. (bei Variablen) die es mit Anfangswerten versieht, inhaltlich verändert oder bzgl. ihres Inhalts abfragt. Außerdem enthält der Deklarationsteil Konstantendefinitionen; dieses sind Festlegungen von Bezeichnern mit korrespondierenden Werten, die dann im folgenden mit diesem Wert durch Nennung des Bezeichners verwendet werden können. Außerdem werden hier die internen Unterprogramme (bei Pascal in Form von

Prozeduren und Funktionen) deklariert. Der Anweisungsteil eines Programms beschreibt die Aktionen, die vom Programm mit den im Deklarationsteil definierten Daten und Prozeduren vorgenommen werden sollen. Weitere Details beschreibt Kapitel 6.

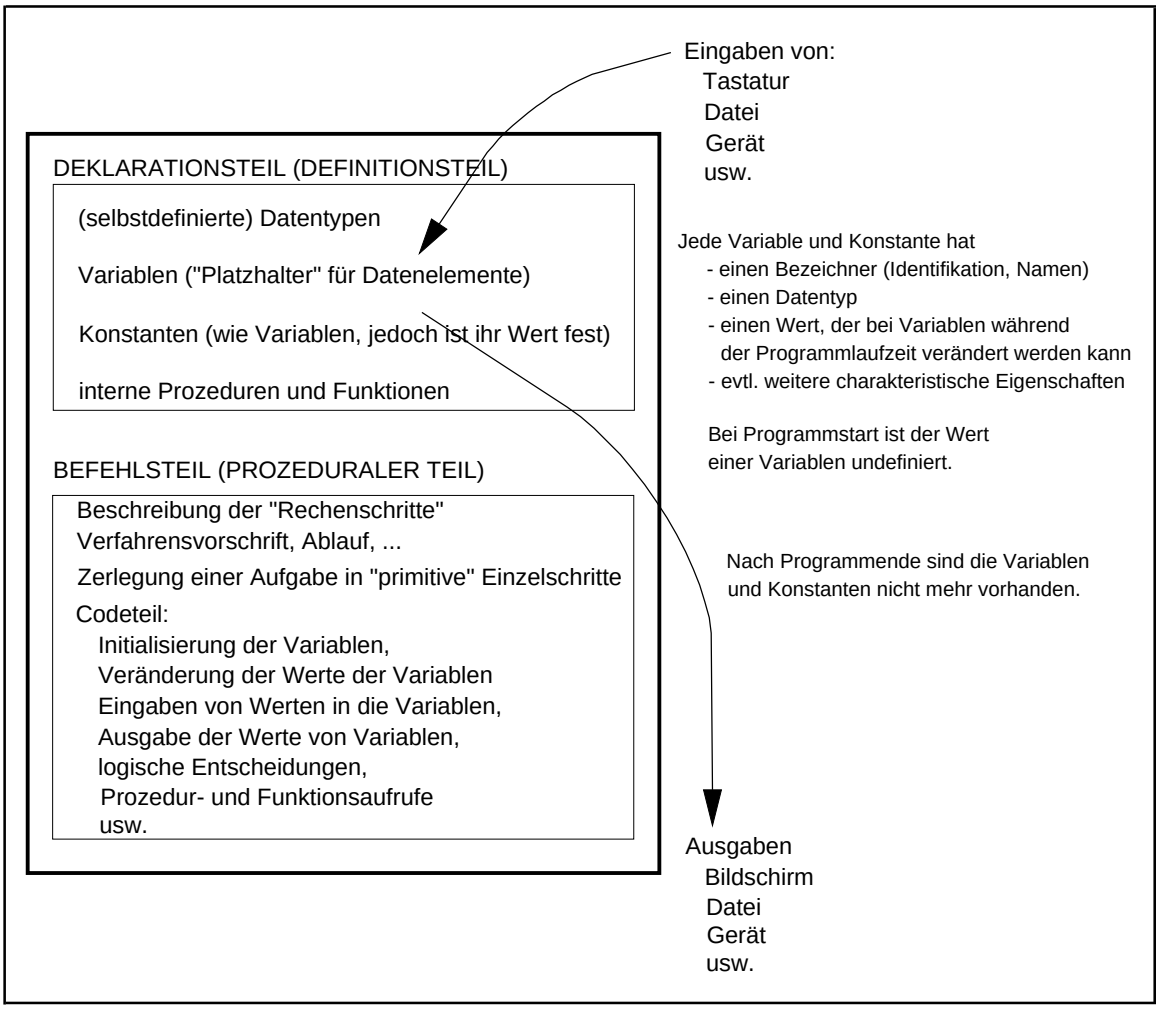


Abbildung 3.4-1: Programmaufbau in einer höheren Programmiersprache

Die hier beschriebenen allgemeinen Regeln über den Programmaufbau werden gelegentlich "abgemildert". Beispielsweise können interne Unterprogramme in C auch nach dem Befehlsteil stehen, jedoch muß das Aufrufformat im Deklarationsteil vor dem Befehlsteil, in dem das Unterprogramm verwendet werden soll, aufgeführt werden. Eine Trennung von Aufrufformat und Code eines Unterprogramms findet man auch im Unit-Konzept von PASCAL.

Zu beachten ist, daß (prinzipiell) über die Programmvariablen in einer höheren Programmiersprache in keiner Weise Bezug auf einzelne Register der CPU oder Arbeitsspeicherzellen genommen werden kann. Diese hardwaremäßigen Details sind für ein Programm in einer höheren Programmiersprache nicht sichtbar; ***es verwendet ausschließlich die im Deklarationsteil festgelegten Datenobjekte, und zwar benannt durch die dort definierten Bezeichner.*** Die später beschriebene Einbeziehung von Bezeichnern bei der PASCAL-Programmierung, die in externen Prozeduren, genauer in selbständigen Units,

definiert sind, ordnet sich in das Konzept ein. In C kann man bei der Definition einer Variablen den Compiler anweisen, eine Variable auf ein Register abzubilden (Speicherklasse register); dabei wird nicht festgelegt, um welches Register es sich handelt, und eine Manipulation des Registerinhalts ist nur indirekt über die Variable möglich.

Üblicherweise wird bei der Programmierung eine strikte Trennung zwischen Daten- und Anweisungsteil eingehalten, die von der Syntax der verwendeten Programmiersprache erzwungen wird. Diese Trennung spiegelt sich dann auch meist im Ergebnis der Übersetzung wider, d.h. es sind immer noch Teile, die ausführbaren Programmcode enthalten, strikt getrennt von Teilen, die ausschließlich aus Datenbereichen bestehen. Als Konsequenz ergibt sich, daß bei diesem Konzept ***die Verwendung selbstmodifizierender Programme ausgeschlossen wird***. Dabei handelt es sich um Programme, die einen anderen Programmteil und sogar den eigenen Code während der Ausführung als Daten behandeln und diesen vor seiner Ausführung modifizieren, eine in den Frühtagen der Betriebssystementwicklung gelegentlich zur Codereduktion eingesetzte Programmiertechnik. Auf HOL-Ebene sollte man diese Praxis vermeiden; sie wird i.a. von der Programmiersprache auch nicht unterstützt.

Bemerkung: Im Protected Mode des INTEL 80x86 wird eine Trennung zwischen Code- und Datenteilen (Segmenten) durch das Setzen entprechender Segmentattribute erreicht. Der Inhalt eines Datensegments darf nicht ohne weiteres als Code ausgeführt werden. Es ist jedoch möglich, auf ein Datensegment über einen sogenannten Aliascodeselektor so zuzugreifen, als enthalte es ausführbaren Code. Entsprechendes gilt für ein Codesegment, dessen Inhalt nicht wie ein Datensegment gelesen bzw. verändert werden darf. Auch hier verschafft ein Aliascodeselektor Abhilfe (vgl. [BAU]).

Nach der Übersetzung sind die Variablen je nach Typ und Stellung innerhalb des Programms eventuell in Speicherplatzreservierungen und alle Namensbezüge auf die Variablen in Adreßbezüge (virtuelle Adressen) umgesetzt worden. Für einige Variablen, z.B. die lokalen Variablen in Unterprogrammen und durch Pointer referenzierte Daten (siehe Kapitel 4), erfolgt eine Speicherplatzreservierung erst zur Laufzeit gemäß eines sogenannten Lebensdauerkonzepts. Im folgenden wird das **Laufzeit-Layout eines Pascal-Programms** beschrieben[13]. Programmiersprachen wie C oder C++, allgemein sogenannte **stackorientierte Programmiersprachen** (siehe [G/J]), haben ein ähnliches Laufzeit-Layout. Zur Vereinfachung wird angenommen, daß alle Programmteile statisch gebunden, d.h. daß also alle externen Referenzen durch Anbinden entsprechender Programm- und Datenteile aufgelöst wurden. Bei dynamischem Binden ergibt sich ein ähnliches Bild; allerdings sind Code- und statischer Datenbereich dann "noch nicht vollständig".

In einem Pascal-Programm kommen Datendeklarationen und die daraus erzeugten Datenobjekte an unterschiedlichen Positionen vor: im Hauptprogramm, in Unterprogrammen als Formalparameter und als lokale Daten und als dynamisch erzeugte Daten in allen Programmteilen. Je nach dem Ort dieser Deklaration im Quellprogramm finden sich die entsprechenden Datenreservierungen im Laufzeit-Layout an unterschiedlichen Stellen. Aus

[13] Die Definition der Sprache Pascal macht natürlich keine Angaben über das Laufzeit-Layout. Dieses ist ein Thema der Realisierung der Sprache für einen speziellen Rechnertyp, d.h. ein Aspekt des Compilerbaus.

Programmiersicht auf HOL-Ebene sind die Orte der Datenreservierungen im übersetzten Programm weitgehend von untergeordnetem Interesse. Daher benötigt der Anwender auf Pascal-Sprachebene, etwa durch Angaben von Speicherklassen für Variablen wie in C oder C++, keinen expliziten Einfluß auf deren Lage.

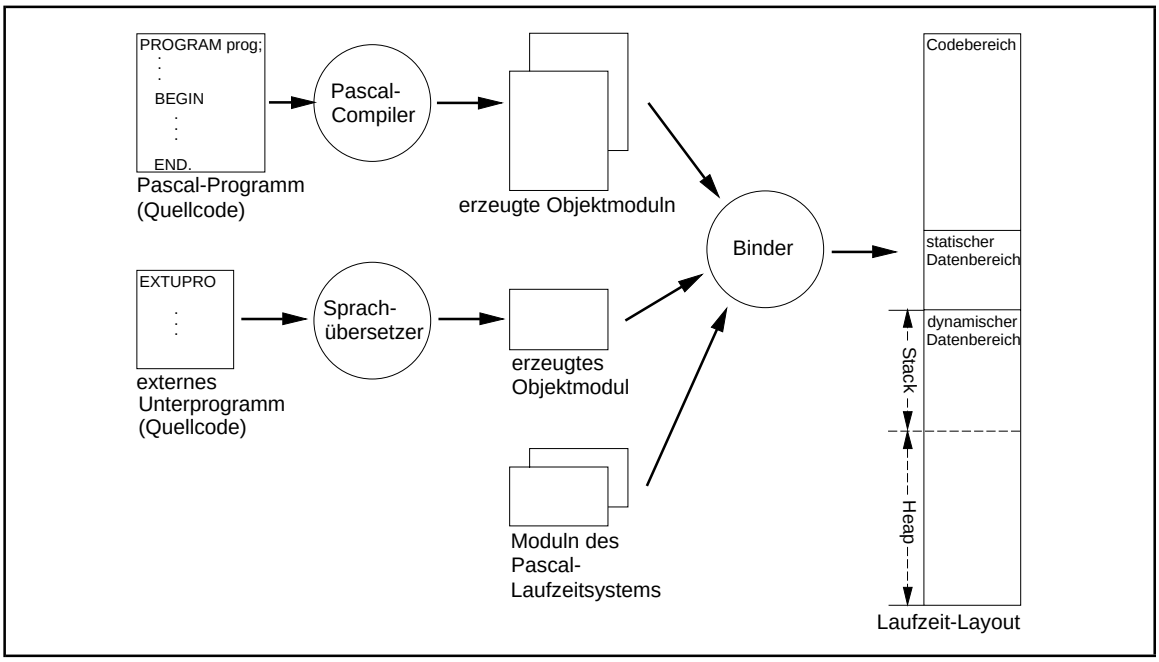


Abbildung 3.4-2: Laufzeit-Layout bei der Pascal-Programmierung

Das typische Laufzeit-Layout des virtuellen Adreßraums bei der Pascal-Programmierung ist in Abbildung 3.4-2 dargestellt. Ein Pascal-Programm mit Namen `prog`, das interne Unterprogramme, Datendefinitionen und den Aufruf eines externen Unterprogramms `EXTUPRO` enthält, wird vom Pascal-Compiler übersetzt. Das externe Unterprogramm, das in einer anderen Programmiersprache geschrieben sein kann, wird ebenfalls mit dem erforderlichen Sprachübersetzer übersetzt. Der Binder bindet neben den erzeugten Objektmoduln weitere Moduln des Laufzeitsystems von Pascal an.

Das ablauffähige Programm besteht aus folgenden unterscheidbaren Bereichen:

- Der **Codebereich** enthält alle ausführbaren Anweisungen des Programms `prog` und aller internen Prozeduren (aus `prog`) und externen Prozeduren (`EXTUPRO` und Prozeduren des Laufzeitsystems). Ausserdem sind alle deklarierten Konstanten in den Code eingearbeitet (ein Pascal-Compiler verarbeitet üblicherweise Konstantendefinitionen, indem er sie an den entsprechenden Stellen direkt in den Code einsetzt).

- Der **statische Datenbereich** enthält alle im Hauptprogramm `prog` definierten Variablen, sofern ihr Speicherplatzbedarf bereits zur Übersetzungszeit bekannt ist. Über Pointer definierte (dynamische) Variablen gehören nicht dazu, wohl aber die Variablen, die die Pointer aufnehmen, die auf derartige Daten verweisen (sofern sie im Hauptprogramm definiert sind). Sogenannte typisierte Konstanten des Hauptprogramms, d.h.

Konstanten-Deklarationen, die mit Anfangswerten initialisiert werden und z.B. in Borland Pascal erlaubt sind, liegen ebenfalls im statischen Datenbereich, da sie wie Variablen behandelt werden.

Der **dynamische Datenbereich** ist in zwei Teile unterteilt: in den **Stack** und den **Heap**. Die Gesamtgröße beider Bereiche läßt sich durch Compileroptionen festlegen; das Größenverhältnis beider Teilbereiche zueinander verändert sich dynamisch während der Laufzeit. Dabei wächst im allgemeinen der Heap von niedrigen zu hohen Adressen und der Stack in umgekehrter Richtung. In einigen Implementationen wird die Richtung umgedreht. Falls sich die Bereichsgrenzen während der Laufzeit überschneiden sollten, kommt es zu einem Laufzeitfehler mit Programmabbruch (**heap overflow** bzw. **stack overflow**). Heap und Stack sind komplett unterschiedlich organisiert und werden durch Routinen des Laufzeitsystems verwaltet.

Der **Heap** enthält alle während der Laufzeit durch die Pascal-Standardprozedur `New` eingerichtete Variablen, die über Zeiger (Pointer) angesprochen werden.

Der **Stack** dient der Aufnahme aller lokalen Variablen einer Prozedur: Dazu gehören im wesentlichen alle Variablen, die innerhalb der Prozedur deklariert werden. Variablen, die in einer eingebetteten Prozedur deklariert werden, sind lokal zu dieser eingebetteten Prozedur.

Eine Prozedur wird im allgemeinen von mehreren Stellen aus aufgerufen, wobei ihr vom Aufrufer jeweils wertmäßig unterschiedliche Sätze an Daten übergeben werden. In Abhängigkeit von diesen übergebenen Werten ermittelt die Prozedur "Rückgabewerte" und gibt sie an den Aufrufer zurück (der Mechanismus der Parameterübergabe ist Thema des Kapitels 5). Zur Aufnahme dieser zum jeweiligen Aufruf gehörenden aktuellen Werte (**Aktualparameter, aktuelle Parameter**) sind Variablen vorgesehen, die zu den lokalen Variablen der Prozedur zählen. Sie heißen **Formalparameter (formale Parameter)** der Prozedur und liegen ebenfalls im Stack. Die Liste der Formalparameter einer Pascal-Prozedur wird im Prozedurkopf spezifiziert.

Außerdem werden im Stack bei einem Unterprogrammsprung die Rücksprungadresse zum Aufrufer abgelegt und Sicherstellungsbereiche für Registerinhalte eingerichtet. Eventuell erforderliche Speicherplätze für Zwischenergebnisse bei arithmetischen Operationen können ebenfalls im Stack liegen. Ein Stackeintrag für eine Prozedur, d.h. die Reservierung der beschriebenen Daten, erfolgt erst, wenn die Prozedur während der Laufzeit auch wirklich aufgerufen wird.

Das hier beschriebene Layout zur Laufzeit ist nur als prinzipiell und beispielhaft zu betrachten. Je nach Sprachdialekt, eingesetztem Betriebssystem und Speichermodell des verwendeten Rechners gibt es Abweichungen (z.B. die relative Lage der einzelnen Bereiche zueinander).

4 Datenobjekte

Grundlage einer höheren Programmiersprache ist das darin enthaltene **Datenkonzept**. Programmiersprachen wie Pascal, C++, Ada oder Java verfügen über ein ausgeprägtes **Typkonzept**, das die Verwendung selbstdefinierter Datentypen erlaubt, andererseits aber die Regeln der Typisierung einzuhalten erzwingt. Eine Umgehung des Typkonzepts, wie es in COBOL oder C durch Überlagerung von Datendefinitionen üblich ist, erhöht nicht die Flexibilität, wohl aber die Fehleranfälligkeit der Programmierung. Die Zuverlässigkeit eines Programms wird im allgemeinen erhöht, indem der Compiler die Kompatibilität verwendeter Daten, die Einhaltung von Feldgrenzen oder Wertebereichen schon zur Übersetzungszeit des Programms prüft. Andererseits gibt es Situationen in der systemnahen Programmierung, in denen die Einhaltung eines stringenten Typkonzepts problematisch erscheint. Hier haben sich beispielsweise die semantischen Spracherweiterungen des Pointerkonzepts, die ein Dialekt wie Borland Pascal gegenüber dem in [ISP] definierten Pascal-Standard enthält, als sehr nützlich erwiesen.

In den folgenden Kapiteln werden **Datenobjekte auf verschiedenen Abstraktionsebenen** betrachtet:

- In einer möglichen **internen Darstellung** im Arbeitsspeicher eines Rechners. Diese Darstellungsform ist maschinenabhängig. Die Kenntnis der Umsetzung von Anwenderdefinitionen in maschinennahe Darstellung ist spätestens bei der Fehlersuche hilfreich bzw. zwingend erforderlich, nämlich dann, wenn ein Speicherauszug eines Programms zu interpretieren ist

- In Form der **Deklarationen in einer höheren Programmiersprache.** Diese Abstraktionsebene ist anwendungsbezogen und maschinenunabhängig, auch wenn es je nach Programmiersprache noch implizite Beziehungen zu einer Modellvorstellung eines möglichen verwendeten Rechners gibt.

Ein Datenobjekt "existiert" erst, wenn es eine Adresse im virtuellen Adreßraum hat. Sein Datentyp beschreibt, auf welche Weise es im virtuellen Adreßraum abgelegt wird und welche Operationen mit ihm erlaubt sind. Die Abstraktionsebene der internen Darstellung gibt daher ein Datenobjekt vollständig wieder. Die höheren Abstraktionsebenen, die Datenobjekte in der Assembler-Sprache des eingesetzten Rechners oder in einer höheren Programmiersprache deklarieren, ***beschreiben*** lediglich Datenobjekte und die Art, wie sie später zur Laufzeit "ins Leben gerufen" und wertmäßig manipuliert werden. Durch eine Deklaration eines Datenobjekts in einem HOL-Programm wird noch kein Datenobjekt erzeugt, auch wenn der geläufige Sprachgebrauch einer Deklaration diese Vorstellung suggeriert. Im folgenden kann diese Vorstellung der Erzeugung eines Datenobjekts durch eine Deklaration in einer Programmiersprache trotzdem zugrundegelegt werden. Ein Anwender möchte die internen Details seiner deklarierten Datenobjekte ja nicht sehen; daher hat er ja seine Problemlösung in einer höheren Programmiersprache formuliert. Aus seiner Sicht wird durch eine Deklaration ein Datenobjekt erzeugt. Es ist also sinnvoll, von der Umsetzung einer Deklarationen aus der höheren Programmiersprache in die interne Darstellung des Rechners zu sprechen.

Ein **Datenobjekt** zeichnet sich durch verschiedene Charakteristika aus (Abbildung 4-1).

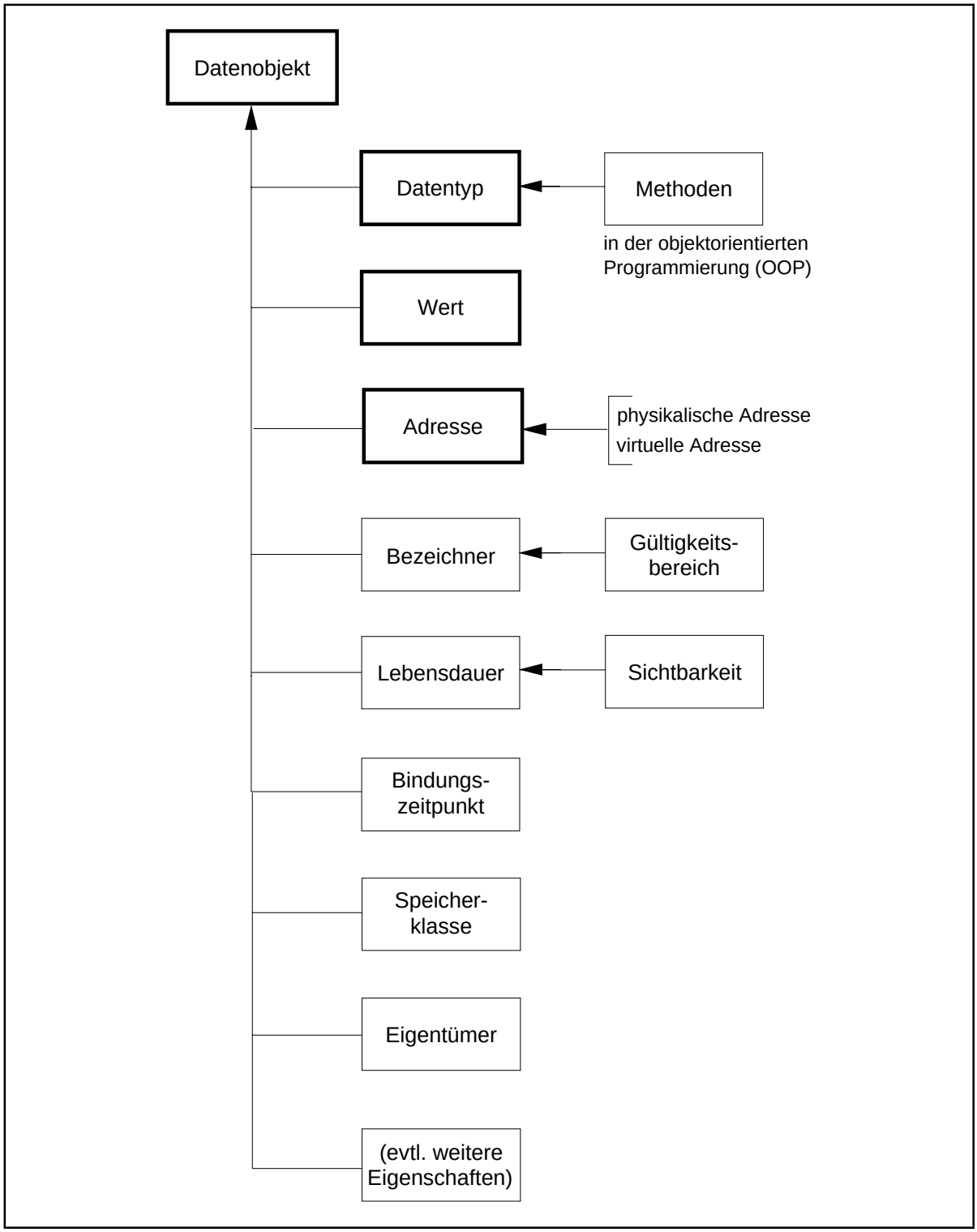


Abbildung 4-1: Charakteristische Eigenschaften eines Datenobjekts

Die minimale Menge dieser Eigenschaften (stark umrandeten Felder in Abbildung 4-1) beinhaltet:

- den **Datentyp** eines Datenobjekts, d.h. die Menge der Attribute, insbesondere die Darstellungsform, den möglichen Wertebereich und die mit dem Objekt erlaubten Operationen. In der **objektorientierten Programmierung (OOP)** lassen sich durch den Datentyp neben den Grundoperationen weitere Operationen in Form von **Methoden** und weitere typabhängige Attribute wie **Vererbungshierarchien** definieren

- den gegenwärtige **Wert** eines Datenobjekts, der im Lauf der Zeit durch das Programm verändert werden kann. Vor der ersten Wertzuweisung an das Datenobjekt ist der Wert im allgemeinen undefiniert

- die **Adresse** eines Datenobjekts, d.h. die Stelle innerhalb des virtuellen oder (während der Laufzeit) physikalischen Adreßraums, an der es sich befindet; diese Adresse ist eindeutig bestimmt, auch wenn sie dem Anwender nicht immer direkt zugänglich ist.

Auf der Abstraktionsebene einer höheren Programmiersprache kommen zusätzliche Eigenschaften eines Datenobjekts hinzu, die in den folgenden Kapiteln genauer behandelt werden, wie

- der **Bezeichner (Name)**, der angibt, wie das Datenobjekt im Programm angesprochen wird. Der **Gültigkeitsbereich des Bezeichners** beschränkt die Verwendbarkeit dieses Bezeichners

- die **Lebensdauer**, die die Zeitspanne umfaßt, in der das Datenobjekt im Rechner existiert. Dabei kann es möglich sein, daß ein Datenobjekt zeitweise nicht sichtbar ist, d.h. daß es nicht angesprochen bzw. adressiert werden kann.

Weitere Charakteristika, die nicht alle Programmiersprachen vorsehen, sind beispielsweise

- der **Bindungszeitpunkt** des Bezeichners an das Datenobjekts: in einigen Programmiersprachen (z.B. einigen Sprachen der 4. Generation) wird ein Datenobjekt an einen Bezeichner erst dann gebunden, d.h. eingerichtet, wenn eine Anweisung zum ersten Mal durchlaufen wird, die den Bezeichner enthält. Je nach Situation wird jetzt erst der Datentyp festgelegt und damit implizit die mit dem Datenobjekt anschließend erlaubten Operationen[14]. Ein derartiges Verhalten ist natürlich nur in nicht streng typisierten Programmiersprachen möglich, in denen Typfestlegungen erst zur Laufzeit erfolgen, oder falls der betroffene Sprachdialekt eine Aufweichung eines an sich stringenten Typkonzepts zuläßt

- die **Speicherklasse**, die bestimmt, in welchem Teil des Laufzeit-Layouts der Compiler das Datenobjekt anlegen soll

[14] Kommt beispielsweise der Bezeichner `xyz` in einer Eingabeoperation vor und ist diese Anweisung die erste, die den Bezeichner `xyz` enthält und durchlaufen wird, so wird das Datenobjekt mit dem Bezeichner `xyz` eine `INTEGER`-Variable, falls jetzt eine ganze Zahl eingegeben wird. Wird dagegen eine Zeichenkette eingegeben, so ist das Datenobjekt `xyz` eine Variable vom Typ `STRING`.

- der **Eigentümer**, d.h. derjenige Programmteil, der die Rechte bezüglich des Zugriffs auf das Datenobjekt kontrolliert.

Die übliche Art, ein Datenobjekt in einer höheren Programmiersprache zu deklarieren, besteht in der Angabe des Typs des Objekts und damit implizit der zulässigen Operationen und Methoden, die auf das Datenobjekt dieses Typs anwendbar sind, und der Angabe des Bezeichners für das Datenobjekt, etwa in der in Pascal üblichen Form:

als Variable	als typisierte Konstante

`TYPE data_typ = ...;`
 	
`VAR  datenobjekt : data_typ;`
	`TYPE data_typ = ...;`
	
	`CONST datenobjekt : datentyp = ...`
	`{ Anfangswert };`

Für `data_typ` kommen auch in der Programmiersprache vordefinierte Datentypen in Frage; dann erübrigt sich die explizite Deklaration des Datentyps mittels der `TYPE`-Angabe (Einzelheiten behandelt Kapitel 4.2). Die übrigen Charakteristika ergeben sich (zumindest in Pascal) aus dem Zusammenhang, in dem die Deklaration vorkommt.

4.1 Grundlegende Datentypen eines Rechners

Dieses Kapitel beschreibt die Grunddatentypen, über die ein Rechner üblicherweise verfügt und auf denen die Datentypen auf höheren Abstraktionsebenen aufbauen. Ein Datenobjekt eines Programms, das in einer höheren Programmiersprache definiert und eventuell anwendungsspezifisch strukturiert ist, muß letztlich auf ein (eventuell zusammengesetztes) **internes Datenobjekt** übersetzt werden, d.h. auf ein Datenobjekt im virtuellen Adreßraum und schließlich im Arbeitsspeicher. Die Art der Abbildung und damit der internen Datendarstellung ist Rechnertyp-abhängig. Exemplarisch sollen hier der INTEL 80x86-Prozessor und die in der kommerziellen Großrechnerwelt häufig anzutreffenden Bytemaschinen vom Typ IBM /370 gegenübergestellt werden (vgl. [THI], [MAT], [HOF]). Beide Rechner verwenden ähnliche Grunddatentypen, die sich jedoch an einigen Stellen wesentlich unterscheiden.

Die kleinste adressierbare Einheit ist das **Byte** mit jeweils 8 Bits. Die einzelnen Bits innerhalb eines Bytes, auch wenn sie nicht einzeln durch Adressen ansprechbar sind, sind ***von rechts nach links*** von 0 bis 7 durchnumeriert. Die Nummern bezeichnet man als **Bitpositionen (innerhalb des Bytes)**. Ein Bit kann die Werte 0 oder 1 annehmen, so daß für den Inhalt eines Bytes $2^8 = 256$ verschiedene Bitmuster möglich sind. Eine Bitfolge der Form $[a_7 a_6 \ldots a_1 a_0]_2$ kann als Binärzahl interpretiert werden. Sie entspricht dann dem Wert $\sum_{i=0}^{7} a_i 2^i$.

Ein Byte kann also in dieser Interpretation die Dezimalwerte 0, 1, ..., 255 enthalten.

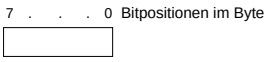
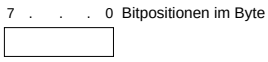
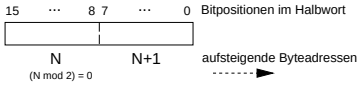
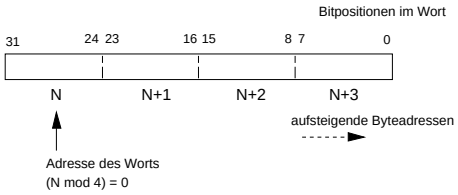
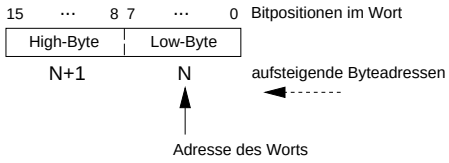
Üblicherweise faßt man zur Schreibvereinfachung jeweils 4 nebeneinanderliegende Bitwerte (von rechts beginnend, wobei eventuell links führende binäre Nullen zur Auffüllung gedacht werden müssen) zu einer Sedezimalziffer zusammen und erhält damit die gleichwertige Sedezimaldarstellung. Als Ziffern kommen in der Sedezimaldarstellung die sechzehn Zeichen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F vor.

Um zu unterscheiden, ob eine Zahl als Binär- , Sedezimal- bzw. Dezimaldarstellung anzusehen ist, wird in nichteindeutigen Situationen die Basis 2, 16 bzw. 10 als Index angehängt.

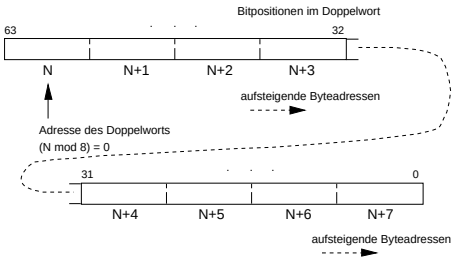
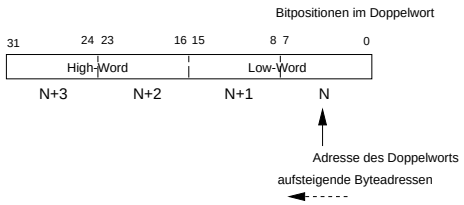
Binärdarstellung (Basis 2) $[a_7 a_6 \ldots a_1 a_0]_2$	Sedezimaldarstellung (Basis 16)	Dezimalwert (Basis 10) $\sum_{i=0}^{7} a_i 2^i$
0000 0000	00	0
0000 0001	01	1
0000 0010	02	2
0000 0011	03	3
0000 0100	04	4
0000 0101	05	5
0000 0110	06	6
0000 0111	07	7
0000 1000	08	8
0000 1001	09	9
0000 1010	0A	10
0000 1011	0B	11
0000 1100	0C	12
0000 1101	0D	13
0000 1110	0E	14
0000 1111	0F	15
0001 0000	10	16
0001 0001	11	17
...	...	...
0001 1111	1F	31
0010 0000	20	32
0010 0001	21	33
...	...	...
0010 1111	2F	47
0011 0000	30	48
0011 0001	31	49
...	...	...
0011 1111	3F	63
...	...	...
1111 0000	F0	240
1111 0001	F1	241
...	...	...
1111 1110	FE	254
1111 1111	FF	255

Abbildung 4.1-1: Byteinhalt

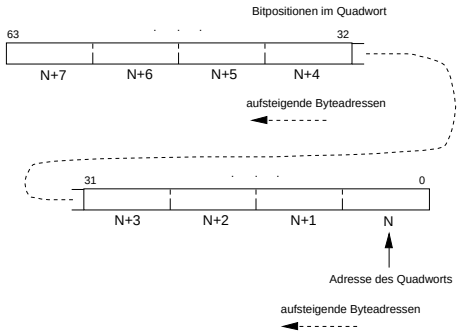
Üblicherweise werden aufeinanderfolgende Bytes zu den weiteren logischen Einheiten **Halbwort**, **Wort**, **Doppelwort** und **Quadwort** zusammengefaßt. Innerhalb dieser Einheiten werden wieder einzelne Bitpositionen von 0 beginnend gezählt. Zu beachten ist, daß beim IBM /370-Rechner die höherwertigen Bits an den niedrigen Adressen liegen, während beim INTEL 80x86-Prozessor die höherwertigen Bits an den höheren Adressen stehen. Eine Gruppe (Halbwort, Wort, Doppelwort bzw. Quadwort) kann als ganzes durch eine Adresse angesprochen werden. Diese Adresse meint in beiden Architekturen den Anfang der Gruppe, d.h. die kleinste Byteadresse innerhalb der Gruppe. Bei IBM /370 ist dies das Byte mit den höchstwertigen Bits, bei INTEL 80x86 das Byte mit den niedrigstwertigen Bits; entsprechend nennt man die IBM /370-Rechner **Big-Ending**-[15] und die INTEL 80x86-Rechner **Little-Ending**-Maschinen. Außerdem meinen die jeweiligen Gruppierungsbegriffe in den Architekturen unterschiedliche Byteanzahlen. Abbildung 4.1-2 faßt die Gruppierungen zusammen.

	Big Ending (z.B. IBM /370)	Little Ending (z.B. INTEL 80x86)
Byte		
Halbwort	2 hintereinanderliegende Bytes, deren erstes bei einer geraden Adresse beginnt (**Ausrichtung auf Halbwortgrenze**). 	---
Wort	4 hintereinanderliegende Bytes (2 Halbworte), deren erstes bei einer durch 4 teilbaren Adresse beginnt (Ausrichtung auf **Wortgrenze**)  Zu beachten ist, daß das Byte, das die höchste Bitposition im Wort enthält, auf der niedrigsten Byteadresse innerhalb des Worts steht.	2 hintereinanderliegende Bytes, die an einer beliebigen Adresse beginnen können. Das Byte, das Bitposition 0 enthält, heißt **Low-Byte**, das Byte mit Bitposition 15 **High-Byte**.  In der Reihenfolge der Byteadressierung gilt der Grundsatz ***"Low-Byte first"***. Beginnt ein Wort an einer geraden Adresse, so heißt es **ausgerichtetes Wort** (word aligned).

[15] Ein weiterer typischer Vertreter der Big-Ending-Architektur ist der Motorola 68000-Rechner.

<table>
<tr><td>Doppelwort</td><td>8 hintereinanderliegende Bytes (2 Worte), deren erstes bei einer durch 8 teilbaren Adresse beginnt (Ausrichtung auf Doppelwortgrenze)
 
</td><td>4 hintereinanderliegende Bytes (2 Worte), die an einer beliebigen Adresse beginnen können. Das Wort, das Bitposition 0 enthält, heißt Low-Word, das Wort mit Bitposition 31 High-Word.
 

In der Reihenfolge der Byteadressierung gilt der Grundsatz "Low-Word first" (und innerhalb eines Worts "Low-Byte first").

Beginnt ein Doppelwort an einer durch 4 teilbaren Adresse, so heißt es ausgerichtetes Doppelwort (doubleword aligned).</td></tr>
<tr><td>Quadwort</td><td>---</td><td>8 hintereinanderliegende Bytes (2 Doppelworte), die an einer beliebigen Adresse beginnen können. Das Wort, das Bitposition 0 bis 31 enthält, heißt Low-Doubleword, das Wort mit Bitposition 32 bis 63 High-Doubleword.
 
</td></tr>
<tr><td colspan="3">Jedes Byte in einem Halbwort, Wort bzw. Doppelwort hat weiterhin seine eigene Adresse. Die Adresse des ersten Bytes entspricht der Adresse des Halbworts, Worts, Doppelworts bzw. Quadworts.

Die Ausrichtung der jeweiligen Bytegruppe, die beim INTEL 80x86-Rechner nicht notwendig ist, beschleunigt den Arbeitsspeicherzugriff: Obwohl die Adressierung einzelner Bytes aus einem Programm heraus möglich, werden über den Bus immer Doppelworte übertragen, die an Doppelwortgrenze liegen. Das bedeutet, daß bei dem Zugriff auf eine nicht ausgerichtete Bytegruppe u.U. zwei Datentransfer erforderlich sind, während bei Ausrichtung nur ein Datentransfer ausreicht.</td></tr>
</table>

Abbildung 4.1-2: Halbwort, Wort, Doppelwort, Quadwort

Die meisten Rechnersysteme definieren eine Reihe von **Grunddatentypen**, die auch in höheren Programmiersprachen verwendet werden. Zusätzlich lassen sich (in Programmiersprachen) aus diesen Grunddatentypen mit Hilfe festgelegter Regeln weitere Datentypen ableiten. Die gebräuchlichsten Formen zeigt Abbildung 4.1-3.

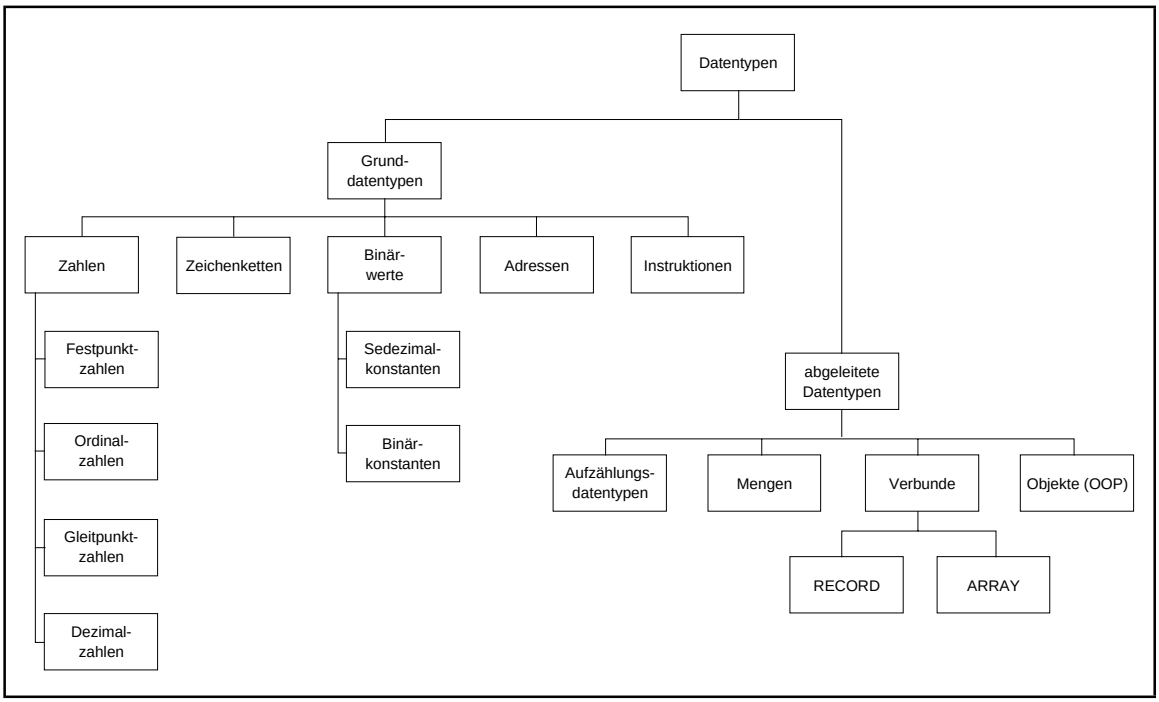


Abbildung 4.1-3: Grunddatentypen

4.1.1 Datentyp Zahl

Der Datentyp **Zahl** wird für Datenobjekte verwendet, deren Werte von arithmetischen Operationen verändert werden können. Eine Ausnahme bildet der Untertyp ungepackte Dezimalzahl, der als Zwischenform zwischen einem Datentyp für Rechenoperationen und einem Datentyp zur Darstellung druckaufbereiteter Zahlen angesehen werden kann. Aufgrund der häufig vorkommenden unterschiedlichen Anwendungen und des zur Verfügung stehenden Instruktionssatzes für arithmetische Operationen unterscheidet man die Datentypen Festpunktzahl, Ordinalzahl, Gleitpunktzahl und Dezimalzahl. ***Für Rechenoperationen mit jedem dieser Datentypen gibt es einen eigenen Satz an Maschineninstruktionen.***

Ein gemäß dem Datentyp **Festpunktzahl** definiertes Datenobjekt belegt je nach Rechnertyp und zusätzlicher Festlegung ein Byte, Halbwort, Wort, Doppelwort oder Quadwort (z.B. IBM /370: Halbwort, Wort; INTEL 80x86: Byte, Wort, Doppelwort, Quadwort) und wird als positive bzw. negative ganze Zahl (in der Darstellung als Binärzahl im Zweierkomplement) interpretiert. Die an der höchsten Bitposition stehende Bitstelle gibt das Vorzeichen der Zahl an: eine positive Zahl beginnt mit 0_2, eine negative Zahl mit 1_2.

Auf diese Weise wird bei n Stellen (n = 8 bzw. n = 16 bzw. n = 32 bzw. n = 64, je nach Rechnertyp) die Bitfolge

$0a_{n-2}a_{n-3}\ldots a_1a_0$ mit $a_i \in \{0,1\}$

als

$$\sum_{i=0}^{n-2} a_i 2^i$$

interpretiert. Entsprechend bedeutet die Bitfolge

$1a_{n-2}a_{n-3}\ldots a_1a_0$

$$\sum_{i=0}^{n-2} a_i 2^i - 2^{n-1}$$

und somit eine negative Zahl, da $\sum_{i=0}^{n-2} a_i 2^i \leq 2^{n-1} - 1$ ist.

Die Größe einer Festpunktzahl bei liegt also bei n = 8 bzw. n = 16 bzw. n = 32 bzw. n = 64 im Bereich -128, ... 127 bzw. -32.768, ..., 32.767 bzw. -2.147.483.648, ..., 2.147.483.647 bzw. -2^{63} , ..., 2^{63} - 1; $2^{63} > 9{,}22 \cdot 10^{18}$.

Der Datentyp **Ordinalzahl** beschreibt eine vorzeichenlose Binärzahl in einem Byte, Wort oder Doppelwort (z.B. IBM /370: Wort; INTEL 80x86: Byte, Wort, Doppelwort), d.h. es werden alle Bits zur Darstellung der Zahl verwendet. Die Bitfolge

$a_{n-1}a_{n-2}\ldots a_1a_0$ mit $a_i \in \{0,1\}$

wird also als

$$\sum_{i=0}^{n-1} a_i 2^i$$

interpretiert. Die Größe einer Ordinalzahl bei liegt bei n = 8 bzw. n = 16 bzw. n = 32 im Bereich 0, ..., 255 bzw. 0, ..., 65.535 bzw. 0, ..., 4.294.947.295.

Der Datentyp **Gleitpunktzahl** dient zur Approximation einer reellen Zahl durch eine rationale Zahl mit endlichem gebrochenem Anteil und beruht auf der Darstellung einer Zahl x mit ganzzahligem und gebrochenem Anteil in der Form

$x = \pm|x| = \pm m B^e$.

± bezeichnet das **Vorzeichen**, m ist die **Mantisse** (in der INTEL 80x86-Architektur **Signifikand** genannt), B die **Basis** der Zahlendarstellung und e der **Exponent** der Zahl.

Der Datentyp Gleitpunktzahl in der IBM /370-Architektur unterscheidet sich in einigen Details wesentlich vom Datentyp Gleitpunktzahl der INTEL 80x86-Architektur.

A. Datentyp Gleitpunktzahl in der IBM /370-Architektur

Hier lautet die Basis B = 16.

Die Mantisse m als eine Ziffernfolge von Sedezimalziffern

$m = b_{-1}b_{-2}\ldots b_{-k}$ mit $b_{-1} \neq 0$

wird als

$$\sum_{i=1}^{k} b_{-i}16^{-i}$$

interpretiert. Damit der Exponent eindeutig bestimmt ist, wird eine **Normalisierung** durchgeführt: Dies geschieht durch Verschieben des "sedezimalen Kommas" und Anpassung des Exponenten, so daß $b_{-1} \neq 0$ gilt. Durch die Normalisierung wird eine höhere Genauigkeit bei der Zahlendarstellung erreicht; außerdem erwarten manche Maschineninstruktionen normalisierte Operanden, um normalisierte Ergebnisse zu erzeugen.

Ein Datenobjekt vom Datentyp Gleitpunktzahl wird in einem Wort, einem Doppelwort oder zwei aufeinander folgenden Doppelworten abgelegt. Man spricht dann von einer **kurzen Gleitpunktzahl** (**Short Real**), einer **langen Gleitpunktzahl** (**Long Real**) bzw. einer **Gleitpunktzahl im erweiterten Format** (**Extended Real**).

Ein Bit repräsentiert jeweils das Vorzeichen, und zwar bedeutet 0_2 das positive Vorzeichen und 1_2 das negative Vorzeichen. Eine negative Zahl wird nicht im Zweierkomplement dargestellt; die Mantisse ist also immer positiv. Je mehr Stellen sie enthält, umso genauer ist die Approximation der dargestellten Zahl an die darzustellende Zahl. Statt des tatsächlichen Exponenten e einer Zahl wird deren **Charakteristik** $c = e + b$ (ohne Übertrag) gespeichert. Die Zahl b heißt **Bias**. Die Funktion des Bias besteht darin, statt eines möglichen positiven oder negativen Exponenten immer eine nichtnegative Charakteristik (vorzeichenlos) abzuspeichern, so daß Gleitpunktzahlen mit gleichem Format und Vorzeichen wie vorzeichenlose Festpunktzahlen verglichen werden können.

Zu beachten ist, daß die Anzahl der Bits für die Charakteristik im wesentlichen die Größe des darstellbaren Wertebereichs festlegt.

In der IBM /370-Architektur mit dem Bias $b = 64$ für alle Datenformate ist beispielsweise die durch den Datentyp kurzes Format darstellbare *größte Zahl* gleich
$7F\ FF\ FF\ FF_{16}$ (Vorzeichen = 0_2; Charakteristik = $1111111_2 = 127_{10}$ entsprechend dem Exponenten 63_{10});
diese Zahl hat den Wert $+0{,}FFFFFF_{16} \cdot 16^{63} = (1-16^{-6}) \cdot 16^{63} \approx 7{,}2370_{10} \cdot 10^{75}$.

Die *kleinste Zahl* ist
$FF\ FF\ FF\ FF_{16}$ (Vorzeichen = 1_2; Charakteristik = $1111111_2 = 127_{10}$ entsprechend dem Exponenten 63_{10});
diese Zahl hat den Wert $-0{,}FFFFFF_{16} \cdot 16^{63} = -(1-16^{-6}) \cdot 16^{63} \approx -7{,}2370_{10} \cdot 10^{75}$.

Die mit dem Datentyp kurze Gleitpunktzahl darstellbare *kleinste positive Zahl größer als Null* ist (bedingt durch die Normalisierung)
$00\ 10\ 00\ 00_{16}$ (Vorzeichen = 0_2; Charakteristik = 0000000_2 entsprechend dem Exponenten -64_{10});
diese Zahl hat den Wert $+0{,}1_{16} \cdot 16^{-64} = 16^{-65} \approx 5{,}3976_{10} \cdot 10^{-79}$.

Für den **absoluten Wertebereich** W_{kG} einer Zahl mit Datentyp kurze Gleitpunktzahl gilt hier also
$16^{-65} \le W_{kG} \le (1-16^{-6})\cdot 16^{63}$,
$16^{-65} \approx 5{,}3976_{10}\cdot 10^{-79}$ und $(1-16^{-6})\cdot 16^{63} \approx 7{,}2370_{10}\cdot 10^{75}$.
Das bedeutet an der unteren Grenze des Wertebereichs, daß die nächstkleinere (gültige) Zahl die Zahl 0 ist. Es wird natürlich jeweils vorausgesetzt, daß die Gleitpunktzahl normalisiert ist[16].

Der absolute Wertebereich W_{lG} einer Zahl vom Datentyp lange Gleitpunktzahl bzw. der absolute Wertebereich W_{eG} einer Zahl vom Datentyp Gleitpunktzahl im erweiterten Format lautet bei IBM /370:
$16^{-65} \le W_{lG} \le (1-16^{-14})\cdot 16^{63} \approx 7{,}2370_{10}\cdot 10^{75}$
bzw.
$16^{-65} \le W_{eG} \le (1 - 16^{-28})\cdot 16^{63} \approx 7{,}2370_{10}\cdot 10^{75}$.
Die möglichen Wertebereiche bei den drei Datentypen Gleitpunktzahl unterscheiden sich hier also nicht wesentlich. Die Unterschiede liegen nicht im Wertebereich, sondern in der **Genauigkeit**, mit der man mit Hilfe eines Datenobjekts vom Typ Gleitpunktzahl eine reelle Zahl approximiert. Diese Genauigkeit wird durch den kleinstmöglichen Abstand Δ_{min} zwischen zwei Zahlen vom Datentyp Gleitpunktzahl bestimmt; je dichter zwei Zahlen zusammenliegen, d.h. je kleiner dieser Abstand ist, umso genauer ist die Approximation. Dieser kleinste Abstand Δ_{min} beträgt
beim kurzen Format: $0{,}000001_{16}\cdot 16^{-64} = 16^{-70} \approx 5{,}1476_{10}\cdot 10^{-85}$,
beim langen Format: $0{,}00000000000001_{16}\cdot 16^{-64} = 16^{-78} \approx 1{,}1985_{10}\cdot 10^{-94}$,
beim erweiterten Format: $0{,}0...01_{16}\cdot 16^{-64} = 16^{-26}\cdot 16^{-64} = 16^{-90} \approx 4{,}2580_{10}\cdot 10^{-109}$.

Vereinbarungsgemäß hat bei IBM /370 der Wert 0 das Vorzeichen 0_2, die Mantisse m = 0 und den Exponenten $0...0_2$.

B. Datentyp Gleitpunktzahl in der INTEL 80x86-Architektur

Hier lautet die Basis $B = 2$.

In der INTEL 80x86-Architektur, die sich am IEEE-Standard 754-1985 orientiert, liegt eine kurze Gleitpunktzahl in einem Doppelwort (4 Bytes), eine lange Gleitpunktzahl in

[16] Läßt man diese Forderung fallen, dann ergibt sich als kleinste positive Zahl die Gleitpunktzahl $00\ 00\ 00\ 01_{16}$ (Vorzeichen = 0_2; Charakteristik = 0000000_2 entsprechend dem Exponenten -64_{10}); diese Zahl hat den Wert $+0{,}000001_{16}\cdot 16^{-64} = 16^{-70} \approx 5{,}1476_{10}\cdot 10^{-85}$.

einem Quadwort (8 Bytes) und eine Gleitpunktzahl im erweiterten Format (auch **Temporary Real** genannt) in einem Feld von 10 zusammenhängenden Bytes. Die eingebaute Gleitkomma-Prozessoreinheit FPU arbeitet mit diesem Format.

Die Mantisse (Signifikand) *m* als eine Ziffernfolge von Binärziffern
$m = a_0 a_{-1} a_{-2} \ldots a_{-k}$
wird so normalisiert, daß $a_0 = 1_2$ gilt, d.h. die Mantisse wird als

$$1 + \sum_{i=1}^{k} a_{-i} 2^{-i}$$

interpretiert. Das Bit $a_0 = 1_2$ wird im kurzen und langen Format nicht mit gespeichert.

Ein Bit repräsentiert das Vorzeichen, und zwar bedeutet 0_2 wieder das positive Vorzeichen und 1_2 das negative Vorzeichen. Statt des tatsächlichen Exponenten *e* einer Zahl wird deren **Charakteristik** $c = e + b$ (ohne Übertrag), hier als **Biased Exponent** bezeichnet, gespeichert. Die Größe des Bias *b* ist vom jeweiligen Datentyp abhängigen[17]. Maximale und minimale Exponenten, sowie den jeweiligen Bias kann man der folgenden Tabelle entnehmen:

	Short Real	Long Real	Temporary Real
Breite insgesamt (Bits)	32	64	80
Breite des Signifikanden (Bits)	23 + 1*	52 + 1*	64
Exponentenbreite (Bits)	8	11	15
maximaler Exponent	127	1.023	16.383
minimaler Exponent	-126	-1.022	-16.382
Bias	127	1.023	16.383
* Die führende 1_2 wird nicht mit gespeichert.			

Die durch den Datentyp kurzes Format darstellbare *größte Zahl* ist jetzt gleich
0 11111110 11...1_2 = 7F 7F FF FF_{16} (Vorzeichen = 0_2; Charakteristik = 11111110_2 = 254_{10} entsprechend dem Exponenten 127_{10});
diese Zahl hat den Wert $+1{,}11111111111111111111111_2 \cdot 2^{127} = (2^{24}-1) \cdot 2^{104} \approx 3{,}4028_{10} \cdot 10^{38}$.

Die *kleinste Zahl* im kurzen Gleitpunktformat ist
FF 7F FF FF_{16} entsprechend $-3{,}4028_{10} \cdot 10^{38}$.

[17] Auch hier besteht die Funktion des Bias darin, statt eines möglichen positiven oder negativen Exponenten immer eine nichtnegative Charakteristik (vorzeichenlos) abzuspeichern, so daß Gleitpunktzahlen mit gleichem Format und Vorzeichen wie vorzeichenlose Festpunktzahlen verglichen werden können.

Die mit dem Datentyp kurze Gleitpunktzahl darstellbare *kleinste positive Zahl größer als Null* ist (bedingt durch die Normalisierung)
$0\ 00000001\ 00...0_2 = 00\ 80\ 00\ 00_{16}$ (Vorzeichen = 0_2; Charakteristik = $00000001_2 = 1_{10}$ entsprechend dem Exponenten -126_{10});
diese Zahl hat den Wert $+1{,}0_2 \cdot 2^{-126} = 2^{-126} \approx 1{,}1755_{10} \cdot 10^{-38}$.

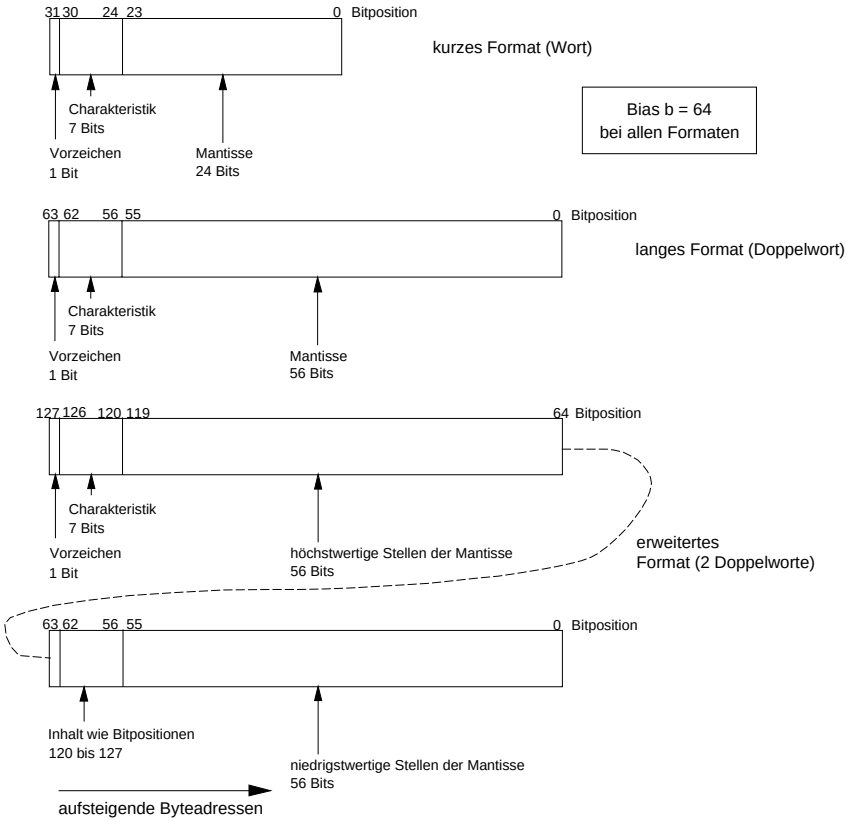
Entsprechend lassen sich die absoluten Wertebereiche für die übrigen Datenformate der Gleitpunktzahlen errechnen (siehe Abbildung 4.1.1-1).

Die Genauigkeit, die den kleinstmöglichen Abstand Δ_{min} zwischen zwei Zahlen vom Datentyp Gleitpunktzahl bestimmt, ist
beim kurzen Format: $2^{-149} \approx 1{,}4013_{10} \cdot 10^{-45}$,
beim langen Format: $2^{-1074} \approx 4{,}9407_{10} \cdot 10^{-324}$,
beim erweiterten Format: $2^{-16445} \approx 3{,}6452_{10} \cdot 10^{-4951}$.

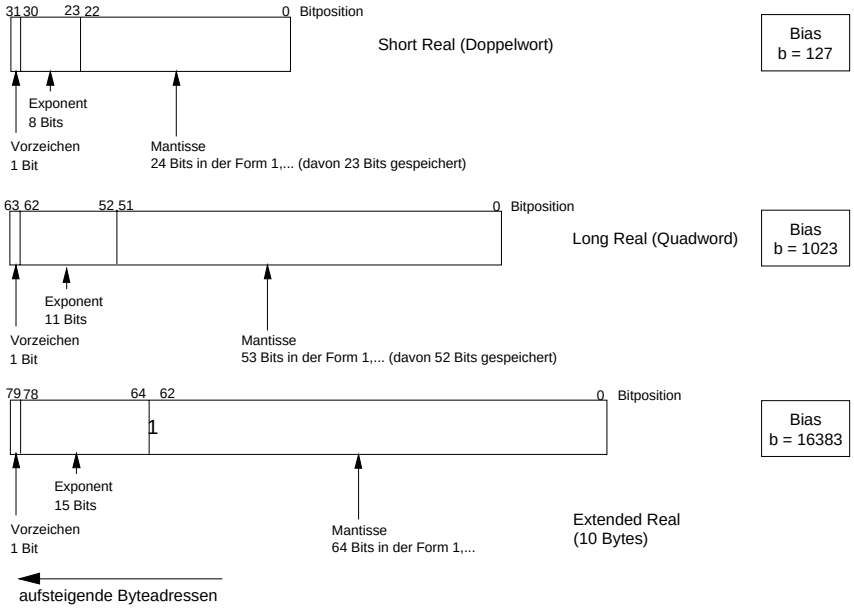
Einige Bitmuster stellen die speziellen Werte der **NaN**s **(Not a Number)** dar: Die Charakteristik (biased Exponent) enthält nur 1_2, und der Signifikand kann jeden Wert außer $10...0_2$ annehmen. Es gibt zwei Typen von NaNs: signalisierende NaNs (höchstwertiges Signifikandenbit = 1_2) und stille NaNs (höchstwertiges Signifikandenbit = 0_2). Eine **signalisierende NaN** wird von der CPU nie erzeugt und kann daher von einem Programm verwendet werden, um besondere Ausnahmesituationen zu behandeln, die in den Signifikandenbits angezeigt werden können. Beispielsweise können alle nichtbelegten Einträge einer ARRAY-Struktur als solche durch NaN-Werte gekennzeichnet werden. Jeder Versuch, ein so gekennzeichnetes Element zu verarbeiten, wird von der CPU als Ausnahmesituation angezeigt; im Signifikandenfeld kann dabei z.B. die entsprechende ARRAY-Position kodiert sein. **Stille NaNs** werden von der CPU bei den verschiedenen Ausnahmesituationen "ungültige Operation" erzeugt (die stille NaN "undefiniert"). Weitere Details findet man beispielsweise in [MAT].

Abbildung 4.1.1-1 faßt die Datenformate für Gleitpunktzahlen zusammen.

IBM /370



INTEL 80x86



absoluter Wertebereich W (normalisiert[18])		
	IBM /370	INTEL 80x86
kurzes F.	$5{,}3976_{10}\cdot10^{-79} < W < 7{,}2371_{10}\cdot10^{75}$	$1{,}1755_{10}\cdot10^{-38} < W < 3{,}4029_{10}\cdot10^{38}$
langes F.	$5{,}3976_{10}\cdot10^{-79} < W < 7{,}2371_{10}\cdot10^{75}$	$2{,}2250_{10}\cdot10^{-308} < W < 1{,}7977_{10}\cdot10^{308}$
erw. F.	$5{,}3976_{10}\cdot10^{-79} < W < 7{,}2371_{10}\cdot10^{75}$	$3{,}3621_{10}\cdot10^{-4932} < W < 1{,}1898_{10}\cdot10^{4932}$
kleinstmöglicher Abstand Δ_{min} zwischen zwei Zahlen		
	IBM /370	INTEL 80x86
kurzes F.	$\approx 5{,}1476_{10}\cdot10^{-85}$	$\approx 1{,}4013_{10}\cdot10^{-45}$
langes F.	$\approx 1{,}1985_{10}\cdot10^{-94}$	$\approx 4{,}9407_{10}\cdot10^{-324}$
erw. F.	$\approx 4{,}2580_{10}\cdot10^{-109}$	$\approx 3{,}6452_{10}\cdot10^{-4951}$
spezielle Werte (hier nur für Short Real)		
	IBM /370	INTEL 80x86
0	$00\ 00\ 00\ 00_{16}$	$+0 = 00\ 00\ 00\ 00_{16}$ $-0 = 80\ 00\ 00\ 00_{16}$
$+\infty$		$7F\ 80\ 00\ 00_{16}$
$-\infty$		$FF\ 80\ 00\ 00_{16}$
NaN		$7F\ 80\ 00\ 01_{16}$, ..., $7F\ FF\ FF\ FF_{16}$, $FF\ 80\ 00\ 01_{16}$, ..., $FF\ FF\ FF\ FF_{16}$
unbestimmt		$FF\ 00\ 00\ 00_{16}$
	aufsteigende Adressen →	← aufsteigende Adressen

Abbildung 4.1.1-1: Datentyp Gleitpunktzahl

Ein weiterer Datentyp zur internen Darstellung von Zahlen ist der Datentyp **Dezimalzahl** mit den beiden Untertypen gepackte Dezimalzahl bzw. ungepackte Dezimalzahl. Beide Datentypen dienen der Darstellung ganzer Zahlen im Dezimalformat, d.h. mit den

[18] An der Obergrenze ist im Faktor die letzte Dezimalstelle nach dem Komma des Faktors aufgerundet, an der Untergrenze abgerundet.

Dezimalziffern $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Die IBM /370-Architektur besitzt eine ausgeprägte Unterstützung der Dezimalarithmetik durch entsprechende Maschineninstruktionen im Gegensatz zur INTEL 80x86-Architektur.

Bei einer Zahl vom Datentyp **gepackte Dezimalzahl** wird jede Dezimalziffer (die ja auch als Sedezimalziffer angesehen werden kann) so abgelegt, daß jeweils zwei Dezimalziffern der Zahl ein Byte belegen.

A. Datentyp gepackte Dezimalzahl in der IBM /370-Architektur

Das Vorzeichen der Zahl wird hier im äußerst rechten Byte (höchste Adresse in der internen Darstellung der Zahl) in der rechten Ziffernstelle, d.h. der kleinsten Ziffernposition, kodiert: dazu wird für das positive Vorzeichen die Sedezimalziffer C_{16} angehängt, für das negative Vorzeichen die Sedezimalziffer D_{16}. Die Sedezimalziffern A_{16} und E_{16} werden ebenfalls als Kodierungen des positiven Vorzeichens, die Sedezimalziffer B_{16} als Kodierung des negativen Vorzeichens interpretiert.

Die Zahl wird eventuell links mit einer führenden 0 aufgefüllt, so daß immer eine volle Anzahl von Bytes entsteht. Zur Darstellung einer *n*-stelligen Dezimalzahl als gepackte Dezimalzahl werden $\lceil (n+1)/2 \rceil$ viele Bytes benötigt. Es besteht die Einschränkung, daß diese Stellenzahl maximal 16 sein darf. Eine Ausrichtung auf Halbwort-, Wort- oder Doppelwortgrenze gibt es nicht.

B. Datentyp gepackte Dezimalzahl in der INTEL 80x86-Architektur

Die INTEL 80x86-Architektur unterstützt keine Vorzeichendarstellung (außer in einem speziellen Untertyp der Gleitpunkt-Verarbeitungseinheit FPU).

Bei einer Zahl vom Datentyp **ungepackte Dezimalzahl** belegt jede Dezimalziffer (die wieder auch als Sedezimalziffer angesehen werden kann) ein Byte, wobei die Dezimalziffer in den Bitpositionen 0 bis 3 steht; die Bitpositionen 4 bis 7 enthalten die Sedezimalziffer F_{16} (IBM /370) bzw. einen beliebigen Wert (INTEL 80x86). Das Vorzeichen wird bei IBM /370 im Byte der niedrigstwertigen Stelle kodiert, und zwar wird anstelle der Sedezimalziffer F_{16} bei einer positiven Zahl die Sedezimalziffer C_{16} genommen, bei einer negativen Zahl anstelle der Sedezimalziffer F_{16} die Sedezimalziffer D_{16}. Die INTEL 80x86-Architektur unterstützt kein Vorzeichen.

Eine *n*-stellige Zahl belegt *n* Bytes, so daß Zahlen beliebiger Größe darstellbar sind.

4.1.2 Datentyp Zeichenkette (String)

Der Datentyp **Zeichenkette** (**String**) stellt eine Aneinanderreihung von Bytes zur Verfügung, die einen beliebigen Inhalt aufnehmen können. Meist enthält eine Kette die rechnerinterne Darstellung druckbarer Zeichen, wobei unterschiedliche Zeichenkodierungen verwendet werden. Die IBM /370-Architektur setzt standardmäßig den **EBCDI-Code** (extended binary coded decimal interchange code) zur Darstellung von Zeichen ein, in der INTEL 80x86-Architektur wird der **ASCII-Code** (American standard code for information interchange) verwendet (die entsprechenden Codetabellen findet man in Kapitel 13.2).

4.1.3 Datentyp Binärwert

Neben den in druckbaren Zeichenketten vorkommenden Bitkombinationen gibt es Bitkombinationen in einem Byte, denen keine Bedeutung fest zugeordnet sind und die weder als Zeichen einer druckbaren Zeichenkette noch als Zahlen sinnvoll interpretiert werden können. Ein Datenobjekt vom Datentyp **Binärwert** kann als Wert eine beliebige Bitkombinationen annehmen und eine Anzahl von Bytes belegen, die durch die Definition des Datenobjekts festgelegt ist. Die Interpretation hängt von der Anwendung ab. Daher stellt der Datentyp Binärwert den allgemeinsten Datentyp dar.

Je nach Interpretation der vorkommenden Bitkombinationen spricht man von **Sedezimalkonstanten** bzw. **Binärkonstanten**.

4.1.4 Datentyp Adresse

Der Datentyp **Adresse** dient der Darstellung virtueller Adressen in einem Programm. Dieser Datentyp ist Rechnertyp- bzw. Speichermodell-spezifisch. Beispielsweise verwenden die in Kapitel 3.3 beschriebenen Speichermodelle des INTEL 80x86-Rechners sehr unterschiedliche Arten der internen Adreßdarstellung und -interpretation. Dabei wird noch zwischen Near und Far Pointern unterschieden. Die IBM /370-Architektur wiederum verwendet in Maschinenbefehlen ein Adreßformat, das sich aus zwei Teilen zusammensetzt: einer CPU-Registernummer (0_{10}, ..., 15_{10}) und einem 12 Bits langen als Displacement bezeichneten Offset. Für die CPU-Registernummer sind 4 Bits erforderlich, so daß eine derartige Adresse 2 Bytes belegt. Der Inhalt des angesprochenen Registers wird als 31-, 25- oder 24-Bitadresse, je nach Maschinentyp, interpretiert, zu der der Offset hinzuaddiert wird und so eine lineare virtuelle Adresse liefert. Der Anwender ist dafür verantwortlich, daß das CPU-Register mit einem "sinnvollen" Wert geladen wird. Auf weitere Details soll hier mit Verweis auf Abbildung 4.1.4-1 verzichtet werden.

<table>
<tr><td colspan="3">INTEL 80x86 (vgl. Kapitel 3.3)</td></tr>
<tr><td></td><td>Near Pointer</td><td>Far Pointer</td></tr>
<tr><td>Real Mode</td><td>Format:
[offset] 16 Bits</td><td>Format:
[segmentnummer] 16 Bits
[offset] 16 Bits</td></tr>
<tr><td>offset = Adresse innerhalb eines Segments

physikalische Adresse = Segmentbasisadresse + offset (20 Bits)</td><td>Die 16 höherwertigen Bits der Segmentbasisadresse werden zur Laufzeit aus dem Inhalt eines Segmentregisters der CPU (z.B. CS, DS, SS) genommen und um 4 binäre 0_2 rechts ergänzt (16 + 4 Bits)</td><td>Segmentbasisadresse
= segmentnummer * 16
(20 Bits);
durch den Ladevorgang sind zur Laufzeit alle Segmentnummern der EXE-Datei um die Startsegmentnummer des Programms erhöht worden</td></tr>
<tr><td>Protected Mode</td><td>Format:
[offset] 32 Bits</td><td>Format:
[segmentselektor] 16 Bits
[offset] 32 Bits</td></tr>
<tr><td>offset = Adresse innerhalb eines Segments

physikalische Adresse = Segmentbasisadresse + offset (32 Bits)</td><td>Die Segmentbasisadresse steht in einem Deskriptorregister der CPU (zusammen mit weiteren Informationen)</td><td>Der Segmentselektor wird als Index in eine Deskriptortabelle interpretiert; der so indizierte Eintrag enthält (mit weiteren Informationen) die Segmentbasisadresse</td></tr>
<tr><td>flaches Speichermodell

physikalische Adresse = Ergebnis des auf die Linearadresse angewendeten Pagingverfahrens (32 Bits)</td><td colspan="2">Format:
[linearadresse] 32 Bits</td></tr>
<tr><td colspan="3">IBM /370</td></tr>
<tr><td></td><td colspan="2">Format:
[registernummer] 4 Bits
[displacement] 12 Bits</td></tr>
<tr><td>physikalische Adresse = Ergebnis des auf die Linearadresse angewendeten Pagingverfahrens (31 bzw. 25 bzw. 24 Bits)</td><td colspan="2">Zum Inhalt des durch die Registernummer bestimmten CPU-Registers (31 bzw. 25 bzw. 24 Bits je nach Maschinentyp) wird das Displacement addiert, so daß eine virtuelle lineare Adresse entsteht.</td></tr>
</table>

Abbildung 4.1.4-1: Datentyp Adresse

Auf Anwenderebene und auch in der systemnahen Programmierung mit Hilfe höherer Programmiersprachen kann man sich meist auf die Behandlung von Adressen als Pointer (siehe Kapitel 4.2.4) beschränken.

4.1.5 Datentyp Instruktion

Der Datentyp **Instruktion** beschreibt die interne Darstellung einer Maschineninstruktion. Diese besitzen ein festes Format (meist mehrere Bytes, weiter unterschieden nach Instruktionstypen), in denen die jeweilige Maschineninstruktion und die beteiligten Operanden kodiert sind. Operanden werden über virtuelle Adressen, als Direktwerte, über Registerbezeichnungen der CPU usw. angesprochen. Eine weiterführende Behandlung des Themas Instruktionen ist der Assemblerprogrammierung vorbehalten (vgl. auch Kapitel 3.1).

4.2 Datentypen in höheren Programmiersprachen

In diesem Kapitel werden die Definitionsmöglichkeiten für Datenobjekte und ihre Abbildung auf die interne Darstellung am Beispiel der höheren Programmiersprache Pascal, insbesondere des Sprachdialekts Borland Pascal (PASCAL), näher betrachtet.

Während sich Datendefinitionen in einigen höheren Programmiersprachen sehr an der internen Datendarstellung orientieren (z.B. bei COBOL denke man an die `PICTURE`- und `USAGE`-Klauseln und ihre Abbildung auf interne Datenobjekte, die direkt auf Assembler-Deklarationen der IBM /370-Architektur übertragbar sind, vgl. [HOF]), abstrahiert Pascal weitgehend von der zugrundeliegenden Rechnerarchitektur und gewährleistet damit eine Portabilität, zumindest auf Quellprogrammebene. Die Möglichkeiten der Datendefinition in einem Pascal-Programm orientieren sich wesentlich an den Belangen der Anwendungen als an den Konzepten, die von einer Hardware-Architektur bereitgestellt werden. In der Praxis jedoch, insbesondere bei Problemen der systemnahen Programmierung, ist eine Hinwendung zu mehr rechnerorientierten Deklarationsmöglichkeiten eventuell erwünscht, und Borland Pascal (PASCAL) erscheint hier als ein möglicher Kompromiß.

Wie bereits anfangs beschrieben besteht die übliche Art, ein Datenobjekt in einer höheren Programmiersprache zu deklarieren, in der Angabe des Typs des Objekts und damit implizit der zulässigen Operationen und Methoden, die auf das Datenobjekt dieses Typs anwendbar sind, und der Angabe des Bezeichners für das Datenobjekt, etwa in der Form:

```
TYPE data_typ = ...;

VAR  datenobjekt : data_typ;
```

Zunächst sollen die in PASCAL vordefinierten Datentypen an Beispielen erläutert werden. Das Syntaxdiagramm ([BP7]) der Typdeklaration zeigt Abbildung 4.2-1. Offensichtlich kann ein definierter Typ mittels seines Typbezeichners für weitere Definitionen verwendet werden. Außerdem gibt es durch Schlüsselworte der Sprache vordefinierte Typbezeichner.

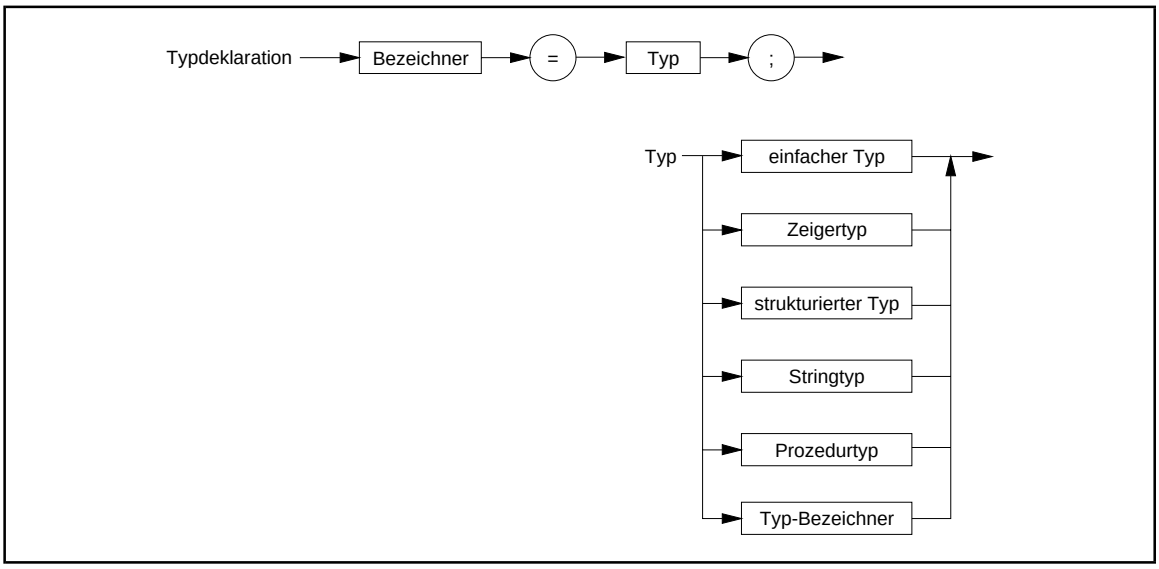


Abbildung 4.2-1: Typdeklaration in PASCAL

Wegen ihrer besonderen Bedeutung werden die strukturierten Datentypen der objektorientierten Programmierung in gesonderten Kapitel behandelt.

4.2.1 Einfacher Typ

Das Sytaxdiagramm einfacher Typen ist in Abbildung 4.2.1-1 zu sehen.

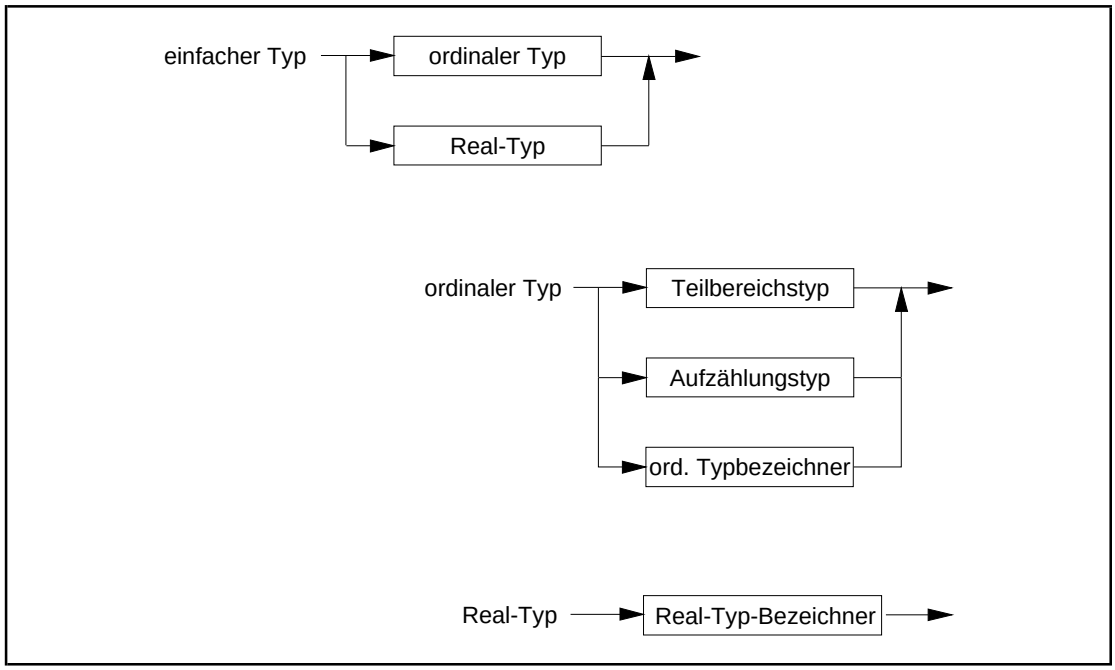


Abbildung 4.2.1-1: Einfacher Typ

Die Werte eines Datenobjekts vom Datentyp **Ordinaltyp** bilden eine geordnete Menge. Jedem Wert ist eine ganzzahlige Ordinalzahl zugeordnet. Mit Ausnahme von Werten des Ordinaltyps Integer hat der erste Wert eines Ordinaltyps die Ordinalzahl 0, der nächste die Ordinalzahl 1 usw. Ein Wert des Ordinaltyps Integer entspricht der bezeichneten ganzen Zahl.

Die Pascal-Standardfunktion **`Ord`** kann auf den Wert eines Datenobjekts vom Datentyp Ordinaltyp angewendet werden und liefert dessen Ordinalzahl. Entsprechend liefert die Pascal-Standardfunktion **`Succ`** die nachfolgende Ordinalzahl des Werts eines Datenobjekts. Mit der Pascal-Standardfunktion **`Pred`** kann man die Ordinalzahl des Vorgängers eines Werts eines Datenobjekts erhalten, falls dieser existiert; ansonsten kommt es zu einem Laufzeitfehler. Zusätzlich gibt es die Standardfunktionen **`Low`** und **`High`**, die den niedrigsten bzw. höchsten Wert im Bereich des Ordinaltyps anzeigen, soweit diese definiert sind.

Folgende Ordinaltypen sind vordefiniert: Integer-Datentypen (`ShortInt`, `INTEGER`, `LongInt`, `Byte`, `Word`), Boolesche Datentypen (`BOOLEAN`, `ByteBool`, `WordBool`, `LongBool`), `CHAR`, Aufzählungstyp und Teilbereichstyp[19].

Ein Datenobjekt mit **Integer-Datentyp** (**`ShortInt`**, **`INTEGER`**, **`LongInt`**, **`Byte`**, **`Word`**) wird intern auf eine Festpunktzahl (mit Vorzeichen) bzw. eine Ordinalzahl (ohne Vorzeichen) abgebildet. Entsprechend gilt für den jeweiligen **Wertebereich**:

Typ	internes Format (INTEL 80x86)	Wertebereich	interner Datentyp (INTEL 80x86)
`ShortInt`	Byte	-128, ..., 127	Festpunktzahl
`INTEGER`	Wort	-32.768, ..., 32.767	Festpunktzahl
`LongInt`	Doppelwort	-2.147.483.648, ..., 2.147.483.647	Festpunktzahl
`Byte`	Byte	0, ..., 255	Ordinalzahl
`Word`	Wort	0, ..., 65.535	Ordinalzahl

Sind an einer Operation mehrere Variablen unterschiedlicher Integer-Datentypen beteiligt, so erfolgt intern zunächst eine automatische Umwandlung auf den "kleinsten" Datentyp, der die beteiligten Datentypen enthält gemäß den Regeln

```
ShortInt ≤ INTEGER ≤ LongInt,
ShortInt ≤ Word,
Byte ≤ INTEGER,
Byte ≤ Word.
```

[19] Zur Schreibweise ist die Bemerkung am Ende von Kapitel 1 zu beachten.

Werte, die ein einzelnes Byte Speicherplatz belegen, werden intern vor Ausführung der Operation vorzeichenrichtig auf 16 Bits erweitert und sind damit zu `INTEGER`- und `Word`-Operanden kompatibel.

Die in arithmetischen Ausdrücken **mit Variablen vom Integer-Datentyp zulässigen Operatoren** werden in Abbildung 4.2.1-2 zusammengefaßt (die Abbildung enthält auch die Angaben für Real-Datentypen). Dabei steht wie üblich ein binärer Operator zwischen zwei Operanden, ein unärer Operator vor einem einzelnen Operator.

Binäre Operatoren				
Operator	Operation	Operandentyp	Ergebnistyp	Bemerkung
`+ bzw. - bzw. *`	Addition bzw. Subtraktion bzw. Multiplikation	Integer Real	Integer Real	
`/`	Division	Integer Real	Real Real	Laufzeitfehler bei Division durch 0
`DIV`	ganzzahlige Division	Integer	Integer	*i* DIV *j* ist gleich $\lfloor i/j \rfloor$; Laufzeitfehler bei $j = 0$
`MOD`	Modulo	Integer	Integer	*i* MOD *j* ist gleich *i* - (*i* DIV *j*) * *j*
Unäre Operatoren				
`+`	Identität	Integer Real	Integer Real	
`-`	Negation	Integer Real	Integer Real	
Logische Operatoren				
`NOT`	bitweise Negation	Integer	Integer	`NOT` $0_2 = 1_2$, `NOT` $1_2 = 0_2$
`AND`	bitweises UND	Integer	Integer	0_2 `AND` $0_2 = 0_2$, 0_2 `AND` $1_2 = 0_2$, 1_2 `AND` $0_2 = 0_2$, 1_2 `AND` $1_2 = 1_2$
`OR`	bitweises ODER	Integer	Integer	0_2 `OR` $0_2 = 0_2$, 0_2 `OR` $1_2 = 1_2$, 1_2 `OR` $0_2 = 1_2$, 1_2 `OR` $1_2 = 1_2$
`XOR`	bitweises EXKLUSIVES ODER	Integer	Integer	0_2 `XOR` $0_2 = 0_2$, 0_2 `XOR` $1_2 = 1_2$, 1_2 `XOR` $0_2 = 1_2$, 1_2 `XOR` $1_2 = 0_2$

SHL	bitweise Linksverschiebung	Integer	Integer	*i* SHL *j* verschiebt den Wert von *i* um soviele Bitpositionen nach links, wie der Wert von *j* angibt, d.h. *i* SHL *j* ist gleich $i \cdot 2^j$
SHR	bitweise Rechtsverschiebung	Integer	Integer	*i* SHR *j* verschiebt den Wert von *i* um soviele Bitpositionen nach rechts, wie der Wert von *j* angibt, d.h. *i* SHR *j* ist gleich *i* DIV 2^j
Die mit Integer- und Real-Datentypen erlaubten relationalen Operatoren (Vergleichsoperatoren) werden insgesamt mit den Vergleichsoperatoren aller Datentypen in Kapitel 4.2.7 behandelt.				

Abbildung 4.2.1-2: Operatoren mit Integer- und Real-Datentypen

Der Datentyp **Aufzählungstyp** dient der Darstellung geordneter Mengen. Er wird durch die Benennung der einzelnen Elemente definiert, wobei die Ordinalzahlen der Elemente durch die Reihenfolge festgelegt werden. Durch die Deklaration sind die Elemente ebenfalls als Konstanten festgelegt. Beispielsweise definiert der Datentyp

```
studiengang_typ = (Informatik, BWL, Mathematik,
                   sonstiger);
```

einen Aufzählungstyp und die drei Konstanten `Informatik`, `BWL` und `Mathematik`; dabei gilt `Ord(Informatik) = 0, Ord(BWL) = 1, Ord(Mathematik) = 2` und `Ord(sonstiger) = 3`.

Die interne Darstellung eines Datenobjekts mit Aufzählungstyp ist ein Byte, wenn die Aufzählung weniger als 256 Elemente enthält, sonst ein Wort (2 Bytes). Jeder Wert eines Datenobjekts von einem Aufzählungstyp wird intern durch seine Ordinalzahl repräsentiert.

Boolesche Datentypen sind **`BOOLEAN, ByteBool, WordBool, LongBool`** mit der internen Darstellung in einem Byte (`BOOLEAN, ByteBool`) bzw. einem Wort (`WordBool`) bzw. einem Doppelwort (`LongBool`). Der in allen Anwendungen vorzuziehende Boolesche Datentyp ist `BOOLEAN`, der als Aufzählungstyp durch

```
TYPE BOOLEAN = (FALSE, TRUE);
```

definiert ist. Bei allen anderen Booleschen Datentypen wird der ordinale Wert 0 als `FALSE`, ein ordinaler Wert ungleich 0 als `TRUE` interpretiert (eine Reminiszenz an die Sprache C und an WINDOWS).

Datenobjekte mit Booleschem Datentyp können durch **logische Operatoren** miteinander verknüpft werden (Abbildung Abbildung 4.2.1-3).

Operator	Operation	Operanden- und Ergebnistyp	Bemerkung
`NOT`	logische Negation	Boolesch	`NOT FALSE = TRUE,` `NOT TRUE  = FALSE,`

`AND`	logisches UND	Boolesch	`FALSE AND FALSE = FALSE,`
			`FALSE AND TRUE  = FALSE,`
			`TRUE  AND FALSE = FALSE,`
			`TRUE  AND TRUE  = TRUE`

`OR`	logisches ODER	Boolesch	`FALSE OR FALSE = FALSE,`
			`FALSE OR TRUE  = TRUE,`
			`TRUE  OR FALSE = TRUE,`
			`TRUE  OR TRUE  = TRUE`

`XOR`	logisches EXKLUSIVES ODER	Boolesch	`FALSE XOR FALSE = FALSE,`
			`FALSE XOR TRUE  = TRUE,`
			`TRUE  XOR FALSE = TRUE,`
			`TRUE  XOR TRUE  = FALSE`

Abbildung 4.2.1-3: Logische Operatoren für Boolesche Datentypen

Ein **Teilbereichstyp** umfaßt einen ordinalen Wertebereich und wird durch die Angaben des niedrigsten und des höchsten Werts (jeweils als Konstanten) festgelegt, z.B. ist nach Definition des obigen Aufzählungstyps studiengang_typ die Definition des Teilbereichstyps

```
stud_gang_typ = BWL .. sonstiger
```

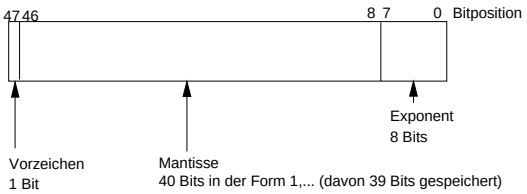
möglich. Die interne Darstellung hängt von den Werten der unteren und oberen Grenze ab; alle Werte eines Teilbereichtyps müssen darstellbar sein.

Mit Hilfe der Pascal-Standardfunktion **`Chr`**, die auf den `INTEGER`-Werten 0 bis 255 definiert ist und das der jeweiligen Zahl zugeordnete Zeichen (EBCDI- oder ASCII-Code, je nach Rechnertyp) liefert, ist der Datentyp **`CHAR`** als Teilbereichstyp durch

```
CHAR = Chr(0) .. Chr(255);
```

definiert.

Vom Datentyp **Real-Typ** gibt es die durch **`REAL`**, **`Single`**, **`Double`**, **`Extended`** und **`Comp`** bezeichneten Arten. Ein Datenobjekt vom Typ `Single` bzw. `Double` bzw. `Extended` wird intern als eine Gleitpunktzahl im kurzen bzw. langen bzw. erweiterten Format behandelt. Daraus ergeben sich für diese Teiltypen die in Abbildung 4.1.1-1 angegebenen möglichen Wertebereiche und die entsprechenden Genauigkeiten der Annäherung an eine reelle Zahl. Ein Datenobjekt vom Typ `REAL` belegt intern 6 Bytes (48 Bits), und zwar 1 Bit für das Vorzeichen, 8 Bits für die Charakteristik (biased Exponent) und 39 Bits für die Mantisse (Signifikant), siehe Abbildung 4.2.1-4.



	interne Darstellung von REAL
Breite insgesamt (Bits)	48
Breite des Signifikanden (Bits)	39 + 1*
Exponentenbreite (Bits)	8
maximaler Exponent	126
minimaler Exponent	-128
Bias	129

absoluter Wertebereich W (normalisiert[20])
$2{,}9387_{10} \cdot 10^{-39} < W < 1{,}7015_{10} \cdot 10^{38}$
kleinstmöglicher Abstand Δ_{min} zwischen zwei Zahlen
$\approx 5{,}3455_{10} \cdot 10^{-51}$

* Die führende 1_2 wird nicht mit gespeichert

Abbildung 4.2.1-4: Datentyp REAL (Borland Pascal)

Der Bias ist so gewählt, daß die Charakteristik c einer von 0 verschiedene REAL-Zahl die Bedingung $0 < c < 255$ erfüllt. Jede Charakteristik $c = 0$ führt zur Interpretation 0 des Werts für das Datenobjekt. Der Datentyp REAL kennt weder nicht-normalisierte Zahlen noch NaNs oder Unendlichkeiten (siehe Kapitel 4.1.1): nicht-normalisierte Zahlen werden auf 0 gesetzt, NaNs und Unendlichkeiten erzeugen einen Überlauf.

Die Operatoren, die für Datenobjekte mit Real-Typ zulässig sind, umfassen die üblichen arithmetischen und relationalen Operatoren und sind der Abbildung 4.2.1-2 bzw. Kapitel 4.2.5 zu entnehmen.

4.2.2 Stringtyp

Für die Behandlung variabel langer Zeichenketten ist der Datentyp **`STRING`** vorgesehen. Die Deklaration

```
VAR data : STRING [n];
```

[20] An der Obergrenze ist im Faktor die letzte Dezimalstelle nach dem Komma des Faktors aufgerundet, an der Untergrenze abgerundet.

definiert ein Datenobjekt, dessen Werte Zeichenketten der maximalen Länge von *n* Zeichen sein können. Fehlt die Angabe von *n*, so wird implizit der maximal mögliche Wert für *n* genommen (bei Borland Pascal *n* = 255). Zur internen Darstellung des Datenobjekts werden *n*+1 aufeinanderfolgende Bytes belegt, die von 0 bis *n* indiziert sind. Das Byte an der Position 0 (an der niedrigsten Adresse) enthält als (vorzeichenlose) Ordinalzahl die Länge der gerade im Datenobjekt gespeicherten Zeichenkette. Die Standardfunktion **`Length`** liefert diese Längenangabe. Die Zeichen der Zeichenkette belegen die Bytes mit Index 1 bis *n* (aufsteigende Adressen).

Zwei Zeichenketten sind gleich, wenn ihre aktuellen Längen und ihre Werte übereinstimmen. Eine Zeichenkette *X* ist kleiner als eine Zeichenkette *Y*, wenn entweder die Länge von *X* kleiner als die Länge von *Y* ist oder wenn sie bei gleicher Länge bis zu einer Position *k* dieselben Zeichen enthalten und der (ASCII-) Code des Zeichens an der Position *k*+1 von *X* kleiner ist als der entsprechende Code an der Position *k*+1 von *Y*. Das kleinste Datenobjekt mit `STRING`-Typ ist ein **Nullstring**, d.h. eine Zeichenkette der Länge 0.

Eine Reihe von Standardfunktionen zur Manipulation von Datenobjekten mit `STRING`-Typ definieren Funktionen zur Aneinanderreihung von Zeichenketten (Konkatination), das Entfernen von Teilen aus einer Zeichenkette, das Einfügen von Zeichenketten in eine andere ab einer gegebenen Position und das Suchen der kleinsten Position eines vorgegebenen Zeichens in einer Zeichenkette. Die Konkatination zweier Zeichenketten bewirkt auch durch den Operator +. Die Anweisungen

```
xzeichen := '123 A';
xzeichen := '### ' + xzeichen + '???';
```

setzen die Variable `xzeichen` mit `STRING`-Typ auf den Wert ### 123 A???.

Eine weitere Möglichkeit der Deklaration eines Datenobjekts mit Zeichenkettenwerten ist durch

```
VAR data : PACKED ARRAY [index_typ] OF CHAR;
```

gegeben, wobei `index_typ` ein Teilbereichstyp ist. Die interne Darstellung besteht aus einem Datenobjekt mit Datentyp Zeichenkette, die soviele Bytes belegt, wie die Differenz der Ober- und Untergrenze von index_typ angibt.

Die Deklaration

```
VAR data : ARRAY [0..idx] OF CHAR;
```

definiert ein Datenobjekt mit **nullbasierten Zeichenkettentyp**, wobei `idx` ein ganzzahliger von 0 verschiedener Ordinalwert ist. Dieses Datenobjekt kann eine **nullterminierte Zeichenkette** aufnehmen, d.h. eine Zeichenkette, deren Ende durch das Zeichen

#0 definiert ist. Eine derartige Zeichenkette kann eine maximale Länge von 65.535 Zeichen haben. Spezielle Standardfunktionen erlauben Operationen mit nullterminierten Zeichenketten und Datenobjekten (vgl. [BP7]).

4.2.3 Strukturierter Typ

Aufbauend auf den vordefinierten Grunddatentypen können anwendungsbezogen komplexe strukturierte Datentypen deniert werden. Ein **strukturierter Datentyp** setzt sich aus Komponenten zusammen, die selbst wieder strukturiert sein können, wobei dann deren Deklaration vorher erfolgt sein muß. Es sind beliebig viele Strukturierungsebenen möglich.

Abbildung 4.2.3-1 faßt die strukturierten Datentypen zusammen; der `OBJECT`-Typ wird erst in Kapitel 9 im Zusammenhang mit der objektorientierten Programmierung behandelt. Das vordefinierte Schlüsselwort **`PACKED`** bewirkt eine möglichst kompakte interne Speicherung (die bei Borland Pascal auch ohne dieses Schlüsselwort eingehalten wird). Die folgende Darstellung gibt lediglich einen Überblick über die Deklarationsmöglichkeiten strukturierter Datentypen. Details können in den angegebenen Handbüchern nachgeschlagen werden.

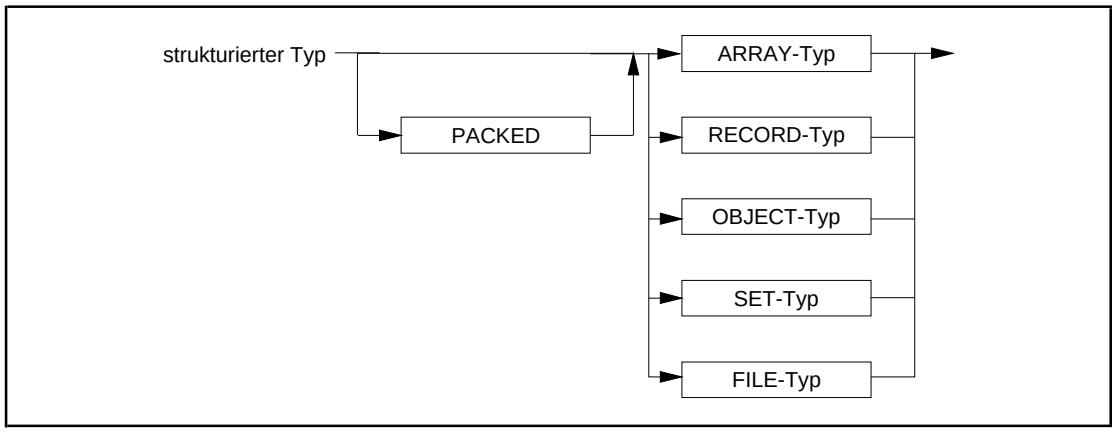


Abbildung 4.2.3-1: Strukturierte Datentypen

Ein **`RECORD`-Typ** enthält eine festgelegte Anzahl an Komponenten. Die Deklaration legt die Typen und die Bezeichner für jede Komponente fest. Er enthält einen **invarianten Teil**, an den sich eventuell ein **varianter Teil** anschließt. Formal wird ein `RECORD`-Typ mit *n* Komponeten durch

```
TYPE data_typ = RECORD
                  komponente_1 : datentyp_1;
                  ...
                  komponente_n : datentyp_n
                END;
```

definiert. Ein Datenobjekt mit RECORD-Typ wird intern als Folge seiner Komponenten behandelt. Enthält ein RECORD-Typ einen varianten Teil, so ergibt sich der interne Speicherbedarf eines Datenobjekts mit diesem RECORD-Typ aus der Summe des invarianten und der größten Variante

Im folgenden Beispiel dient der durch

```
TYPE funktions_typ = (Verwaltungsmitarbeiter,
                      gewerblicher_Mitarbeiter,
                      leitender_Angestellter);
```

deklarierte Aufzählungstyp der Klassifizierung der Funktionen eines Mitarbeiters in einem Unternehmen. Als mögliche Personalnummer eines Mitarbeiters sei der Zahlbereich 10000 bis 99999 zugelassen:

```
TYPE personalnr_bereich = 10000..99999.
```

Um die Höhe des Gehalts eines Mitarbeiters, seines Krankenkassenbeitrags und den Namen der Krankenkasse abzulegen, der er angehört (wobei ein leitender Angestellter lediglich einen Krankenkassenzuschuß erhält, ohne den Namen der Krankenkasse festzuhalten), wird der RECORD-Typ

```
TYPE ma_typ = RECORD
                  { invarianter Teil: }
                personalnummer : personalnr_bereich;
                gehalt         : REAL;
                  { varianter Teil: }
                CASE funktion: funktions_typ OF
                  Verwaltungsmitarbeiter,
                  gewerblicher_Mitarbeiter:
                    (beitrag      : REAL;
                     krankenkasse : STRING [15]);
                  leitender_Angestellter:
                    (zuschuss     : REAL)
              END;
```

deklariert. Den durch

```
VAR mitarbeiter_1, mitarbeiter_2 : ma_typ;
```

deklarierten Variablen werden Werte zugewiesen:

```
WITH mitarbeiter_1 DO
  BEGIN
    personalnummer := 12000;
    gehalt         := 2940.00;
    funktion       := gewerblicher_Mitarbeiter;
    beitrag        := 338.10;
    krankenkasse   := 'XYZ Kasse'
  END;
```

```
WITH mitarbeiter_2 DO
  BEGIN
    personalnummer := 25000;
    gehalt         := 9500.00;
    funktion       := leitender_Angestellter;
    zuschuss       := 110.00
  END;
```

Ein **ARRAY-Typ** spezifiziert die Struktur eines Felds, indem der Datentyp der einzelnen Feldelemente, deren Anzahl (in Form einer Dimensionsangabe mit unterer und oberer Indexgrenze für jede Dimension) und der Datentyp der Indizes (Ordinaltyp außer `LongInt`) festgelegt wird. Pascal akzeptiert nur statische Felder, so daß der Speicherplatzbedarf eines Datenobjekts mit ARRAY-Typ zur Übersetzungszeit festliegt[21].

Ein **mehrdimensionales Feld** kann als Datenobjekt mit (eindimensionalen) ARRAY-Typ definiert werden, deren Feldelemente wieder vom ARRAY-Typ sind. So führen die in der folgenden Deklaration vorkommenden ARRAY-Typen `feld_typ_1` und `feld_typ_2` zu Datenobjekten mit derselben internen Darstellung.

```
TYPE  anzahl = 1..10;

      feld_typ_1 = ARRAY [anzahl] OF
                     ARRAY [BOOLEAN] OF
                       ARRAY [1..10] OF REAL;
      feld_typ_2 = ARRAY [anzahl, BOOLEAN, 1..10] OF REAL;
```

Ein Datenobjekt mit ARRAY-Typ wird intern durch die Folge der einzelnen Feldelemente realisiert. Die interne Speicherung eines mehrdimensionalen Felds ergibt sich aus der Definition als zusammengesetzter ARRAY-Typ niedrigerer Dimensionen.

Ein Datenobjekt mit **SET-Typ** stellt eine (Teil-) Menge im mathematischen Sinn aus Elementen einer Grundmenge dar. Der Datentyp der Grundmenge ist ein Ordinaltyp, dessen Werte sich im Bereich 0..255 bewegen.

Im folgenden Beispiel werden der Ordinaltyp `farbe` einer Grundmenge und ein SET-Typ `farb_menge` definiert. Der Ordinaltyp `farbe` gibt die möglichen Elemente der Grundmenge an. Datenobjekte mit Datentyp `farb_menge`, wie die Variablen `f1, f2, regenbogen_farben`, repräsentieren Teilmengen aller möglichen Farbwerte, die in der Grundmenge vorkommen:

[21] Ein im ISO-Pascal-Standard definierter Prozedurparameter vom Datentyp **CONFORMANT ARRAY** definiert feste, aber in der Liste der Formalparameter zunächst unbestimmte Feldgrenzen, die dann bei jedem Aufruf durch die Aktualparameter determiniert werden. Es handelt sich also ebenfalls um feste Feldgrenzen (nämlich die Feldgrenzen der jeweiligen Aktualparameter), die sich je nach Prozeduraufruf unterscheiden können.

```
TYPE farbe = (schwarz, weiss, rot, orange, gelb,
              gruen, blau, indigo, violett,
              sonstige_Farbe);

     farb_menge = SET OF farbe;

VAR  f1, f2, regenbogenfarben : farb_menge;
```

Mit diesen Deklarationen sind beispielsweise die Anweisungen

```
f1               := [schwarz, weiss];
f2               := [schwarz..sonstige_Farbe];
regenbogenfarben := f2 - f1;
f1               := []    { leere Menge };
```

möglich. Die Konstruktion [...] stellt eine Konstante mit SET-Typ dar. Datenobjekte mit SET-Typ können mit den **Operatoren** +, - und * verknüpft werden:

Operator	Operation	Operandentyp
+	Vereinigung	kompatible Mengen
-	Differenz	kompatible Mengen
*	Durchschnitt	kompatible Mengen

Ein Datenobjekt mit SET-Typ und Namen `set_var` wird als Bit-Feld, das nach Bedarf eventuell mehrere Bytes umfaßt, gespeichert, wobei die Position des jeweiligen Bits (in der Zählung von rechts nach links beginnend mit Position 0) für die Ordinalität und sein Wert für das Vorhandensein des entsprechenden Elements der Grundmenge steht: Der Bitwert 0_2 besagt, daß das der Bitposition entsprechende Element der Grundmenge in `set_var` nicht vorhanden ist, der Bitwert 1_2 besagt, daß es in `set_var` vorkommt.

Es seien ord_{min} und ord_{max} die kleinste bzw. die größte Ordinalzahl innerhalb der Grundmenge. Die Anzahl benötigter Bytes zur Darstellung eines Datenobjekts mit SET-Typ errechnet sich dann zu

$$(ord_{max} \text{ DIV } 8) - (ord_{min} \text{ DIV } 8) + 1.$$

Ein Datenobjekt vom Datentyp **`FILE`-Typ**, eine Datei, ist eine sequentielle Aneinanderreihung von strukturierten Elementen, die im allgemeinen auf einem externen Speichermedium liegen. Die Syntax der Deklaration eines FILE-Typs wird an dem Beispiel

```
TYPE file_typ = FILE OF element_typ;
```

deutlich; für `element_typ` ist eine beliebige Typdeklaration erlaubt. Häufig werden jedoch keine strukturierten Komponenten zugelassen, die ihrerseits wieder FILE-Typen enthalten. Die Angabe `element_typ` kann sogar fehlen; es handelt sich dann um die Deklaration eines FILE-Typs einer **untypisierten Datei**, die bei der (maschinennahen) Eingabe- oder Ausgabeoperationen verwendet wird. Ein weiterer FILE-Typ ist der Typ

```
text
```

der eine Datei mit zeilenweise angeordneten Textfolgen, eine **Textdatei**, beschreibt.

Eine Datei unterscheidet sich von einem Datenobjekt mit ARRAY-Typ, einem Feld, das ja ebenfalls als geordnete Menge gleichartiger (meist strukturierter) Elemente angesehen werden kann, im wesentlichen in zwei Punkten: Die Größe eines Felds ist statisch, während die Anzahl der Elemente einer Datei unbegrenzt ist; die Zugriffszeit auf ein Feldelement ist nicht abhängig von der Position (dem Index) innerhalb des Felds, während der Zugriff auf Dateielemente sequentiell erfolgt.

Die interne Darstellung eines Datenobjekts vom Datentyp FILE-Typ hängt natürlich wesentlich vom Rechnertyp ab. Borland Pascal bildet eine Variable mit FILE-Typ auf ein Datenobjekt mit RECORD-Typ ab: Eine Dateivariable mit FILE-Typ wird auf eine interne Variable mit folgendem Typ umgesetzt:

```
TYPE FileRec = RECORD
                 Handle   : Word;
                 Mode     : Word;
                 RecSize  : Word;
                 Private  : ARRAY [1..26] OF Byte;
                 UserData : ARRAY [1..16] OF Byte;
                 Name     : ARRAY [1..79] OF CHAR;
               END;
```

Bei einer Textdatei lautet die interne Darstellung

```
TYPE TextBuf = ARRAY[0..127] OF CHAR;
     TextRec = RECORD
                 Handle    : Word;
                 Mode      : Word;
                 BufSize   : Word;
                 Private   : Word;
                 BufPos    : Word;
                 BufEnd    : Word;
                 BufPtr    : ^TextBuf;
                 OpenFunc  : Pointer;
                 InOutFunc : Pointer;
                 FlushFunc : Pointer;
                 CloseFunc : Pointer;
                 UserData  : ARRAY [1..16] OF Byte;
                 Name      : ARRAY [1..79] OF CHAR;
                 Buffer    : TextBuf;
               END;
```

(Die Datentypen Pointer und ^TextBuf definieren Zeigertypen, die in Kapitel 4.2.4 genauer beschrieben werden.)

Wenn die Datei geöffnet ist, enthält die Komponente Handle die von MS-DOS vergebene Datei-Referenznummer (File Handle). Die Komponente Mode zeigt den Dateityp und die Zugriffsart auf die Datei an; sie enthält einen der Werte

```
CONST fmClosed  = $D7B0;   { Datei ist geschlossen }
      fmInput   = $D7B1;   { Textdatei; geöffnet nur zum Lesen }
      fmOutput  = $D7B2;   { neu angelegte Textdatei;
                                    geöffnet; nur zum Schreiben }
      fmInOut   = $D7B1;   { typisierte oder untypisierte Datei;
                                    geöffnet zum Lesen und Schreiben }
```

Die Komponente `UserData` steht dem Programmierer für eigene Zwecke zur Verfügung. Die Komponente `Name` enthält den Dateinamen als nullterminierte Zeichenkette. `RecSize` enthält die Größe (in Bytes) der Komponenten der Datei. Die Komponente `Private` ist reserviert. In Textdateien stellt der Inhalt der Komponente `BufPtr` einen Adreßverweis auf den Puffer mit der durch `BufSize` bestimmten Größe dar. `BufPos` enthält den Index des nächsten zu lesenden bzw. zu schreibenden Zeichens innerhalb dieses Puffers, `BufEnd` die Gesamtzahl der momentan im Puffer abgelegten Zeichen. Die Adreßverweise in den Komponenten `OpenFunc, InOutFunc, FlushFunc` und `CloseFunc` zeigen auf Routinen, über die die jeweilige Funktion ausgeführt wird (siehe [BP7]).

4.2.4 Zeigertyp

Datenobjekte, die über die bisher beschriebenen Datentypen deklariert werden, belegen intern eine durch den jeweiligen Datentyp bestimmte Anzahl von Bytes. Diese Speicherplatzbelegung erfolgt automatisch ohne Zutun des Anwenders, sobald der Block (Prozedur, Programmteil auf HOL-Ebene), der die Deklaration enthält, "betreten" wird. Selbst wenn ein Datenobjekt dann aufgrund der Programmlogik während der Laufzeit gar nicht verwendet wird, belegt es trotzdem Speicherplatz. Eine Möglichkeit, **einem Datenobjekt Speicherplatz dynamisch erst während der Laufzeit nach Bedarf zuzuweisen bzw. zu entziehen** und damit seine Lebensdauer in Abhängigkeit von der Anwendung zu begrenzen, besteht in der Verwendung von Zeigertypen und einem "indirekten Zugriff" auf das jeweilige Datenobjekt. Dazu wird neben dem betreffenden Datenobjekt `D` ein weiteres Datenobjekt `D_Ptr` angelegt, das die Adresse von `D` aufnimmt, sobald `D` auch wirklich benötigt und Speicherplatz bereitgestellt wird. Der Wert von `D_Ptr` wird als **Zeiger** (**Pointer**) auf `D` bezeichnet; der Datentyp von `D_Ptr` ist ein Zeigertyp.

Ein Datenobjekt mit Datentyp **Zeigertyp** (**Pointertyp**) kann als Wert eine Adresse, d.h. eine Referenz auf ein weiteres Datenobjekt, oder den Wert **`NIL`** enthalten. Der Wert `NIL` repräsentiert hier einen Adreßverweis, der "nirgendwo hinzeigt" (undefinierte Adresse). Intern belegt ein Datenobjekt mit Zeigertyp soviele Bytes, wie zur internen Darstellung einer Adresse notwendig sind.

Die Syntax des Datentyps Zeigertyp wird an folgenden Beispiel deutlich:

```
TYPE pointertyp = ^grund_typ;
          { ^ bedeutet Verweis auf ein Datenobjekt,
            das den auf ^ folgenden Datentyp besitzt }
     grund_typ  = ...;

VAR  datenobjekt : pointertyp;
```

Das Datenobjekt mit Bezeichner `datenobjekt` und dem Zeigertyp `pointertyp` kann die Adresse eines anderen Datenobjekts aufnehmen, das den Datentyp `grund_typ` hat, oder den Wert `NIL`.

Für ein so deklariertes Datenobjekt mit Pointertyp sind nur wenige **Operationen** zugelassen:

- Zuweisung der Adresse eines Datenobjekts mit Datentyp grund_typ oder die Zuweisung des Werts `NIL`

- Vergleich auf Wertgleichheit mit einem anderen Datenobjekt mit Datentyp `pointertyp` oder mit dem Wert `NIL`

- "Freigabe" des Datenobjekts, dessen Adresse gerade als Wert in `datenobjekt` enthalten ist.

Ist das Datenobjekt mit Bezeichner `ref_data` durch

```
VAR  ref_data : pointertyp   { siehe oben };
```

definiert, so sind beispielsweise die Operationen

```
ref_data    := NIL;
datenobjekt := ref_data;
...
IF ref_data = NIL THEN ... ELSE ...;
...
IF datenobjekt <> ref_data THEN ...;
```

erlaubt.

Durch den Aufruf der Pascal-Standardprozedur **`New`** in der Form

```
New (datenobjekt);
```

wird ein neues Datenobjekt mit Datentyp `grund_typ` erzeugt und dessen Adresse (als Wert) dem Datenobjekt mit Bezeichner `datenobjekt` zugewiesen. Dieses neue Datenobjekt liegt im Heap (siehe Kapitel 3.4).

Ein **Zugriff auf den Wert** dieses neu erzeugten Datenobjekts erfolgt ausschließlich über die in `datenobjekt` abgelegte Adresse; es hat (anders als die "normalen" Datenobjekte) keinen Bezeichner. Ist der Datentyp `grund_typ` wie oben durch

```
TYPE grund_typ = RECORD
                   wert   : LongInt;
                   belegt : BOOLEAN
                 END;
```

definiert, so werden die Werte 100 und `TRUE` in die Komponenten `wert` bzw. `belegt` durch

```
datenobjekt^.wert   := 100;
datenobjekt^.belegt := TRUE;
```

gesetzt; die Anweisung

```
IF datenobjekt^.wert > 100 THEN ... ELSE ...;
```

prüft den Wert der Komponente `wert`.

Die Pascal-Standardprozedur **`Dispose`** in der Form

```
Dispose (datenobjekt);
```

gibt das Datenobjekt, dessen Adresse in `datenobjekt` steht, frei, d.h. dieses Datenobjekt existiert nach Ausführung der Prozedur `Dispose` nicht mehr.

Durch die Deklarationen

```
TYPE pointertyp = ^grund_typ;
     grund_typ  = ...;

VAR  datenobjekt : pointertyp;
```

wird also noch kein Datenobjekt mit Datentyp `grund_typ` deklariert. Erst durch den Aufruf der Prozedur

```
New (datenobjekt);
```

wird es eingerichtet, seine Adresse in `datenobjekt` abgelegt, und es beginnt seine Lebensdauer. Sie endet mit der Ausführung der Prozedur `Dispose`, der als Parameter die Adresse des Datenobjekts mitgegeben ist; ein Zugriff auf dieses Datenobjekt ist nun nicht mehr möglich.

Der Heap belegt in der Regel den gesamten restlichen verfügbaren Speicherplatz (das ist der freie Speicherplatz vor Programmstart abzüglich des Speicherplatzes für den Programmcode, den statischen Datenbereich und den Stack), oder seine Größe wird durch Steuerungsparameter bestimmt. Bei Programmstart ist der gesamte Heap als frei gekennzeichnet. Die Ausführung von `New` reserviert fest den Speicherplatz für ein neues Datenobjekt (z.B. vom Datentyp `grund_typ`). Der so reservierte Platz innerhalb des Heaps wird erst durch `Dispose` zur weiteren Verwendung wieder freigegeben. Auf diese Weise enthält der Heap reservierte und freie Bereiche, deren Größen und Verteilung sich während der Programmlaufzeit dynamisch ändern. Das Laufzeitsystem führt eine Liste der nicht-reservierten Bereiche (Anfangsadresse und Anzahl Bytes) im Heap. Die Einträge dieser Liste sind über Adreßverweise logisch miteinander verkettet und nach Lückengröße geordnet. Bei Aufruf von `New` wird zunächst versucht, für das neu einzurichtende Datenobjekt eine durch vorhergehende `Dispose`-Aufrufe entstandene Lücke zu nutzen. Schließt sich bei einem `Dispose`-Aufruf die neu entstehende Lücke nahtlos an eine bereits existierende Lücke an, so werden diese Bereiche zu einem zusammenhängenden freien Bereich zusammengelegt. Aus Performancegründen wird auf komplexere Verfahren (Garbage Collection) zur Heaporganisation meist verzichtet.

```
PROGRAM beispiel;

  TYPE pointertyp = ^grund_typ;

       grund_typ = RECORD
                     wert   : INTEGER;
                     belegt : BOOLEAN
                   END;

  VAR  datenobjekt : pointertyp;
       ref_data    : pointertyp;
  ...
  BEGIN

   New (datenobjekt);
   datenobjekt^.wert   := 100;
   datenobjekt^.belegt := TRUE;    { Bild (a) }

   New (ref_data);
   ref_data^.wert   := 200;
   ref_data^.belegt := TRUE;       { Bild (b) }

   New (datenobjekt);
   datenobjekt^.wert   := 300;
   datenobjekt^.belegt := TRUE;

   Dispose (ref_data);             { Bild (c) }

   ref_data := datenobjekt;
   New (datenobjekt);
   datenobjekt^.wert   := 400;
   datenobjekt^.belegt := TRUE;    { Bild (d) }

   ...

  END.
```

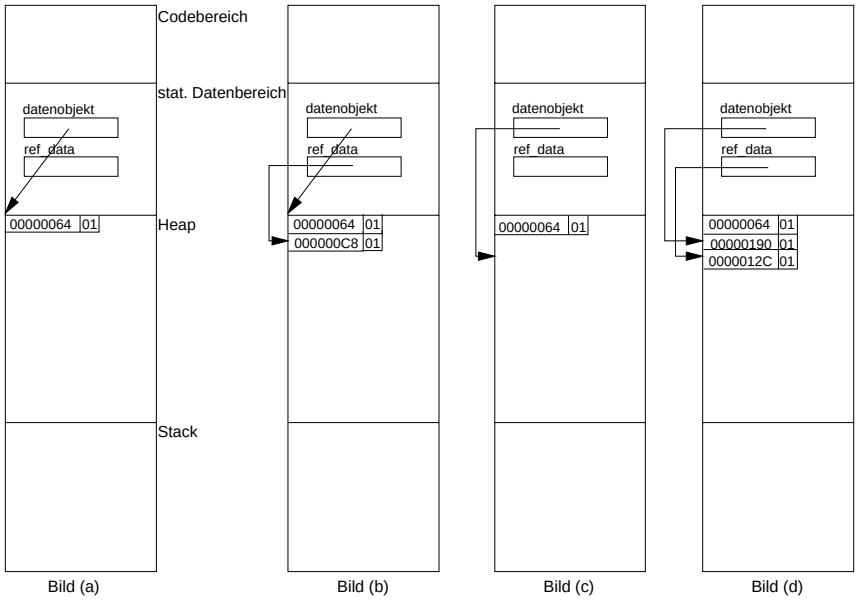


Abbildung 4.2.4-1: Belegung des Heaps (Beispiel)

Das Beispiel in Abbildung 4.2.4-1 zeigt schematisch das Zusammenspiel von `New` und `Dispose` und die Speicherbelegung bei einem Programmausschnitt. Durch eine fehlerhafte Reihenfolge im Gebrauch der Prozedur `New` (zu sehen beim Übergang von Bild (b) nach Bild (c)) ist der Adreßverweis auf das Datenobjekt mit `wert`-Komponente 100 überschrieben worden. Der Speicherplatz, den dieses Datenobjekt im Heap belegt, kann nicht mehr durch `Dispose` freigegeben werden; er bleibt während der gesamten Programmlaufzeit reserviert.

Im allgemeinen bewegt sich die Belegungsobergrenze des Heaps in Richtung auf den Stack, der seinerseits in Richtung auf den Heap wächst. Überschneiden sich Heap und Stack, kommt es zu einem Laufzeitfehler. Daher sollte man möglichst den Speicherplatz für ein durch New kreiertes dynamisches Datenobjekt mit Dispose wieder freigeben, sobald es nicht mehr benötigt wird. Andererseits ist gewisse Vorsicht geboten. Im folgenden Beispiel kommt es zu einem Laufzeitfehler, da auf ein nicht-existentes Datenobjekt zugegriffen wird.

```
PROGRAM achtung;

TYPE pointertyp = ^grund_typ;

        grund_typ = RECORD
                      wert   : INTEGER;
                      belegt : BOOLEAN
                    END;

VAR  datenobjekt : pointertyp;
     ref_data    : pointertyp;

BEGIN
  New (datenobjekt);
  ref_data := datenobjekt
            { beide Datenobjekte enthalten
              dieselben Adreßverweise        };
  Dispose (datenobjekt);
  ref_data^.wert := 100
            { das Datenobjekt, dessen Adresse
              in ref_data steht, existiert
              nicht mehr                     }
END.
```

Ein weiterer Zeigertyp (Borland Pascal) ist der Datentyp **`Pointer`**, der für einen **untypisierten Zeiger** steht. Dieser Datentyp ist wie die Konstante `NIL` mit allen Zeigertypen kompatibel. Möchte man allerdings auf ein Datenobjekt, das durch den Wert eines Datenobjekts vom Typ `Pointer` referenziert wird, zugreifen, so muß vorher eine **explizite Typisierung** vorgenommen werden. Das Vorgehen erläutert das folgende Beispiel:

```
...
TYPE grund_typ    = RECORD
                      belegt : BOOLEAN;
                      wert   : INTEGER;
                    END;

     feld_typ     = ARRAY [1..300] OF grund_typ;
     feld_typ_ptr = ^feld_typ;
```

```
     Buftyp = ARRAY[1..4096] OF Byte;

VAR  ptr    : Pointer;
     BufPtr : ^Buftyp;

BEGIN

  ...
  New(BufPtr);           { Puffer im Heap anlegen            }
  ptr := BufPtr;         { Puffer über ptr adressieren       }

  { Ein ARRAY mit Datentyp feld_typ über den Puffer legen,
    d.h. den Puffer, wie feld_typ angibt, strukturieren, und
    einen Wert in das erste Feldelement übertragen           }
  feld_typ_ptr(ptr)^[1].belegt := TRUE;
  feld_typ_ptr(ptr)^[1].wert   := 200;
  ...

END.
```

Über den **Adreßoperator @** kann die Adresse einer Variablen, Prozedur, Funktion oder Methode (siehe Kapitel 9) ermittelt und einem Zeiger zugeordnet werden.

In den folgenden Kapiteln wird noch intensiv von Zeigertypen Gebrauch gemacht. Entwicklungsbibliotheken wie Turbo Vision oder Object Vision als Teil des Borland Pascal Compilers stellen umfangreiche Mengen an vordefinierten Zeigertypen bereit.

4.2.5 Prozedurtyp

Pascal sieht den Datentyp **Prozedurtyp** standardmäßig für Formalparameter einer Prozedur, nicht aber als Datentyp für Datenobjekte. Pascal-Dialekte wie Borland Pascal kennen jedoch auch den Datentyp Prozedurtyp. Die Syntax einer Variablen mit Datentyp Prozedurtyp wird an folgendem Beispiel deutlich:

```
TYPE math_proc_typ = PROCEDURE (x1, x2       : REAL;
                                VAR resultat : REAL);

VAR  f : math_proc_typ;
```

Der Prozedurtyp `math_proc_typ` definiert den Datentyp einer Prozedurvariablen, der als Wert ein Prozedurname zugewiesen werden kann. Aufgrund der Definition ist jede Prozedur mit drei Parametern vom Datentyp `REAL` erlaubt, wobei die Werte der ersten beiden Parameter durch die Prozedur nicht verändert werden können; durch das Pascal-Schlüsselwort `VAR` wird gekennzeichnet, daß die Prozedur den Wert des dritten Parameters verändern kann.

Sind die Variablen `r1, r2` und `r3` mit `REAL`-Typ definiert, so kann beispielsweise der so definierten Variablen mit Namen `f` der Name einer Prozedur mit drei Formalparametern mit Datentyp `REAL` zugewiesen und anschließend "`f` aufgerufen" werden:

```
PROCEDURE proc_1 (a, b         : REAL;
                  VAR ergebnis : REAL);
BEGIN { proc_1 }
 ...
END   { proc_1 };


PROCEDURE proc_2 (u, v  : REAL;
                  VAR w : REAL);
BEGIN { proc_2 }
 ...
END   { proc_2 };


VAR r1 : REAL;
    r2 : REAL;
    r3 : REAL;


BEGIN
 ...
  f := proc_1;
  f (r1, r2, r3)    { Aufruf von proc_1(r1, r2, r3) };
 ...
  f := proc_2;
  f (r1, r2, r3)    { Aufruf von proc_2 (r1, r2, r3) };
 ...
END.
```

Ein Datenobjekt mit Datentyp Prozedurtyp wird in Pascal intern wie ein Datenobjekt mit Zeigertyp behandelt. Bei einer Wertzuweisung an ein so deklariertes Datenobjekt wird die Adresse der benannten Prozedur übertragen.

4.2.6 Typkompatibilität

Bei Operationen, Wertzuweisungen und Prozedur- und Funktionsaufrufen müssen die Datentypen der beteiligten Datenobjekte **Kompatibilitätsregeln** genügen. Im folgenden Beispiel (vgl. [W/C]) werden sechs Felder deklariert, die jeweils aus 5 Feldelementen mit Datentyp `INTEGER` bestehen:

```
TYPE feld_typ_1 = ARRAY [1..5] OF INTEGER;
     feld_typ_2 = ARRAY [1..5] OF INTEGER;

VAR  a    : feld_typ_1;
     b    : feld_typ_2;
     c    : ARRAY [1..5] OF INTEGER;
     d, e : ARRAY [1..5] OF INTEGER;
     f    : feld_typ_1;
```

In der internen Darstellung unterscheiden sich diese Felder nicht; sie sind **strukturäquivalent**. Pascal setzt für die **Gleichheit von Datentypen** jedoch **Namensäquivalenz** voraus, d.h. zwei Variablen haben nur dann einen gleichen Datentyp, wenn sie mit demselben Datentypbezeichner definiert wurden. Im Beispiel haben `a` und `f` bzw. `d` und `e` denselben Datentyp, `a, b, c` und `d` jedoch nicht (da `ARRAY [1..5] OF INTEGER` kein Datentyp*bezeichner* ist, haben `c` und `d` unterschiedliche Datentypen).

Für die Datentypen der Aktualparameter und der korrespondierenden Formalparameter bei Prozedur- und Funktionsaufrufen wird in Pascal Gleichheit im Sinne der Namensäquivalenz vorausgesetzt. Erweiterte Kompatibilitätsregeln gelten bei der Bildung von Ausdrücken, relationalen Operationen und Wertzuweisungen. Die Einhaltung dieser Regeln, von denen die wichtigsten in Abbildung 4.2.6-1 zusammengefaßt sind, wird vom Compiler überprüft.

Zwei Datentypen sind kompatibel, wenn mindestens eine der Bedingungen erfüllt ist:
Beide Typen sind gleich (Namensäquivalenz).
Beide Typen sind Real-Datentypen.
Beide Typen sind Integer-Datentypen.
Ein Typ ist ein Teilbereichstyp des anderen.
Beide Typen sind Teilbereiche derselben Grundmenge.
Beide Typen sind `SET`-Typen mit kompatiblen Datentypen der Grundmenge.
Beide Typen sind `STRING`-Typen mit der gleichen Anzahl von Komponenten.
Der eine Typ ist ein `STRING`-Typ und der andere ist ein `PACKED ARRAY`-Typ mit nicht mehr Zeichen, als im `STRING`-Typ aufgrund seiner Definition maximal zugelassen sind.
Ein Typ ist der Datentyp `Pointer`, der andere ein beliebiger Zeigertyp.
Beide Typen sind Prozedurtypen mit identischen Ergebnistypen und gleicher Anzahl und Reihenfolge von Parametern gleichen Typs (Namensäquivalenz).

Der Wert eines Datenobjekts mit Datentyp T_2 kann einem Datenobjekt mit Datentyp T_1 zugewiesen werden, wenn eine der folgenden Bedingungen erfüllt ist (**Zuweisungskompatibilität)**:

- T_1 und T_2 sind vom gleichen Datentyp. Keiner von beiden ist ein FILE-Typ oder ein strukturierter Datentyp, der einen `FILE`-Typ als Komponente enthält.
- T_1 und T_2 sind kompatible Ordinaltypen, und der Wert von T_2 liegt im möglichen Wertebereich von T_1.
- T_1 und T_2 sind kompatible Real-Datentypen, und der Wert von T_2 liegt im möglichen Wertebereich von T_1.
- T_1 hat den Datentyp `REAL` und T_2 den Datentyp `INTEGER`.
- T_1 und T_2 sind `STRING`-Typen.
- T_1 ist ein `STRING`-Typ, und T_2 ist ein `CHAR`-Typ.
- T_1 ist ein `STRING`-Typ, und T_2 ist ist ein `PACKED ARRAY`-Typ mit nicht mehr Zeichen, als im `STRING`-Typ aufgrund seiner Definition maximal zugelassen sind.
- T_1 und T_2 sind kompatible `SET`-Typen, und alle Elemente des Werts von T_2 fallen in den möglichen Wertebereich von T_1.
- T_1 und T_2 sind kompatible Zeigertypen.
- T_1 und T_2 sind kompatible Prozedurtypen.
- T_1 und T_2 sind Prozedurtypen mit identischen Ergebnistypen sowie gleicher Anzahl und Reihenfolge von Parametern identischen Typs.
- (Zusätzliche Regeln gelten für die objektoritntierte Programmierung.)

Abbildung 4.2.6-1: Kompatibilitätsregeln von Datentypen

4.2.7 Vergleichsoperatoren

Vergleichsoperatoren vergleichen die Werte von Datenobjekten und haben je nach Zusammenhang, in dem sie stehen, unterschiedliche Bedeutung. Sind `v1` und `v2` Datenobjekte und OP ein Vergleichsoperator, so hat das Ergebnis des Vergleichs

```
(v1 OP v2)
```

immer den Ergebnistyp `BOOLEAN`. Abbildung 4.2.7-1 faßt die Vergleichsoperatoren zusammen.

Operator	Operation	Operandentyp	Bemerkung
`=`	gleich	kompatible einfache, Zeiger-, `SET`-, `STRING`- oder gepackte `STRING`-Typen	Bei Zeigertypen können nur die Operatoren = und <> benutzt werden.
`<>`	ungleich		
`<`	kleiner als		
`>`	größer als		
`<=`	kleiner/gleich		
`>=`	größer/gleich		
`<=`	Teilmenge von	kompatible `SET`-Typen	
`>=`	Obermenge von		
`IN`	Element von	linker Operand: Ordinaltyp `T`; rechter Operand: `SET OF T`	

Abbildung 4.2.7-1: Vergleichsoperatoren

5 Anweisungen in einer höheren Programmiersprache

Der prozedurale Ablauf eines Programms wird in seinem **Anweisungsteil** beschrieben, im folgenden erläutert an den wichtigsten Anweisungstypen in Pascal. Die einzelnen Anweisungen werden durch Semikolons getrennt. Die wichtigsten Anweisungstypen sind (eine ausführliche Beschreibung findet man z.B. in [H/D]):

- Die **leere Anweisung**, die aus der leeren Zeichenkette besteht.

- Die **Wertzuweisung**, bei der einer Variablen der Wert eines **Ausdrucks** zugewiesen wird. Der Ausdruck wird aus Operanden (Variablen, Aufruf vordefinierter Standardfunktionen oder selbstdefinierter Funktionen), Operatoren und Klammern nach den üblichen Regeln gebildet. Die syntaktische Form einer Wertzuweisung lautet:

  ```
  variable := ausdruck;
  ```

  Hierbei wird der rechts stehende Ausdruck ausgewertet und dieser Wert als neuer Wert der links stehenden Variablen zugewiesen. Der links stehende Variablenbezeichner kann in dem Ausdruck selbst vorkommen.

- Die **strukturierte Anweisung**. Dazu gehören die zusammengesetzte Anweisung, bei der eine Gruppe von Anweisungen durch die Schlüsselwörter `BEGIN` und `END` eingeschlossen wird, bedingte Anweisungen (`IF ... THEN ... ELSE...;` `CASE ... OF ... END`), Wiederholungsanweisungen (`WHILE ...DO...;` `FOR ... DO...;` `REPEAT ... UNTIL...;`).

- Der **Prozeduraufruf**, bei dem ein Unterprogramm aktiviert wird, dem je nach Definition aktuelle Parameter übergeben werden. Das Unterprogramm verändert eventuell je nach festgelegter **Parameterübergabemethode** (call-by-value, call-by-reference) die Werte der aktuellen Parameter. Nach Ausführung des Unterprogramms wird der Programmablauf genau hinter der Aufrufstelle fortgesetzt. Weitere Details folgen in Kapitel 8. Die Ein- und Ausgabe erfolgt ebenfalls über Aufrufe der Standardprozeduren `Read`, `Write`, `Readln`, `Writeln` usw.

Die hier beschriebene Sichtweise der Programmierung kann als **prozedurorientiert** bezeichnet werden: Basierend auf den definierten Daten (-objekten) liegt der Schwerpunkt auf der Definition des prozeduralen Ablaufs. Die Gesamtaufgabe wird gemäß den Grundsätzen der Strukturierten Programmierung in geschlossene Teilaufgaben (Blöcke, Prozeduren) zerlegt, die durch genau einen Start- und einen Endpunkt definiert sind.

5.1 Wichtige strukturierte Anweisungen

Die **bedingte Anweisung (`IF`-Anweisung)** hat die syntaktische Form

```
IF bedingung                    oder    IF bedingung
THEN BEGIN                              THEN BEGIN
       anweisungsblock_1                       anweisungsblock_1
     END                                     END;
ELSE BEGIN
       anweisungsblock_2
     END;
```

Die linke Alternative hat folgende Bedeutung:

Die hinter dem `IF` stehende logische Bedingung wird ausgewertet. Ergibt sich dabei der Wert `TRUE`, so wird der `THEN`-Zweig (`anweisungsblock_1`) ausgeführt; ergibt sich der Wert `FALSE`, wird der `ELSE`-Zweig (`anweisungsblock_2`) ausgeführt. In beiden Fällen wird bei der nach der `IF`-Anweisung fortgesetzt.

Bei der rechten Alternative wird der `IF`-Zweig nur ausgeführt, falls die Auswertung der Bedingung den Wert `TRUE` ergibt.

Besteht ein Anweisungsblock nur aus einer einzigen Anweisung, so kann die jeweilige umschließende `BEGIN-END`-Klammer entfallen.

Die **`CASE`-Anweisung** stellt eine Verallgemeinerung der `IF`-Anweisung dar. Ihre syntaktische Form lautet:

```
CASE v OF
v1 : BEGIN
       anweisungsblock_1
     END;
v2 : BEGIN
       anweisungsblock_2
     END;
...
vn : BEGIN
       anweisungsblock_n
     END;
ELSE BEGIN
       anweisungsblock_sonst
     END;
END;
```

Auch hier kann eine `BEGIN-END`-Klammer entfallen, falls der entsprechende Anweisungsblock nur aus einer einzigen Anweisung besteht; ebenfalls kann der gesamte `ELSE`-Zweig entfallen.

Die Anweisung hat folgende Bedeutung: Ist `v` eine Variable, die mindestens die Werte `v1`, ..., `vn` annehmen kann, so wird für den Fall, daß `v` gerade einen dieser Werte, etwa `vi`, hat, die Alternative `anweisungsblock_i` durchlaufen und anschließend hinter der `CASE`-Anweisung fortgesetzt. Hat `v` gerade keinen der Werte `v1`, ..., `vn`, so wird der `ELSE`-Zweig `anweisungsblock_sonst` durchlaufen (falls vorhanden) und anschließend hinter der `CASE`-Anweisung fortgesetzt.

Alternativen können zusammmengefaßt werden: Soll etwa ein Anweisungsblock `A` durchlaufen werden, wenn `v` den Wert `vi` oder `vj` hat, so lautet die entsprechende `CASE`-Anweisung

```
CASE v OF
v1 : BEGIN
       anweisungsblock_1
     END;
v2 : BEGIN
       anweisungsblock_2
     END;
...
vi, vj:
     BEGIN
       A
     END;
...
vn : BEGIN
       anweisungsblock_n
     END;
ELSE BEGIN
       anweisungsblock_sonst
     END;
END;
```

Es gibt mehrere syntaktische Formen der **Iteration**: Die **`WHILE`-Anweisung** hat die Form

```
WHILE wiederholungsbedingung DO
  BEGIN
    anweisungsblock
  END;
```

Hierbei wird die (logische) Wiederholungsbedingung ausgewertet. Ergibt sich dabei der Wert `TRUE`, wird der angegebene Anweisungsblock einmal durchlaufen. I.a. haben die einzelnen Anweisung in diesem Block Einfluß auf das Ergebnis einer erneuten Auswertung der Wiederholungsbedingung. Der Vorgang (Auswertung der Wiederholungsbedingung und erneutes Durchlaufen des Anweisungsblocks bei Ergebnis `TRUE`) wird solange wiederholt, bis sich bei einer Auswertung der Wiederholungsbedingung der Wert `FALSE` ergibt. Dann wird hinter der `WHILE`-Anweisung fortgesetzt. Das gleiche geschieht, wenn bereits bei der ersten Auswertung der Wiederholungsbedingung das Ergebnis `FALSE` entsteht. In diesem Fall wird der Anweisungsblock überhaupt nicht durchlaufen.

Besteht der Anweisungsblock nur aus einer einzigen Anweisung, so kann die `BEGIN-END`-Klammer entfallen.

Ein Spezialfall stellt die **`FOR`-Anweisung** dar: Hier hängt der Anweisungsblock von den Werten einer Variablen `i` ab und soll nacheinander für `i` = i_{min}, `i` = i_{min} + 1, ..., `i` = i_{max} durchlaufen werden. Dann ist auch folgende Form möglich:

```
FOR i := i_min TO i_max DO
  BEGIN
    anweisungsblock
  END;
```

Soll der Anweisungsblock in umgekehrter Reihenfolge, d.h. für `i` = i_{max}, `i` = i_{max} - 1, ..., `i` = i_{min} durchlaufen werden, so lautet die entsprechende Anweisung

```
FOR i := imax DOWNTO imin DO
  BEGIN
    anweisungsblock
  END;
```

Die folgenden Anweisungen haben in der Regel dieselbe Wirkung:

```
i := imin;
WHILE i <= imax DO
  BEGIN
    anweisungsblock;
    i := i + 1;
  END;
```

bzw.

```
FOR i := imin TO imax DO
  BEGIN
    anweisungsblock
  END;
```

Offensichtlich wird der Anweisungsblock bei $i_{min} > i_{max}$ nicht durchlaufen.

Eine weitere Iterationsanweisung ist die **`REPEAT`-Anweisung**:

```
REPEAT
  BEGIN
    anweisungsblock
  END
UNTIL abbruchbedingung;
```

Hierbei wird der Anweisungsblock einmal durchlaufen, bevor das erste Mal die `abbruchbedingung` ausgewertet wird. Ergibt sich hierbei der Wert `FALSE`, so werden der Anweisungsblock erneut ausgeführt und der Wert der Abbruchbedingung ermittelt. Der Vorgang wiederholt sich solange, bis die Abbruchbedingung den Wert `TRUE` ergibt.

Um eine `WHILE`-, `FOR`- oder `REPEAT`-Anweisung vorzeitig abzubrechen, d.h. bevor alle ursprünglich vorgesehenen Durchläufe ausgeführt wurden, gibt es in PASCAL folgende beiden Möglichkeiten, hier demonstriert für die `FOR`-Anweisung. Die Bedingung `abbruch` soll abhängig vom aktuellen Wert der Laufvariablen `i` den Wert `TRUE` liefern, falls die `FOR`-Anweisung abgebrochen werden soll.

```
FOR i := imin TO imax DO
  BEGIN
    anweisungsblock;
    IF abbruch
    THEN i := imax + 1;
  END;
```

bzw.

```
FOR i := imin TO imax DO
  BEGIN
    anweisungsblock;
    IF abbruch THEN Break;
  END;
```

Die "eingebaute" Funktion **`Break`** bricht die gesamte Iterationsanweisung ab.

5.2 Ein- und Ausgabe

Für die Ein- und Ausgabe stehen u.a. die Standardprozeduren
`Read`, **`Readln`** für die Eingabe und
`Write`, **`Writeln`** für die Ausgabe

zur Verfügung. Daneben gibt es weitere Prozeduren zur Manipulation von Dateien und Dateiinhalten. Dieses Kapitel gibt nur einen vereinfachten Einblick in die Ein- und Ausgabefunktionen, speziell für Borland Pascal; weitere Details findet man z.B. in [BP7].

Die Eingabe von Werten in Programmvariablen über die Tastatur erfolgt aus der dem Programm standardmäßig zugeordneten Textdatei `Input`, die mit der Tastatur verbunden ist, durch Aufruf von `Read` bzw. `Readln` (wie `Read`, nach der Eingabe erfolgt eine Positionierung auf die nächste Zeile der Eingabedatei). Das syntaktische Format lautet (hier nur für `Read`):

`Read (Input,` v_1`, ..., `v_n`);`
bzw. (da `Input` standardmäßig zugeordnet ist)
`Read (`v_1`, ..., `v_n`);`

Von der Tastatur werden hierdurch Werte in die durch die Bezeichner v_1, ..., v_n angegebenen Variablen gelesen, die unterschiedliche einfache Datentypen haben können. `Read` akzeptiert bei jedem Aufruf eine unterschiedliche Anzahl von Variablenbezeichnern. Die Umwandlung der Eingabewerte in das jeweilige Datenformat erfolgt implizit durch `Read` selbst.

Die Ausgabe von Werten aus Programmvariablen aus einem Programm auf den Bildschirm erfolgt durch Ausgabe der Werte in die standardmäßig zugeordneten Textdatei `Output`, die mit dem Bildschirm verbunden ist, durch Aufruf von `Write` bzw. `Writeln` (wie `Write`, nach der Ausgabe erfolgt eine Positionierung auf die nächste Zeile der Ausgabedatei). Das syntaktische Format lautet (hier nur für `Write`):

`Write (Output,` v_1`, ..., `v_n`);`
bzw. (da `Output` standardmäßig zugeordnet ist)
`Write (`v_1`, ..., `v_n`);`

Die Werte der durch v_1, ..., v_n bezeichneten Variablen eventuell unterschiedlichen einfachen Datentyps werden auf dem Bildschirm ausgegeben. Hierbei ist es nicht erforderlich, die Variablenwerte zunächst z.B. aus dem Datentyp `INTEGER` oder `REAL` in das `STRING`-Format umzuwandeln; die entsprechenden Umwandlungen werden von `Write` selbst vorgenommen.

Allgemein arbeiten `Read` (`Readln`) bzw. `Write` (`Writeln`) mit **sequentiellen Dateien** als Eingabequellen bzw. Ausgabeziel zusammen. Die Datei Input stellt dabei eine spezielle nur-lesbare Textdatei dar, die bereits mit der Tastatur verbunden ist. Entsprechend ist die Datei `Output` eine spezielle mit dem Bildschirm verbundene nur-beschreibbare Textdatei[22]. Der Datenverkehr zwischen einem Programm und einer physikalischen Datei namens `'DATEINAM.DAT'`, die Einträge vom (strukturierten oder unstrukturierten) Datentyp `datatyp` enthält, bedarf einiger programmtechnischer Vorbereitungsschritte (Borland Pascal):

[22] Eine Textdatei kann als eine Datei mit Datentyp `FILE OF CHAR` angesehen werden.

1. **Definition einer Dateivariablen**, die innerhalb des Programms das logische Bild der Datei repräsentiert, und einer Variablen, die Einzeleinträge aus der Datei aufnehmen kann:

```
...
{ ===> im Deklarationsteil <=== }

TYPE datatyp = ...;
...
VAR  dateivar : FILE OF datatyp;     { Dateivariable }
     entry    : datatyp;             { Einzeleintrag }
...
```

2. **Zuordnung des physikalischen Dateinamens** auf die (logische) Datei (-variable) mit Hilfe der Standardprozedur **`Assign`**:

```
...
{ ===> im Anweisungsteil <=== }

Assign (dateivar, 'DATEINAM.DAT');
...
```

3. **Öffnen der Datei** mit Hilfe der Standardprozedur `Reset` oder `Rewrite`:
   - **`Reset`** zum Nur-Lesen einer existierenden Datei und Positionieren eines **Lesefensters** auf den ersten Dateieintrag
   - **`Rewrite`** zum Einrichten einer neuen Datei bzw. Überschreiben einer existierenden Datei und Positionieren eines **Schreib/Lesefensters** auf den ersten Dateieintrag
   - **`Append`** zum Öffnen einer existierenden Datei und Anhängen weiterer Dateieinträge an das Dateiende (ein Schreib/Lesefenster wird auf das Dateiende positioniert):

```
...
{ ===> im Anweisungsteil <=== }

Reset (dateivar);
{ bzw. }
Rewrite (dateivar);
{ bzw. }
Append (dateivar);
...
```

4. **Sequentielles Lesen** in die Variable `entry` **bzw. Schreiben** des Werts der Variablen `entry` mit Hilfe der Standardprozeduren `Read` bzw. `Write`; dabei wird das Schreib/Lesefenster jeweils um eine Position in Richtung auf das Dateiende weitergesetzt.

```
...
{ ===> im Anweisungsteil <=== }

Read (dateivar, entry);
{ bzw. }
entry := ...;
Write (dateivar, entry);
...
```

5. **Schließen der Datei**, wenn keine weiteren Ein/Ausgabe-Operationen erforderlich sind. Hierbei werden die durch das Laufzeitsystem angelegten Datenpuffer für den Datenverkehr endgültig geleert und der Dateizugriff abgebaut:

```
...
{ ===> im Anweisungsteil <=== }

Close (dateivar);
...
```

Weitere Standardprozeduren sind (die folgende Auswahl ist hier nur namentlich aufgeführt):

Prozedur/Funktion	Beschreibung
`Eof`	prüft, ob das Ende der Datei erreicht ist und liefert einen entsprechenden Wahrheitswert (`BOOLEAN`)
`FilePos`	liefert die momentane Position des Schreib/Lesefensters innerhalb der Datei (`LongInt`)
`FileSize`	liefert die gegenwärtige Anzahl der Dateieinträge (`LongInt`)
`IOResult`	liefert den Fehlerstatus der zuletzt ausgeführten Ein/Ausgabeoperation (`INTEGER`)
`Seek`	setzt das Schreib/Lesefenster auf eine angegebene Position innerhalb der Datei
`SeekEof`	prüft, ob sich zwischen der gegenwärtigen Position des Schreib/Lesefensters und dem Dateiende noch lesbare Dateieinträge befinden und liefert einen entsprechenden Wahrheitswert (`BOOLEAN`)
`Truncate`	schneidet die datei an der gegenwärtigen Position des Schreib/Lesefensters ab

Für das Arbeiten mit sogenannten untypisierten Dateien stehen weitere Standardprozeduren zur Verfügung (siehe [BP7]).

Unter MS-DOS stellen die Geräte LPT1 bzw. PRN, LPT2, LPT3 (parallele Ausgänge, meist Drucker), CON (Konsole, d.h. bei Eingabe die Tastatur, bei Ausgabe der Bildschirm), COM1 bzw. AUX, COM2 (serielle Schnittstellen) und NUL (Einheit, die sämtliche Ausgabe ignoriert und keine Eingaben liefert) spezielle Dateien dar. Die Textausgabe
`'Dies ist ein Text'`
auf den an LPT1: angeschlossenen Drucker kann also durch

```
...
VAR drucker : Text;
...
BEGIN
...
  Assign  (drucker, 'LPT1');
  Rewrite (drucker);
  Writeln (drucker, 'Dies ist ein Text');
  Close   (drucker);
...
END.
```

erfolgen. Alternativ und vereinfachend kann man auch die in der Standard-Unit (siehe Kapitel 7) `Printer` vordefinierte Textdatei-Variable `Lst` verwenden, die LPT1: zugeordnet ist:

```
...
BEGIN
...
  Writeln (Lst, 'Dies ist ein Text');
...
END.
```

6 Das Blockkonzept einer höheren Programmiersprache

Den prinzipiellen Aufbau eines (PASCAL-) Programms zeigte bereits Abbildung 3.4-1. Pascal zählt zu den **blockorientierten Programmiersprachen**. ***Ein Block ist eine syntaktische Einheit, die alle darin enthaltenen Deklarationen nach außen abkapselt.***

Formal besteht ein **Block** eines Pascal-Programms aus einem Deklarationsteil und einem sich anschließenden Befehlsteil. Ein Block ist Teil eines Programms, einer Prozedurdeklaration oder einer Funktionsdeklaration (Abbildung 6-1).

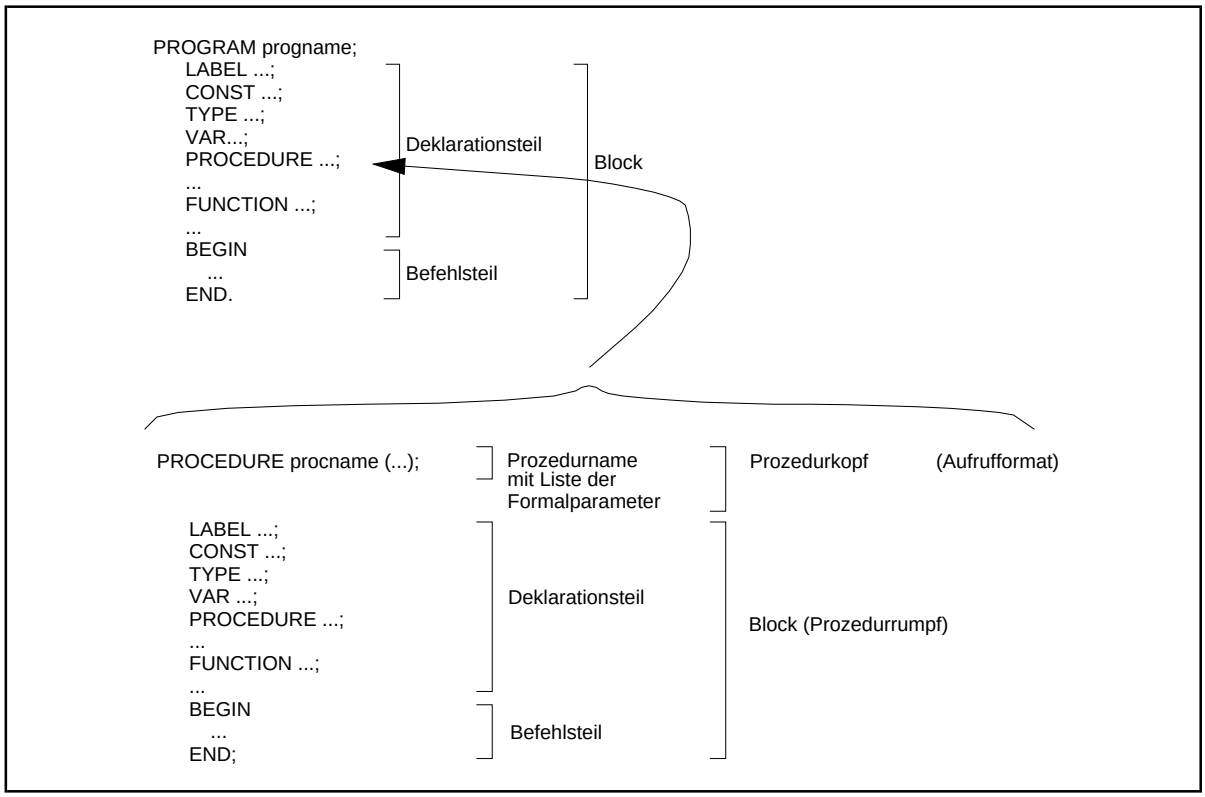


Abbildung 6-1: Block

Eine Prozedur- bzw. Funktionsdeklaration stellt die **Definition eines Unterprogramms** dar, d.h. eines separaten Programmteils, der von verschiedenen Stellen aus aktiviert (aufgerufen) werden kann. Bei **Aufruf eines Unterprogramms** wird in das bezeichnete Unterprogramm verzweigt, die dort definierten Anweisungen werden ausgeführt, und am Ende erfolgt automatisch eine Rückverzeigung (**Rücksprung zum Aufrufer**) genau hinter die Aufrufstelle im aufrufenden Programmteil. Eine Prozedur- bzw. Funktionsdeklaration kann eine Reihe von Variablen benennen (**Formalparameter**), die bei Aufruf der Prozedur bzw. Funktion durch den Aufrufer jeweils mit wechselnden Werten versehen werden können oder dazu dienen, wechselnde Ergebnisse an den Aufrufer zurückzugeben (**Aktualparameter**). In Pascal gibt es zwei verschiedene Methoden, um Aktualparameterwerte an ein Unterprogramm zu übergeben. Sie unterscheiden sich in der Wirkung auf Veränderungen der Aktualparameter im Unterprogramm (Kapitel 7.1).

Eine Prozedurdeklaration kann selbst wieder Prozedurdeklarationen enthalten, die aus Blöcken bestehen, in denen weitere Prozedurdeklarationen vorkommen können usw. Blöcke können also ineinandergeschachtelt sein, aber auch nebeneinander liegen. Bei ineinandergeschachtelten Blöcken kann man dann von **äußeren** und **inneren Blöcken** sprechen: aus Sicht eines Blocks sind anderere Blöcke, die in ihm definiert werden, und alle darin enthaltenen (eingebetteten) Blöcke innere Blöcke; die Blöcke, die einen definierten Block umfassen, sind aus der Sicht dieses Blocks äußere Blöcke. Ein aus Sicht eines Blocks äußerer Block heißt auch **übergeordneter Block**; entsprechend heißt ein innerer Block auch **untergeordneter Block**.

Im **Deklarationsteil** werden **Bezeichner** festgelegt für

- Konstanten, die bei der Übersetzung des Programms direkt in Code übersetzt werden (**Konstantendeklarationsteil** beginnend mit dem Schlüsselwort **`CONST`**)

- Datentypen für selbstdefinierte Datentypen (**Typdeklarationsteil** beginnend mit dem Schlüsselwort **`TYPE`**)

- Datenobjekte (**Variablendeklarationsteil** beginnend mit dem Schlüsselwort **`VAR`**; entsprechend wird ein Datenobjekt auch **Variable** genannt)

- Prozeduren und Funktionen (**Prozedurdeklarationsteil**).

Aus historischen Gründen (veraltete Programmiertechnik) ist die Definition von Sprungzielen möglich, an die der Programmablauf mit Hilfe einer Sprunganweisung (`GOTO`) verzweigen kann (**Labeldeklarationsteil** beginnend mit dem Schlüsselwort `LABEL`). Derartige Programmsprünge sind überflüssig und im Rahmen der Strukturierten Programmierung nicht erlaubt.

Außerdem ***müssen nicht alle Teile vorhanden sein***.

Bezeichner können beliebig lang sein; meist werden jedoch nur die ersten 255 Zeichen eines Bezeichners berücksichtigt. Ein Bezeichner beginnt mit einem Buchstaben (auch das Zeichen _ gilt als Buchstabe) und kann dann auch Ziffern enthalten. Groß- und Kleinschreibung wird nicht unterschieden. Schlüsselworte der Programmiersprache sind als selbstdefinierte Bezeichner nicht zugelassen. ***Innerhalb eines Blocks müssen Bezeichner eindeutig sein.*** Die Regeln über den Gültigkeitsbereich von Bezeichnern (siehe Kapitel 6.1) beschreiben das Verhalten bei Verwendung gleichlautender Bezeichner in ineinandergeschachtelten Blöcken.

Die einzelnen Teile des Deklarationsteils können in beliebiger Reihenfolge und Wiederholung auftreten (Borland Pascal). Das Ende eines Teils wird durch das erneute Auftreten eines Schlüsselworts (`CONST`, `TYPE`, `VAR`) erkannt. Bevor in einer Deklaration ein selbstdefinierter Bezeichner verwendet werden kann, muß er (bis auf gewisse Ausnahmen) deklariert worden sein.

Im **Befehlsteil** stehen die zu dem Block gehörenden Anweisungen. Der Befehlsteil ist in die Schlüsselwörter `BEGIN` und `END` eingeschlossen.

Eine **Prozedurdeklaration** unterscheidet sich von einem Programm syntaktisch nur durch die erste Zeile, den Prozedurkopf. Bei einer Funktion hat diese erste Zeile ein spezielles Format. Der Prozedurrumpf einer Prozedur oder Funktion ist selbst wieder ein Block und besteht aus Deklarations- und Befehlsteil. Eine PASCAL-Funktion ist prinzipiell eine Prozedur, die höchstens einen Rückgabewert, den Funktionswert, an den Aufrufer zurückgibt. Eine im Prozedurdeklarartionsteil definierte Prozedur stellt ein **internes Unterprogramm** dar. Eine Prozedurdeklaration setzt sich aus der Festlegung des **Aufrufformats (Prozedurkopf)** und der Anweisungen zusammen, die bei Aufruf der Prozedur ausgeführt werden sollen (**Prozedurrumpf**). Der Prozedurrumpf ist selbst wieder nach den Regeln aufgebaut, die für einen Block gelten.

6.1 Der Gültigkeitsbereich von Bezeichnern

Unterschiedliche Datenobjekte oder Typbezeichner in unterschiedlichen Blöcken können mit demselben Bezeichner versehen werden. Gleichlautende Bezeichner für verschiedene Datenobjekte im selben Block sind nicht zulässig.

Im Beispiel in Abbildung 6.1-1 werden in den Variablendeklarationsteilen Datenobjekte mit zum Teil gleichlautenden Bezeichnern (und jeweiligem Datentyp INTEGER) deklariert. Zur Unterscheidung der Datenobjekte erhalten sie in der Abbildung die Bezeichnungen d_1, ..., d_{11}. Die *Bezeichner* der einzelnen Datenobjekte lauten:

in `p_program`: `v` für das Datenobjekt d_1,
`w` für das Datenobjekt d_2,

in `a_proc`: `a` für das Datenobjekt d_3,
`b` für das Datenobjekt d_4,

in `b_proc`: `a` für das Datenobjekt d_5,
`i` für das Datenobjekt d_6,
`w` für das Datenobjekt d_7,

in `c_proc`: `a` für das Datenobjekt d_8,
`v` für das Datenobjekt d_9,

in `d_proc`: `v` für das Datenobjekt d_{10},
`w` für das Datenobjekt d_{11}.

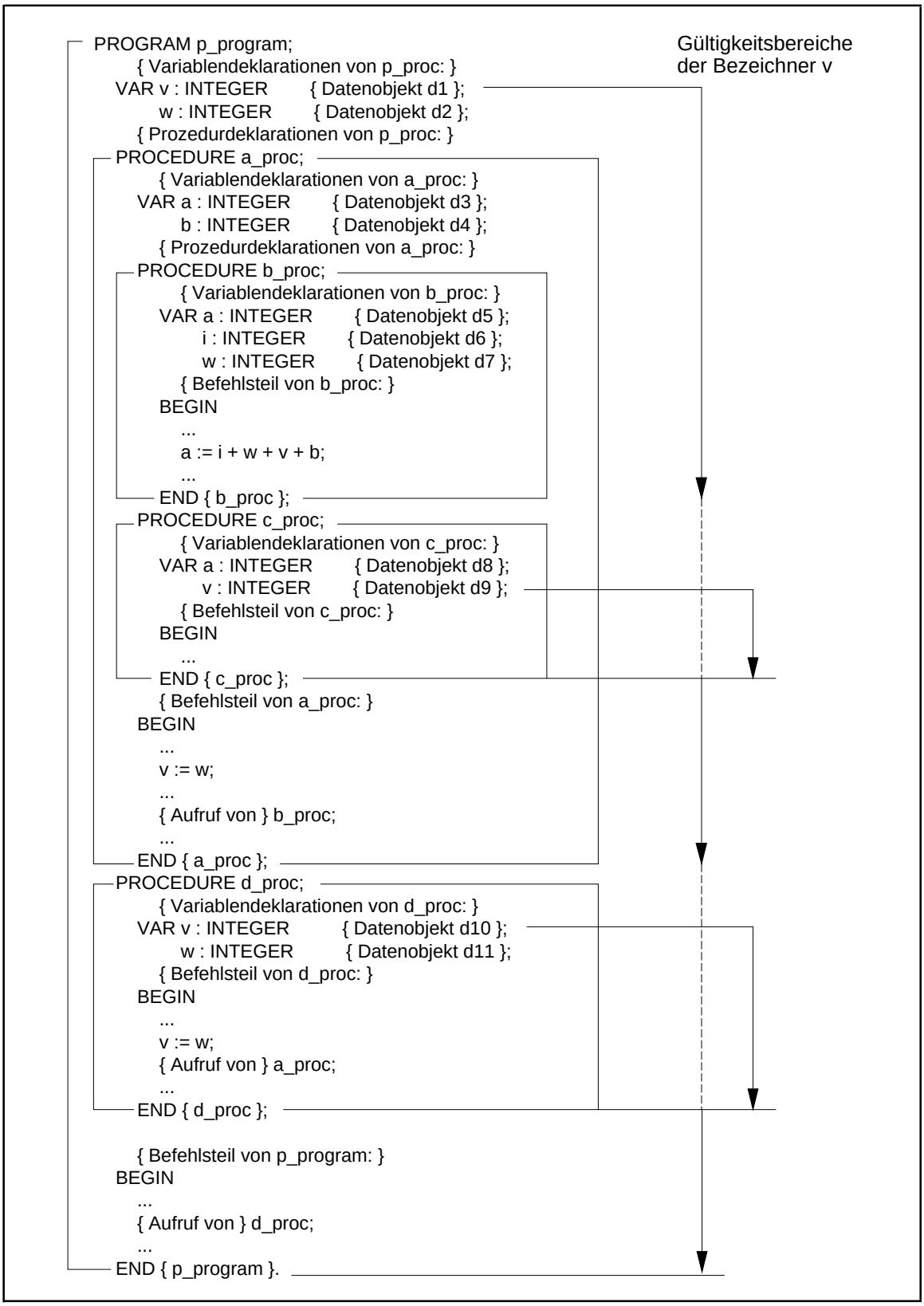


Abbildung 6.1-1: Gültigkeitsbereich von Bezeichnern (Beispiel)

Als generelle Regel für Bezeichner gilt:

Der **Gültigkeitsbereich eines Bezeichners** (für Konstanten, Typen, Variablen, Prozeduren, Funktionen) liegt zwischen dem Ort der Deklaration und dem Ende des die Deklaration enthaltenden Blocks, wobei alle Blöcke in den Gültigkeitsbereich mit eingeschlossen sind, die dieser Block umfaßt. Allerdings gibt es von dieser Regel eine wesentliche **Ausnahme**: Wird ein Bezeichner in einem Block B definiert, so schließt diese Definition den Gültigkeitsbereich eines gleichlautenden Bezeichners in einem äußeren Block A für diesen Block B und seine eingebetteten Blöcke aus.

Anders gesagt: Wenn in einem Block A ein Bezeichner definiert wird und in einem inneren Block B eine weitere Definition mit demselben Bezeichner auftritt, so wird durch diese Neudefinition der innere Block B und alle darin eingebetteten Blöcke aus dem Gültigkeitsbereich des Bezeichners des äußeren Blocks A ausgeschlossen.

Bezeichner müssen deklariert sein, bevor sie benutzt werden können. Sie können in einem Block nur jeweils einmal deklariert werden, es sei denn, ihre Neudeklaration erfolgt innerhalb eines untergeordneten Blocks[23]. Bezeichner sind von außerhalb des definierenden Blocks nicht "sichtbar". ***Ein Block kapselt die Deklaration zu den äußeren Blöcken ab, so daß man einen in einem Block B definierten Bezeichner bezüglich eines diesen Block B umfassenden (äußeren) Blocks A als lokal im Block B ansehen kann.***

Ein lokaler Bezeichner eines Datenobjekts benennt eine **lokale Variable** (**lokales Datenobjekt**). Bezüglich eines im Block B eingebetteten (inneren) Blocks C ist die Deklaration eines Bezeichners **global**. Ein globaler Bezeichner eines Datenobjekts benennt eine **globale Variable**. Die obige Ausnahmeregel besagt: ***Lokale Deklarationen überlagern globale Deklarationen.***

Für den Bezeichner einer Prozedur gilt in Übereinstimmung mit den obigen Regeln, daß er zum Prozedurrumpf selbst global und zu dem Block, der die Prozedurdefinition enthält, lokal ist. Der Gültigkeitsbereich eines Prozedurbezeichners umfaßt also den Block, der die Prozedurdefinition enthält (als lokale Definition), den Prozedurrumpf und alle eingebetteten Blöcke (als globale Definition), wobei obige Ausnahmeregel zu beachten ist. Als Konsequenz ergibt sich beispielsweise, daß sich eine Prozedur in ihrem Prozedurrumpf selbst aufrufen kann, da der Prozedurname im Prozedurrumpf global bekannt ist. Auf diese Möglichkeit des rekursiven Prozeduraufrufs wird in Kapitel 7 genauer eingegangen.

Der Begriffs des Gültigkeitsbereichs eines Bezeichners wird an den Bezeichnern der Datenobjekte (Variablen) in obigem Beispiel verdeutlicht:

Der Gültigkeitsbereich des Bezeichners `v` umfaßt die Blöcke `p_program` als lokale Deklaration und die Blöcke `a_proc` und `b_proc` als globale Deklaration, da `v` in `p_program` definiert wird und `a_proc` und `b_proc` in `p_program` eingebettet sind

[23] Eine Neudeklaration ist auch innerhalb der Komponentenliste eines RECORD-Typs möglich; der RECORD-Typ kapselt die Neudeklaration nauch außen ab.

und keine Neudefinition von v enthalten. Mit dem Bezeichner v wird also in p_program, a_proc und b_proc dasselbe Datenobjekt d_1 angesprochen. Der Gültigkeitsbereich des Bezeichners v für das Datenobjekt d_1 aus der Definition in p_program umfaßt aber nicht die Blöcke c_proc und d_proc, da hier jeweils Neudefinitionen des Bezeichners v vorkommen und damit jeweils unterschiedliche Datenobjekte (d_9 bzw. d_{10}) bezeichnen.

Wichtig ist, die *Reihenfolge* der Deklarationen in einem Block zu beachten: Im Beispiel der Abbildung 6.1-1 liegen die Blöcke a_proc und d_proc nebeneinander auf hierarchisch gleicher Stufe. Da die Deklaration des Bezeichners a_proc *vor* der Deklaration des Bezeichners d_proc erfolgt, liegt der Prozedurrumpf der Prozedur d_proc als zum Block p_program untergeordneter Block im Gültigkeitsbereich des Bezeichners a_proc (innerhalb des Prozedurrumpfs von d_proc ist der Bezeichner a_proc global). Das bedeutet, daß im Prozedurrumpf der Prozedur d_proc der Bezeichner a_proc "sichtbar" ist und die Prozedur a_proc aufgerufen werden kann. Jedoch kann innerhalb der Prozedur a_proc kein Aufruf der Prozedur d_proc erfolgen, da der Prozedurrumpf von a_proc nicht im Gültigkeitsbereich des Bezeichners d_proc liegt. Eine analoge Situation herrscht aufgrund der Reihenfolge, in der die Bezeichner definiert werden, im Verhältnis von b_proc und c_proc zueinander: In c_proc könnte ein Aufruf von b_proc erfolgen, aber nicht umgekehrt.

Die Prozedur b_proc könnte die Prozedur a_proc aufrufen, da der Bezeichner a_proc global bezüglich b_proc ist (der Block b_proc ist im Block a_proc eingebettet). Ein Aufruf der Prozedur d_proc im Prozedurrumpf von b_proc ist aber nicht möglich, da der Gültigkeitsbereich des Bezeichners d_proc den Block d_proc (und alle eingebetteten Blöcke) umfaßt und der Block b_proc nicht in d_proc enthalten ist.

Abbildung 6.1-2 faßt die Gültigkeitsbereiche der Bezeichner für Datenobjekte und die Zugriffsmöglichkeiten aus Abbildung 6.1-1 aus Sicht des jeweiligen Blocks (in Klammern stehen die mit dem jeweiligen Bezeichner angesprochenen Datenobjekte) zusammen. Zusätzlich sind die Gültigkeitsbereiche der Bezeichner der Prozeduren aufgeführt.

Man sieht also, daß der Bezeichner a in der Prozedur b_proc das lokale Datenobjekt d_5 (und nicht etwa das Datenobjekt d_3 wie in a_proc) bezeichnet. Entsprechend bezeichnet der Bezeichner w in b_proc das lokale Datenobjekt d_7. Auf das Datenobjekt d_2 ist aus b_proc heraus nicht zugreifbar, da der Bezeichner w lokal für d_7 verwendet wird.

Block	sichtbare lokale Bezeichner	sichtbare globale Bezeichner
p_program	v (d_1), w (d_2); a_proc, d_proc	p_program
a_proc	a (d_3), b (d_4); b_proc, c_proc	v (d_1), w (d_2); a_proc, p_program
b_proc	a (d_5), i (d_6), w (d_7)	b (d_4), v (d_1); b_proc, a_proc, p_program
c_proc	a (d_8), v (d_9)	b (d_4), w (d_2); c_proc, b_proc, a_proc, p_program
d_proc	v (d_{10}), w (d_{11})	d_proc, a_proc, p_program

Block	Bezeichner	Gültigkeitsbereich
p_program	v (d_1)	p_program, a_proc, b_proc
	w (d_2)	p_program, a_proc, c_proc
	a_proc	p_program, a_proc, b_proc, c_proc, d_proc
	d_proc	p_program, d_proc
a_proc	a (d_3)	a_proc
	b (d_4)	a_proc, b_proc, c_proc
	b_proc	a_proc, b_proc, c_proc
	c_proc	a_proc, c_proc
b_proc	a (d_5)	b_proc
	i (d_6)	b_proc
	w (d_7)	b_proc
c_proc	a (d_8)	c_proc
	v (d_9)	c_proc
d_proc	v (d_{10})	d_proc
	w d_{11})	d_proc

Abbildung 6.1-2: Gültigkeitsbereich von Bezeichnern (Beispiel, Fortsetzung von Abbildung 6.1-1)

6.2 Bemerkungen zur Lebensdauer von Datenobjekten

Ein wichtiges Charakteristikum eines Datenobjekts in der Sprache Pascal (und anderen blockorientierten Sprachen) ist seine **Lebensdauer**. Während der Gültigkeitsbereich eines Datenobjektbezeichners eine statische Eigenschaft eines syntaktischen Konstrukts der Programmiersprache, nämlich eines Bezeichners, ist, beschreibt der Begriff der Lebensdauer eines Datenobjekts in einem Pascal-Programm eine typische dynamische Eigenschaft.

Die **Lebensdauer eines Datenobjekts** beginnt, wenn diesem Datenobjekt Speicherplatz zugewiesen wird und endet, wenn dieser zugewiesene Speicherplatz zur anderweitigen Verwendung wieder freigegeben wird.

In einem Pascal-Programm werden den Datenobjekten des Hauptprogramms bei Programmstart Speicherplatz zugeordnet. Den Datenobjekten der Prozeduren und Funktionen wird erst im Augenblick des Prozedur- bzw. Funktionsaufrufs Speicherplatz zugeordnet. In einem Pascal-Programm gilt daher für die Lebensdauer eines Datenobjekts:

Die **Lebensdauer eines Datenobjekts beginnt** zu dem Zeitpunkt, zu dem die erste Anweisungen (des Objektcodes) des Blocks ausgeführt wird, in dem das Datenobjekt deklariert ist. Sie **endet** mit dem Abschluß der letzten Anweisung (des Objektcodes) des Blocks, in dem das Datenobjekt deklariert ist. Das bedeutet beispielsweise, daß Datenobjekte, die in einer Prozedur definiert sind, die während eines Programmlaufs jedoch nicht aufgerufen wird, auch nicht "zu leben" beginnen, d.h. daß ihnen kein Speicherplatz zugewiesen wird.

Die Lebensdauer der einzelnen Datenobjekte in Abbildung 6.1-1 während des Programmablaufs zeigt Abbildung 6.2-1. Der linke Teil gibt von oben nach unten der Linie folgend den zeitlichenm Ablauf wieder; ein Knick nach rechts symbolisiert einen Unterprogrammsprung, ein Knick nach links eine Rückkehr in das rufende Programm. Im rechten Teil ist die Lebensdauer eines jeden Datenobjekts bei diesem Programmablauf durch eine senkrechte Linie dargestellt, deren Abschlußpunkte jeweils den Beginn bzw. das Ende der Lebensdauer markieren.

Einige Bemerkungen erläutern die Zusammenhänge:

1. Der Beginn der Lebensdauer des Datenobjekts d_{10} mit Bezeichner `v` beendet nicht die Lebensdauer des gleichnamigen Datenobjekts d_1, obwohl ein Zugriff mittels des Bezeichners `v` auf d_1 in diesem Augenblick nicht möglich ist. Die gleiche Aussage gilt für die Datenobjekte mit Bezeichner `w` (d_2 und d_{11}). Die Datenobjekte d_1 und d_2 sind in diesem Augenblick **nicht sichtbar**.

2. Die Lebensdauer des Datenobjekts d_3 ist nicht beendet, obwohl auf d_3 mit dem Bezeichner `a` nicht zugegriffen werden kann.

3. Mit dem Beginn der Lebensdauer kann über den Bezeichner `w` zur Zeit weder auf das Datenobjekt d_2 noch auf das Datenobjekt d_{11} zugegriffen werden; ihre Lebensdauer ist aber noch nicht beendet.

4. Die Datenobjekte d_8 und d_9 beginnen bei diesem Programmablauf nicht zu leben, da keine Anweisungen im sie definierenden Block `c_proc` ausgeführt werden.

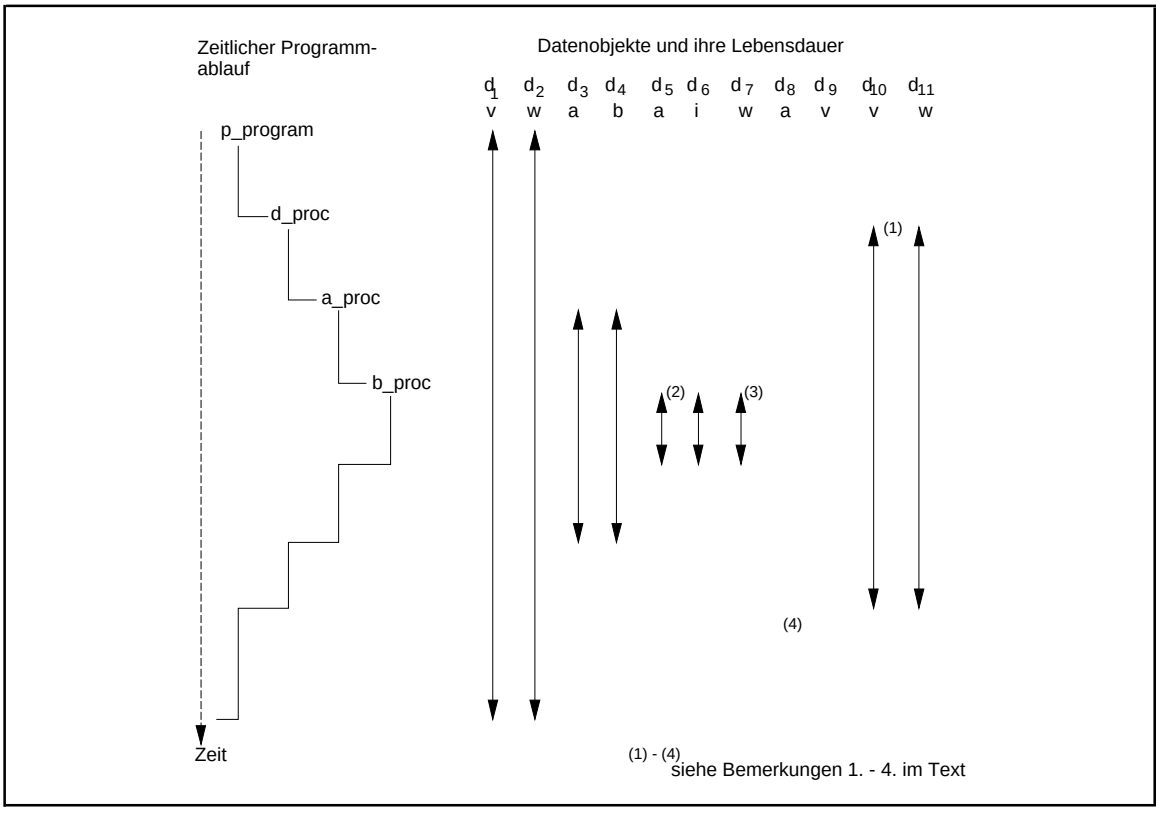


Abbildung 6.2-1: Lebensdauer von Datenobjekten

6.3 Bemerkungen zu Speicherklassen

Gemäß dem Lebensdauerkonzept wird einem Datenobjekt, das lokal in einer Prozedur deklariert ist, Speicherplatz zugeordnet, wenn die Prozedur aufgerufen wird. Sobald die Prozedur wieder verlassen wird und die Lebensdauer des Datenobjekts endet, ist die Zuordnung nicht mehr vorhanden. Bei einem erneuten Prozeduraufruf erfolgt eine neue Speicherplatzzuordnung, die vollkommen unabhängig von der vorigen Speicherplatzzuordnung ist. Insbesondere sind die Werte, die in vorherigen Prozeduraufrufen an das lokale Datenobjekt zugewiesen wurden, beim erneuten Eintritt in die Prozedur nicht mehr vorhanden. Einem derartigen Datenobjekt wird also **dynamisch** Speicherplatz zugewiesen. Eine Variablen, die im Hauptprogramm deklariert wird, beginnt zu leben, sobald das Programm startet, und seine Lebensdauer endet erst mit Programmende. Seine Speicherplatzzuweisung ist **statisch**. Die Art der Speicherplatzzuordnung ist implizit durch die Verwendung der jeweiligen Variablen bestimmt, eine weitergehende Steuerung durch den Anwender ist nicht vorgesehen. Sie ist auch nicht erforderlich, wenn man davon ausgeht, daß die Programmiersprache implementierungstechnische Details vor dem Anwender verbergen soll.

Einen anderen Ansatz verfolgen hier maschinennähere Programmiersprachen wie C oder C++ mit der Definition von **Speicherklassen**, mit denen der Anwender explizit steuern kann

(wenn er es möchte), wo im übersetzten Programm die reservierten Datenobjekte liegen bzw. wann ihre Lebensdauer endet. In C und C++ können folgende Speicherklassen in der Deklaration eines Datenobjekts vereinbart werden:

- **auto** (default-Wert): Das Datenobjekt wird wie in Pascal behandelt

- **static**: Die Lebensdauer des in einer Funktion lokal deklarierten Datenobjekts beginnt, wenn die Funktion zum erstenmal aufgerufen wird. Dann wird sie auch mit einem Anfangswert versehen (explizit angegeben oder default-Wert 0). Die Lebensdauer endet mit Programmende; insbesondere behält das Datenobjekt seinen Wert, wenn die Funktion verlassen wird, und steht mit diesem Wert bei einem erneuten Funktionsaufruf zur Verfügung. Der Gültigkeitsbereich des Bezeichners des Datenobjekts erstreckt sich über den Block, in dem der Bezeichner deklariert ist

- **extern**: Das Datenobjekt ist in einem (getrennt übersetzten) externen Modul deklariert.

- **register**: das lokale Datenobjekt wird vom Compiler auf ein Register der CPU abgebildet, falls es verfügbar ist, um einen schnelleren Zugriff auf das Datenobjekt zu gewährleisten, als es bei Abbildung auf ein internes Datenobjekt im Arbeitsspeicher möglich wäre.

7 Das Prozedurkonzept

In den meisten Programmen gibt es Anweisungsteile, die mehrfach auftreten oder sich zumindest ähneln. Beispielsweise kann es in einem Algorithmus notwendig sein, ein Feld mehrmals nach einem Element mit einer bestimmten Eigenschaft zu durchsuchen; die Berechnung desjenigen Elements, das größenmäßig "in der Mitte" aller gerade im Feld abgelegten Elemente steht, kann ein derartiges Durchsuchen erfordern. Typische andere Beispiele sind die Eingabe und Ausgabe von Daten an mehreren Stellen eines Programms. Derartige sich wiederholende "ähnliche" Anweisungsfolgen brauchen nur einmal in Form eines **Unterprogramms** kodiert zu werden.

Ein Unterprogramm wird in einer Programmiersprache als **Prozedur** deklariert. Pascal kennt als Prozedur das syntaktische Konstrukt der `PROCEDURE` und als Spezialform die mit Hilfe des Schlüsselworts `FUNCTION` vereinbarte **Funktion**. C, C++ und Java kennen nur eine der Funktion entsprechende Form. In COBOL können "interne" Prozeduren als Paragraphen oder Sections (aufgrufen mit `PERFORM`) oder "externe" Prozeduren (aufgerufen mit `CALL`) vereinbart werden.

Im folgenden wird neben einer allgemeinen Behandlung des Prozedurkonzepts einer höheren Programmiersprache im speziellen auf das Prozedurkonzept von Pascal eingegangen.

Eine **Prozedurdeklaration** besteht aus **Prozedurkopf** und **Prozedurrumpf**. In Pascal sieht die Prozedurdeklaration bezüglich des syntaktischen Aufbaus wie ein Programm aus und kann selbst wieder eingebettete Prozeduren enthalten (diese eingebettete Prozeduren sind dann nur innerhalb der Prozedur bekannt).

Der **Prozedurkopf ("Musteranweisung", Aufrufformat)** legt fest:

- mit welchem **Bezeichner eine Prozedur** aufgerufen wird

- die **Liste der Formalparameter** der Prozedur: hierbei wird für jeden Formalparameter
  -- sein Bezeichner innerhalb der Prozedur,
  -- sein Datentyp,
  -- die Reihenfolge, in der er in der Liste der Formalparameter auftritt,
  -- die Art und Weise, wie über diesen Formalparameter Werte an die Prozedur bzw. aus der Prozedur zum Aufrufer zurück Werte übergeben werden können: die **Parameterübergabemethode**

  festgelegt. ***Jeder Formalparameter stellt eine innerhalb der Prozedur lokale Variable dar***, die bei Aufruf der Prozedur vom Aufrufer mit Anfangswerten initialisiert wird. ***Nach außen hin***, d.h. zum Block, der die Prozedurdeklaration enthält, ***ist lediglich der Prozedurbezeichner bekannt*** und nicht etwa der Bezeichner eines Formalparameters. Die Liste der Formalparameter kann auch fehlen; die Prozedur hat dann keine Formalparameter.

Das syntaktische Format einer Prozedurdeklaration in Pascal lautet für eine Prozedur mit Bezeichner `xyz`:

```
PROCEDURE xyz (liste_der_Formalparameter);
```

Der **Prozedurrumpf** enthält eventuell Deklarationen weiterer lokaler Bezeichner und Objekte (Konstanten, Variablen, Prozeduren, Funktionen usw.) und den Anweisungsteil, d.h. die Anweisungen, die bei Aufruf der Prozedur durchlaufen werden. Der Prozedurrumpf stellt einen Block dar; insbesondere gelten hier die Regeln über die Gültigkeit von Bezeichnern.

Zusammenfassend sind die (lokalen) Datenobjekte, auf die innerhalb des Prozedurrumpfs zugegriffen werden kann, ***die Formalparameter und die im Prozedurrumpf deklarierten Datenobjekte.*** Zusätzlich kann eine Prozedur eventuell auf für sie globale Datenobjekte zugreifen, die in einem sie umfassenden Block deklariert sind (wenn diese nicht mit einem Bezeichner benannt werden, der innerhalb der Prozedur für ein lokales Datenobjekt verwendet wird, siehe Kapitel 6.1). ***Die Verwendung*** derartiger ***globaler Objekte sollte in einer Prozedur*** jedoch ***vermieden werden.***

Der **Aufruf einer Prozedur** erfolgt durch Nennung des zugehörigen Bezeichners und durch die Angabe der **Liste der Aktualparameter**, falls die Prozedur in ihrer Prozedurdeklaration Formalparameter vorsieht. ***Für jeden Formalparameter muß es einen korrespondierenden Aktualparameter geben, der im Datentyp mit dem Formalparameter übereinstimmt (kompatibel ist).*** Ein Aktualparameter ist meist ein Variablenbezeichner im Gültigkeitsbereich des Aufrufers; je nach Parameterübergabemethode ist eventuell auch eine Konstante oder ein arithmetischer Ausdruck zugelassen. Bei Prozeduraufruf wird der Aktualparameter "an das Unterprogramm übergeben"; ist der Aktualparameter ein arithmetischer Ausdruck, wird dieser vorher berechnet. Dann verzweigt der Programmfluß in die Anweisungen des bezeichneten Prozedurrumpfs. Nachdem dieser bis zum Ende durchlaufen ist, erfolgt ein **Rücksprung** genau hinter die Aufrufstelle der Prozedur (ein anderes Verlassen einer Prozedur, etwa durch Einsatz spezieller Rücksprungbefehle wie `Exit` in Borland Pascal, sollte man unterlassen). Nach dem Rücksprung sind die lokalen Datenobjekte der Prozedur nicht mehr verfügbar: ihre Lebensdauer, die bei Eintritt des Kontrollflusses in die Prozedur beginnt, ist beendet.

Eine spezielle Form einer Prozedur in Pascal ist eine **Funktion**. Eine Funktion besteht wie eine Prozedur aus einem Prozedurkopf und einem Prozedurrumpf. Die syntaktische Konstruktion lautet:

```
FUNCTION funktionsbezeichner (liste_der_formalparameter) : typbezeichner;
```

z.B.

```
FUNCTION fakultaet (n : INTEGER) : INTEGER;
```

Der Prozedurkopf enthält den Funktionsbezeichner, die Liste der Formalparameter und einen Typbezeichner, der den Datentyp des Funktionsergebnisses festlegt. Für die Liste der Formalparameter gilt das gleiche wie für die Liste der Formalparameter bei einer "normalen" Prozedur. Innerhalb des Prozedurrumpfs gibt es eine Wertzuweisung, die auf der *linken Seite* den Funktionsbezeichner hat und diesem einen Wert vom Typ des Typbezeichners zuordnet. Dadurch wird der Funktionswert festgelegt, der als Rückgabeparameter der Funktion fungiert:

```
...
 funktionsbezeichner := ...;
...
```

Im aufrufenden Programm wird der Funktionsbezeichner auf der *rechten Seite* einer Anweisung in einem Ausdruck (wie eine Variable) verwendet. In diesem Fall verzweigt der Programmfluß wie bei einem Unterprogrammaufruf in den Prozedurrumpf der Funktion, durchläuft die entsprechenden Anweisungen und kehrt direkt hinter die Stelle zurück, an der beim Aufrufer der Funktionsbezeichner steht. Als Besonderheit können Funktionen in Borland Pascal zusätzlich wie normale Prozeduren aufgerufen werden, d.h. einfach durch Angabe des Funktionsbezeichners und nicht etwa auf der rechten Seite einer Wertzuweisung innerhalb eines Ausdrucks. In diesem Fall wird das Funktionsergebnis ignoriert.

Die Einführung des syntaktischen Konzepts der `FUNCTION` bedeutet in vielen Anwendungsfällen ein Vereinfachung in der Programmierung. Natürlich ist jede `FUNCTION` durch eine "normale" `PROCEDURE` ersetzbar: Das Konstrukt

```
FUNCTION funktionsbezeichner (liste_der_formalparameter) : typbezeichner;
```

im Deklarationsteil eines Blocks und die anschließende Verwendung dieser `FUNCTION`, etwa in der Form

```
w := funktionsbezeichner (liste_der_aktualparameter);
```

wobei `w` eine Variable mit zu `typbezeichner` kompatiblen Datentyp ist, kann durch eine `PROCEDURE` mit einem weiteren Variablenparameter ersetzt werden:

```
PROCEDURE procbezeichner (VAR return_wert : typbezeichner;
                          liste_der_formalparameter);
```

Die Verwendung der Procedur wird dann durch

```
procbezeichner (w, liste_der_aktualparameter);
```

ersetzt.

Aufgrund dieser Ähnlichkeit von Prozeduren und Funktionen werden daher im folgenden hauptsächlich Prozeduren behandelt.

7.1 Parameterübergabe

Je nach Programmiersprache werden unterschiedliche Methoden verwendet, um Werte zwischen einer Prozedur und ihrem Aufrufer auszutauschen. Zunächst soll daher ein allgemeiner Überblick über die **Methoden der Parameterübergabe an Unterprogramme** gegeben werden, wie sie in höheren Programmiersprachen üblich sind. Dabei werden in diesem Kapitel weniger Implementierungsaspekte als vielmehr die **Semantik der Methoden zur Parameterübergabe aus Anwendersicht** behandelt.

Um die Methoden graphisch zu illustrieren, soll angenommen werden, daß ein Datenobjekt durch seinen Bezeichner, seine Adresse und seinen Wert charakterisiert wird (die anderen Eigenschaften, die in Abbildung 4-1 beschrieben wurden, spielen in diesem Zusammenhang

zunächst keine Rolle). Eine geeignete Darstellung zeigt Abbildung 7.1-1. Ein Datenobjekt wird durch seinen Bezeichner (im Rechteck), seine virtuelle Adresse (in der Raute) und seinen Wert (im Kreis) beschrieben. Ein Formalparameter einer Prozedur und sein korrespondierender Aktualparameter sind dabei unterschiedliche Datenobjekte, deren Datentypen gleich, zumindest aber kompatibel sind. Die Parameterübergabemethode spezifiziert die "Verbindung" zwischen beiden Datenobjekten; der Compiler der Programmiersprache erzeugt entsprechenden Code, um die so definierte Methode zu realisieren.

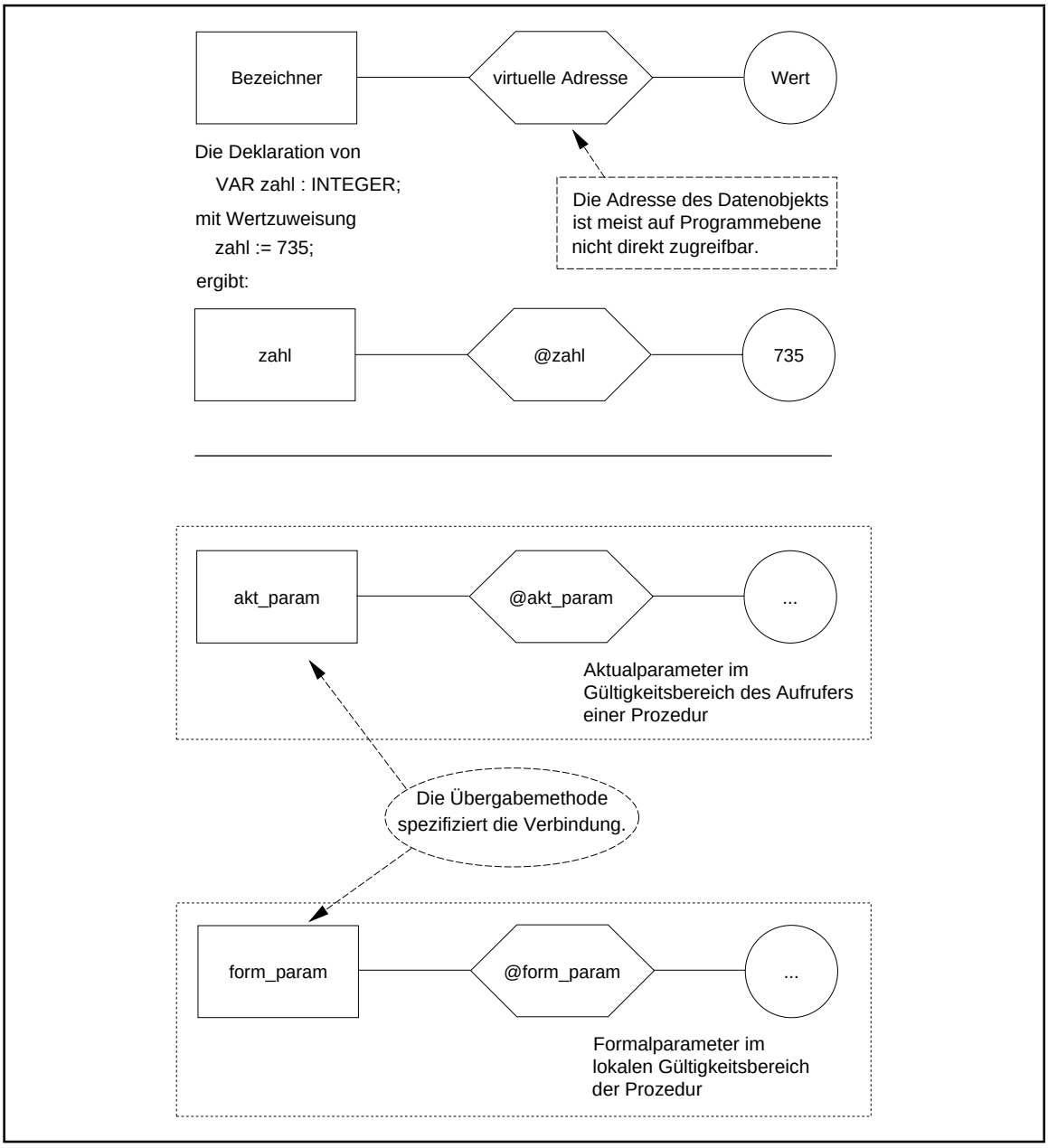


Abbildung 7.1-1: Graphische Darstellung eines Datenobjekts

Die Methoden (vgl. [W/C], [G/J]) lassen sich danach einteilen, ob sie dazu geeignet sind, Parameterwerte an eine Prozedur zu übergeben (Eingabe von Informationen in ein Unterprogramm), Parameterwerte aus einer Prozedur zurückzuerhalten (Rückgabe von Informationen aus einem Unterprogramm an den Aufrufer) oder ob sie "für beide Richtungen" geeignet sind (Änderung von Informationen durch ein Unterprogramm)[24].

Alle Methoden tragen Namen der Form "call-by-..."-Methode.

Eine für alle drei Teilaufgaben geeignete Methode ist die **call-by-reference-Methode** (Abbildung 7.1-2). Hierbei wird beim Eintritt in die Prozedur die ***Adresse des korrespondierenden Aktualparameters*** als Wert an den Formalparameter übergeben. Im gerufenen Unterprogramm wird dann eine Bezugnahme auf den entsprechenden Formalparameter als Referenz auf die Position behandelt, deren Adresse übergeben wurde. Es wird also bei jedem Zugriff auf den Formalparameter tatsächlich eine indirekte Referenz über die bekanntgegebene Adresse auf den nicht-lokalen Aktualparameter durchgeführt. ***Eine als aktueller Parameter übergebene Variable wird somit direkt bei jeder Änderung vom Unterprogramm modifiziert.*** Diese semantische Regel bedeutet, daß durch die Wertzuweisung

```
form_param := wert_neu;
```

innerhalb einer Prozedur mit dem Formalparameter `form_param` nicht etwa das durch `form_param` bezeichnete Datenobjekt den neuen Wert `wert_neu` erhält, sondern daß der Wert von `form_param` als Adresse eines Datenobjekts (nämlich des korrespondierenden Aktualparameters) interpretiert wird, dem nun der Wert `wert_neu` zugewiesen wird. Der Compiler hat bei der Übersetzung entsprechenden Code erzeugt. Analog wird durch die Auswertung bzw. des Lesens des Formalparameters `form_param`

```
IF form_param = ... THEN ...
```

innerhalb des Unterprogramms der Wert des korrespondierenden Aktualparameters ausgewertet.

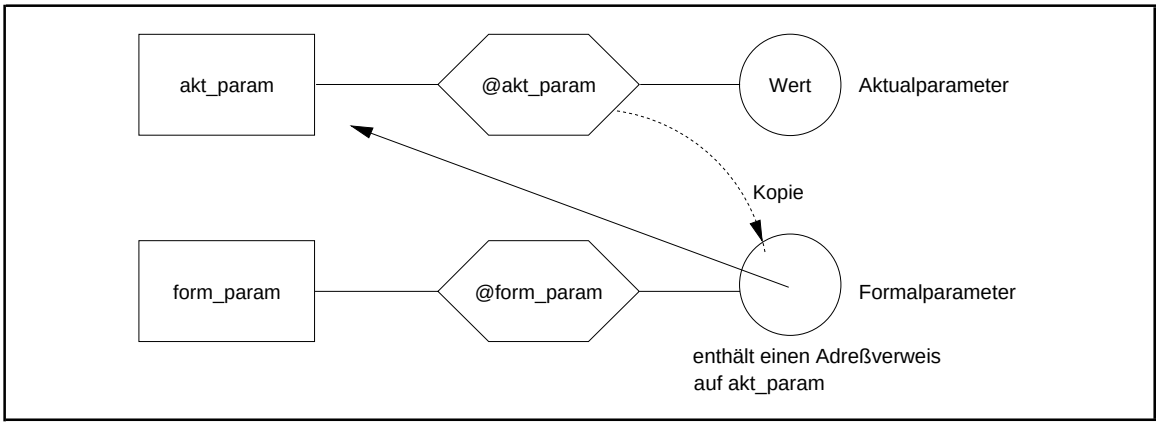


Abbildung 7.1-2: call-by-reference-Methode

[24] Die Sprache Ada beispielsweise enthält eigene Sprachkonstrukte (Spezifikation eines Formalparameters als **in**, **out** bzw. **inout)** für jede dieser Aufgaben.

Eine Besonderheit ist zu beachten: ***Wird innerhalb einer Prozedur ein call-by-reference-Parameter, z.B. mit Bezeichner `ref`, als Aktualparameter für einen weiteren (untergeordneten) Prozeduraufruf nach dem call-by-reference-Prinzip verwendet, so wird nicht die Adresse von `ref` an diesen Prozeduraufruf weitergegeben, wie es die call-by-reference-Methode vorschreibt, sondern der Wert von `ref`.*** Dieser Wert ist die Adresse des "äußeren" Aktualparameters, die somit der untergeordneten Prozedur mitgeteilt wird. Damit wirkt sich jeder Zugriff innerhalb der untergeordneten Prozedur direkt auf den Wert des äußeren Aktualparameters aus (siehe Abbildung 7.1-3).

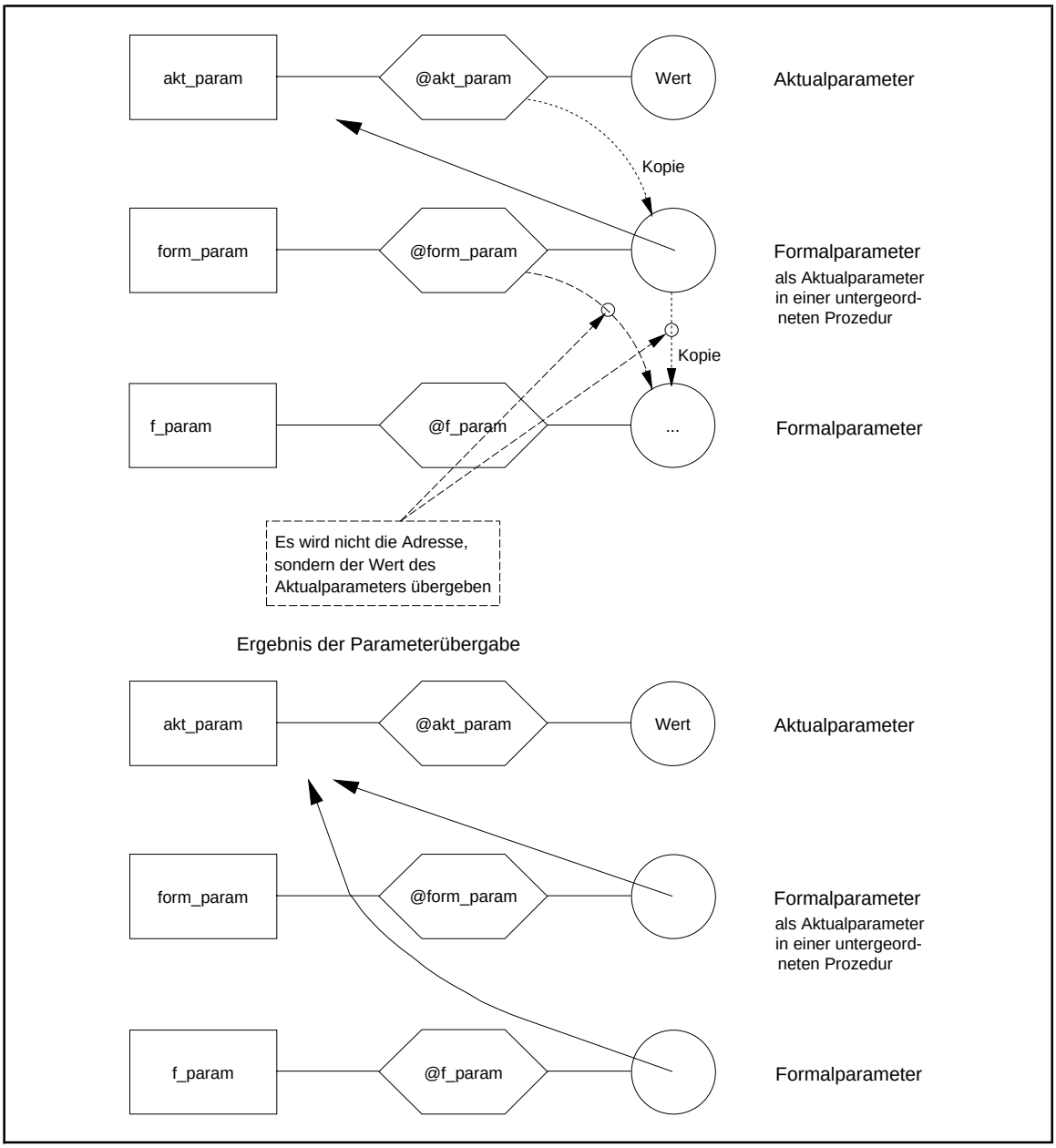


Abbildung 7.1-3: call-by-reference-Parameter als Aktualparameter

In Pascal wird jeder **Variablenparameter**, d.h. jeder Formalparameter, der im Prozedurkopf durch das Schlüsselwort **VAR** gekennzeichnet ist, nach der call-by-reference-Methode behandelt (der dazu korrespondierende Aktualparameter muß eine Variable sein):

PROCEDURE xyz (VAR call_by_ref_parameter : ...);

Um nur aktuelle Parameterwerte an eine Prozedur zu übergeben, eignet sich neben der call-by-reference-Methode im wesentlichen die call-by-value-Methode. Weitere Methoden, die z.B. in der Sprache Ada zur Implementierung von Eingabeparametern (**in**-Parameter) eingesetzt werden, sind call-by-constant-value und call-by-reference-value. Die Sprache COBOL-85 definiert eine weitere Variante: die call-by-content-Methode.

Bei der **call-by-value-Methode** (Abbildung 7.1-4) ist ein Formalparameter eine lokale Variable, die bei Eintritt in die Prozedur mit dem ***Wert des korrespondierenden Aktualparameters*** durch eine Kopie dieses Werts initialisiert wird. Der Aktualparameter kann eine Variable, eine Konstante oder ein Ausdruck sein, da nur der Wert des aktuellen Parameters interessiert; bei einem Ausdruck als Aktualparameter wird dieser vorher ausgewertet. Die Prozedur verändert den Wert des Aktualparameters nicht, auch wenn sie innerhalb des Prozedurrumpfs den Formalparameter mit einem neuen Wert versieht. Es werden also unerwünschte Nebeneffekte für Eingabeparameter von vornherein vermieden. Ein Nachteil dieser Methode besteht darin, daß im Falle eines strukturierten Datenobjekts als Formal- und Aktualparameter, etwa im Fall eines Feldes, eine Kopie des kompletten Aktualparameters in den Formalparameter übertragen werden muß, was u.U. sehr speicherplatz- und zeitaufwendig ist.

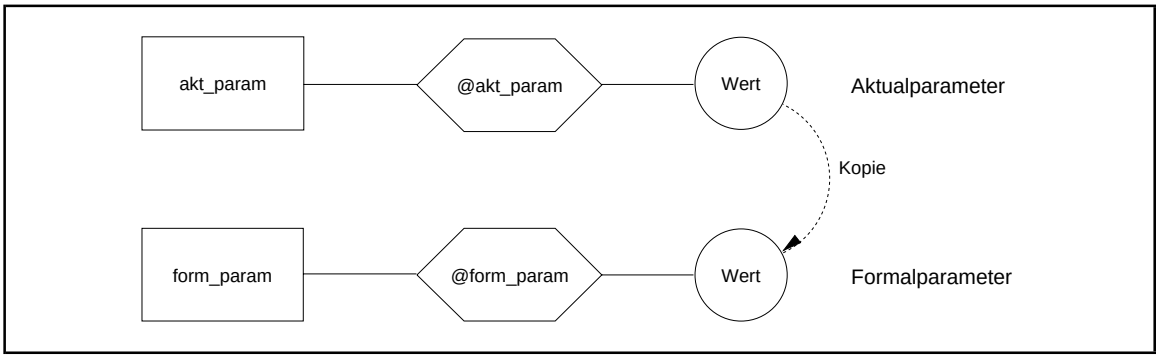


Abbildung 7.1-4: call-by-value-Methode

Bezüglich der Übergabe von Werten als Eingabeinformationen in ein Unterprogramm bietet die call-by-value-Methode natürlich eine größere Sicherheit (z.B. gegen Programmierfehler) als die call-by-reference-Methode, da bei der call-by-reference-Methode der Aktualparameter direkt, bei der call-by-value-Methode nicht der Aktualparameter selbst, sondern nur eine Kopie seines Werts verändert wird.

In Pascal ist für jeden **Wertparameter** die call-by-value-Methode implementiert. Ein derartiger Formalparameter ist in der Liste der Formalparameter im Prozedurkopf *nicht* mit dem Schlüsselwort VAR versehen.

In Borland Pascal werden als Ausnahme alle Wertparameter, die intern mehr als 4 Bytes belegen, nicht nach der call-by-value-Methode, sondern nach der call-by-reference-Methode übergeben, um genau die beschriebene Ineffizienz bei der Übergabe "großer" Aktualparameter zu vermeiden. Diese Ausnahmeregel gilt nicht für Wertparameter vom Real-Typ. Der vom Compiler für den Zugriff auf derartige Formalparameter erzeugt Code bewirkt jedoch, daß sich der Parameter "nach außen hin" wie ein normaler call-by-value-Parameter verhält, d.h. daß insbesondere durch die Prozedur der Wert des korrespondierenden Aktualparameter beim Aufrufer nicht verändert wird.

Die Sprache C kennt nur die call-by-value-Methode, so daß der Anwender durch Übergabe von Pointern selbst dafür sorgen muß, daß Werte von Aktualparametern durch eine Prozedur verändert werden können. C++ hat (in Anlehnung an Pascal) sowohl die call-by-value- als auch die call-by-reference-Methode.

Bei der **call-by-constant-value-** und der **call-by-reference-value-Methode** als Spezialformen der call-by-value-Methode fungieren die Formalparameter als lokale Konstanten, deren Werte beim Eintritt in die Prozedur mit den Werten der korrespondierenden Aktualparameter initialisiert werden und die während des Ablaufs der Prozedur nicht geändert werden können. Borland Pascal bezeichnet einen Formalparameter, der nach der call-by-constant-value-Methode behandelt wird, als **konstanten Parameter**; ein derartiger Formalparameter wird durch Voranstellen des Schlüsselworts `CONST` gekennzeichnet.

Die **call-by-content-Methode** (in COBOL-85) erweitert die ursprünglich in COBOL ausschließlich vorgesehene call-by-reference-Methode. Bei Aufruf einer Prozedur wird für jeden Aktualparameter eine für den Anwender nicht sichtbare Kopie angelegt. Diese Kopie wird dem Unterprogramm als Aktualparameter nach dem call-by-reference-Prinzip übergeben, so daß innerhalb der Prozedur jeder Formalparameter als call-by-reference-Parameter behandelt werden kann. Zum Aufrufer stellt sich dieses Prinzip wie die call-by-value-Methode dar (Abbildung 7.1-5).

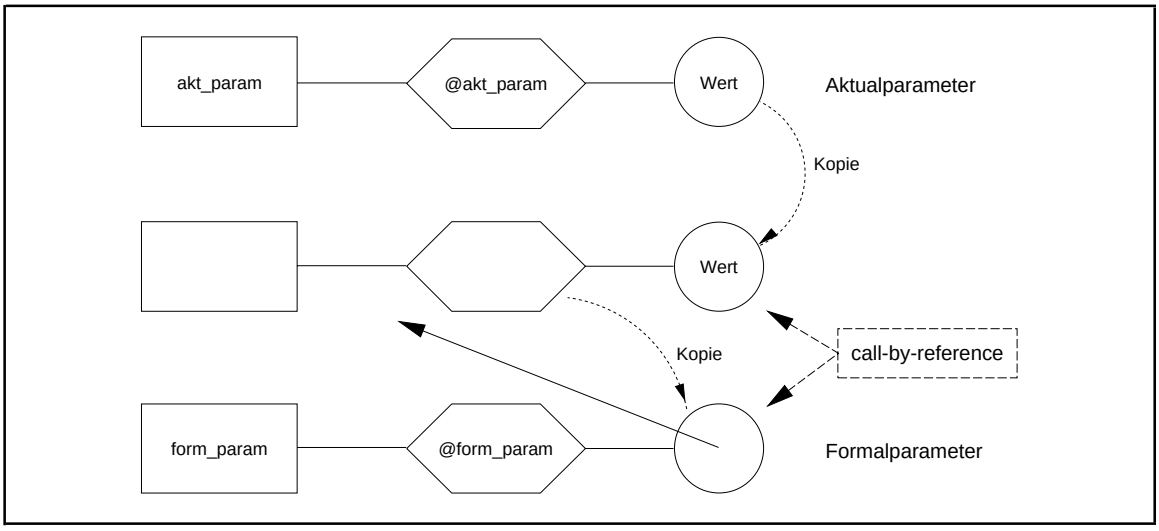


Abbildung 7.1-5: call-by-content-Methode

Die im folgenden beschriebene call-by-result-Methode eignet sich neben der call-by-reference-Methode dazu, um Werte, die eine Prozedur ermittelt hat, an den Aufrufer der Prozedur zurückzugeben.

Bei der **call-by-result-Methode** wird der Formalparameter wie eine nicht-initialisierte lokale Variable benutzt, der während der Ausführung der Prozedur im Prozedurrumpf ein Anfangswert zugewiesen wird, der dann auch verändert werden kann. Am Ende der Prozedur wird der Wert des Formalparameters an den korrespondierenden Aktualparameter übergeben, der eine Variable (keine Konstante und kein Ausdruck) sein muß (Abbildung 7.1-6).

Die call-by-result-Methode wird in Ada bei der Implementierung von **out**-Parametern eingesetzt. Hierbei wird die Adresse des Aktualparameters, der zu einem Formalparameter gemäß call-by-result gehört, *bei Eintritt* in die Prozedur bestimmt; diese Adresse wird dann am Ende der Prozedur zur Rückgabe des Werts des Formalparameters verwendet. Andere Implementierungen dieser Methode sehen vor, diese Adresse erst unmittelbar vor Rückgabe des Werts des Formalparameters, also bei *Verlassen* der Prozedur, zu bestimmen. Beide Ansätze führen in einigen speziellen Situationen zu unterschiedlichen Ergebnissen.

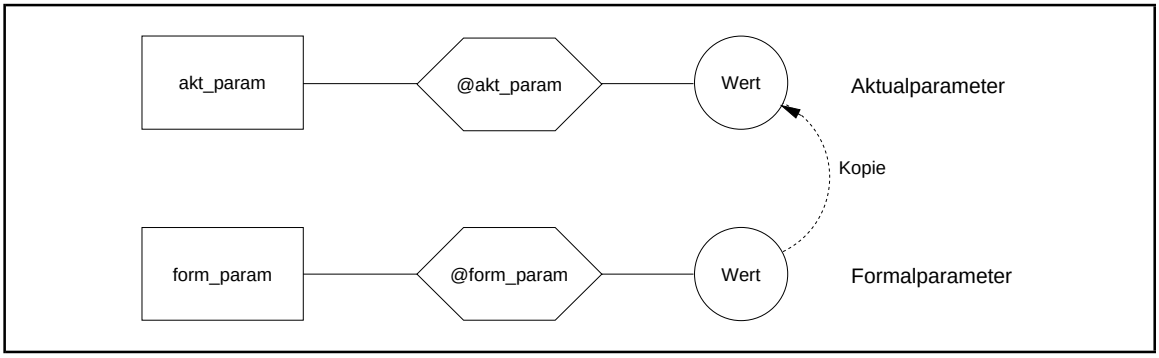


Abbildung 7.1-6: call-by-result-Methode

Für Formalparameter, denen Werte von einem Aufrufer übergeben werden, die dann während der Ausführung der Prozedur verändert und an den Aufrufer zurückgegeben werden können, eignet sich neben der beschriebenen call-by-reference-Methode die **call-by-value-result-Methode**.

Der Formalparameter wird bei dieser Methode wieder als lokale Variable behandelt, die bei Eintritt in die Prozedur mit dem gegenwärtigen Wert des korrespondierende Aktualparameters initialisiert wird. Innerhalb der Prozedur beeinflussen Änderung des Werts des Formalparameters nur diese lokale Kopie (entsprechend der call-by-value-Methode). Am Ende der Prozedur wird der gegenwärtige Wert des Formalparameters an den Aktualparameter zurückgegeben (Abbildung 7.1-7).

Auch hier kommt es wie bei der call-by-result-Methode wieder darauf an, wann die Adresse des Formalparameters ermittelt wird, nämlich beim Eintritt in die Prozedur oder unmittelbar vor dem Verlassen des Unterprogramms.

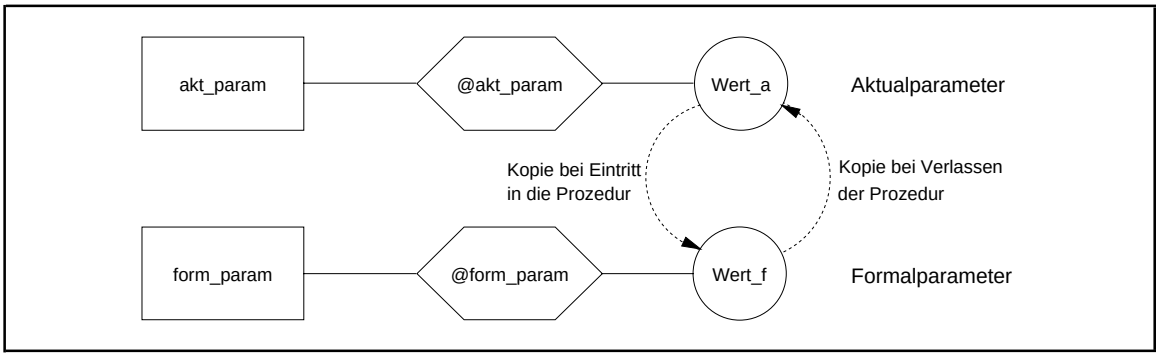


Abbildung 7.1-7: call-by-value-result-Methode

Call-by-value-result wird bei Ada für skalare **in-out**-Parameter verwendet.

In den meisten Fällen ergeben die call-by-value-result- und die call-by-reference-Methode bei vollständiger Abarbeitung einer Prozedur die gleichen Resultate. Ausnahmen stellen Situationen dar, in denen Interaktionen zwischen Parametern und nicht-lokalen Variablen vorkommen. Ein Programmierer sollte aber darauf achten, die Prozeduren so zu schreiben, daß ihre Ergebnisse nicht von den zugrundeliegenden Methoden der Parameterübergabe abhängen.

Call-by-value-result hat die gleichen Nachteile bezüglich der Ineffizienz bei strukturierten Datenobjekten als Parameter wie call-by-value. In diesen Fällen ist die call-by-reference-Methode vorzuziehen. Andererseits hat call-by-value-result gegenüber call-by-reference den Vorteil, daß sich alle Referenzen innerhalb des Unterprogramms auf lokale Variablen (die Formalparameter) beziehen und damit schneller ablaufen, als die bei call-by-reference benutzten indirekten Referenzen auf nicht-lokale Objekte (die Aktualparameter). Eine weitere Konsequenz ergibt sich im Falle eines Abbruchs der Prozedur noch vor ihrem Ende aufgrund eines Fehlers: Bei call-by-reference kann der aktuelle Parameter inzwischen einen durch die Prozedur veränderten Wert haben, während bei call-by-value-result der Wert des Aktualparameters noch der gleiche ist, wie bei Eintritt in die Prozedur, da ja der Wert des Aktualparameters erst am Ende der Prozedur, das noch nicht erreicht ist, aktualisiert wird.

Abschließend soll noch die **call-by-name-Methode** erwähnt werden, die in ALGOL 60 definiert wurde, aber in heutigen Programmiersprachen nicht verwendet wird, da sie sehr viel aufwendiger zu implementieren ist als call-by-reference und ansonsten nur in ausgefallenen Situationen Vorteile bietet.

Bei der call-by-reference-Methode wird die Adresse des Aktualparameters zum Zeitpunkt des Aufrufs bestimmt. Bei call-by-name wird die Adresse des Aktualparameters erst dann bestimmt, wenn der korrespondierende Formalparameter auch benutzt wird, und zwar wird bei *jeder Verwendung des Formalparameters* während des Prozedurlaufs diese *Auswertung erneut* durchgeführt. Diese Methode ermöglicht es, daß unterschiedliche Adressen für verschiedene Zugriffe auf denselben Formalparameter während der Ausführung einer Prozedur verwendet werden.

Die folgende Tabelle faßt die Möglichkeiten der beschriebenen Methoden zur Parameter-

übergabe an Unterprogramme zusammen:

Methode	Parameter geeignet zur		
call-by-	Übergabe von aktuellen Werten an das Unterprogramm	Rückgabe von aktuellen Werten aus dem Unterprogramm	Übergabe und Rückgabe von aktuellen Werten an das Unterprogramm bzw. aus dem Unterprogramm
reference	x	x	x
value / content	x		
constant-value	x		
reference-value	x		
result		x	
value-result	x	x	x

Das folgende Beispiel aus [W/C] erläutert die einzelnen Methoden noch einmal. Es zeigt in Abbildung 7.1-8 eine Prozedur mit Bezeichner `methoden`, die im Pascal-Stil geschrieben ist. In diesem Beispiel soll jedoch der Aktualparameter an die Prozedur nach den verschiedenen beschriebenen Methoden zu übergeben sein (also nicht nur nach der in Pascal in dieser Situation gemäß der Syntax verwendeten call-by-value-Methode). Je nach Methode ergeben sich u.U. verschiedene Resultate. Es muß betont werden, daß der "trickreiche" Programmierstil dieses Beispiels zu vermeiden ist, zumal das Unterprogramm auch noch globale Variablen verwendet; das Beispiel dient lediglich der Demonstration der Unterschiede der einzelnen Methoden.

Innerhalb von `methoden` werden `a[1]` und `element` als globale Variablen behandelt; je nach Methode kann auf `a[1]` bzw. `a[2]` über den Formalparameter `x` zugegriffen werden.

Wenn in diesem Beispiel der Parameter nach der call-by-result-Methode übergeben wird, entsteht ein Fehler, da der Formalparameter `x` vor seiner Verwendung auf der rechten Seite des Ausdrucks

```
x := x + 3
```

nicht initialisiert worden ist.

Wenn der Parameter `x` nach der call-by-reference-Methode übergeben wird, kann auf `a[1]` auf zwei unterschiedliche Arten zugegriffen werden, nämlich entweder direkt als globales Datenobjekt in

```
a[1] := 6
```

oder indirekt als Formalparameter in

```
x := x + 3.
```

Bei der call-by-value-Methode beeinflußt die Anweisung

```
x := x + 3
```

den Wert von `a[1]`, des aktuellen Parameters, nicht, da in `methoden` mit einer lokalen Kopie von `a[1]` gearbeitet wird.

```
PROGRAM beispiel;

TYPE feld_typ = ARRAY [1..2] OF INTEGER;

VAR element : INTEGER;
    a       : feld_typ;

PROCEDURE methoden (x : INTEGER);

  BEGIN { methoden }
    a[1]    := 6;
    element := 2;
    x       := x + 3
  END   { methoden };

BEGIN { beispiel }
  a[1]    := 1;
  a[2]    := 2;
  element := 1;
  methoden (a[element])
END   { beispiel }.
```

Resultate bei den verschiedenen Methoden der Parameterübergabe

Methode	a[1]	a[2]	element
call-by-reference	9	2	2
call-by-value	6	2	2
call-by-value-result (Parameterbestimmung bei Verlassen)	6	4	2
call-by-value-result (Parameterbestimmung bei Eintritt)	4	2	2

Abbildung 7.1-8: Parameterübergabemethoden (Beispiel)

Die Ursache für Unterschiede, die sich bei den beiden call-by-value-result-Methoden ergeben, liegt im Bestimmungszeitpunkt der Adresse für die Rückgabe des Werts im Formalparameter an den korrespondierenden Aktualparameter. Wird diese Adresse, nämlich die Adresse des Aktualparamters `a[element]`, beim Eintritt in die Prozedur bestimmt, dann handelt es sich um die Adresse von `a[1]`. Wird die Adresse unmittelbar vor Verlassen der Prozedur ermittelt, dann ist aufgrund der Anweisung

```
element := 2
```

die Adresse von `a[element]` die Adresse von `a[2]`. Der gegenwärtige Wert von `x` wird also entweder an `a[1]` oder an `a[2]` zurückgegeben, nachdem er in beiden Fällen bei Eintritt in die Prozedur mit dem Wert von `a[1]` initialisiert wurde.

7.2 Der Stack

In Kapitel 3.4 wurde bereits das Laufzeit-Layout eines Programms beschrieben. Insbesondere wurde der Stack erwähnt (Abbildung 3.4-2), der während der Laufzeit u.a. alle lokalen Variablen der aufgerufenen Prozeduren aufnimmt. In diesem Kapitel sollen nun weitere Details zur Behandlung (dynamischer) Aufruffolgen von Prozeduren mit Hilfe des Stackprinzips beschrieben werden. Außerdem wird gezeigt, wie das Lebensdauerkonzept von Datenobjekten (Kapitel 6.2) realisiert wird.

Der **Stack (Stapel, Keller)** ist ein Datenbereich mit (strukturierten und variabel langen) Einträgen, der so organisiert ist, daß immer nur auf den zuletzt eingetragenen Eintrag zugegriffen werden kann (lesen, entfernen). Der Stack kann durch die Reservierung eines fest vereinbarten Datenbereichs und eines global zugreifbaren Datenobjekts implementiert werden, der im folgenden mit dem Bezeichner **SP (Stackpointer)** bezeichnet wird. Befinden sich Daten im Stack, so belegen sie einen zusammenhängenden Bereich; der restliche Abschnitt des Stacks ist frei. Der Wert von SP identifiziert den zuletzt in den Stack aufgenommenen Eintrag. Die Einträge im Stack sind im allgemeinen strukturiert und unterschiedlich groß. Innerhalb jedes Eintrags steht an einer fest definierten Stelle ein Verweis auf den Eintrag, der zeitlich genau vorher in den Stack eingetragen wurde (bei gleichartig strukturierten Stackeinträgen kann man diese hintereinander schreiben; der vorherige Eintrag steht dann in einer konstanten Distanz, entsprechend der Länge eines Stackeintrags, zum Stackeintrag, der durch den Wert von SP identifiziert wird).

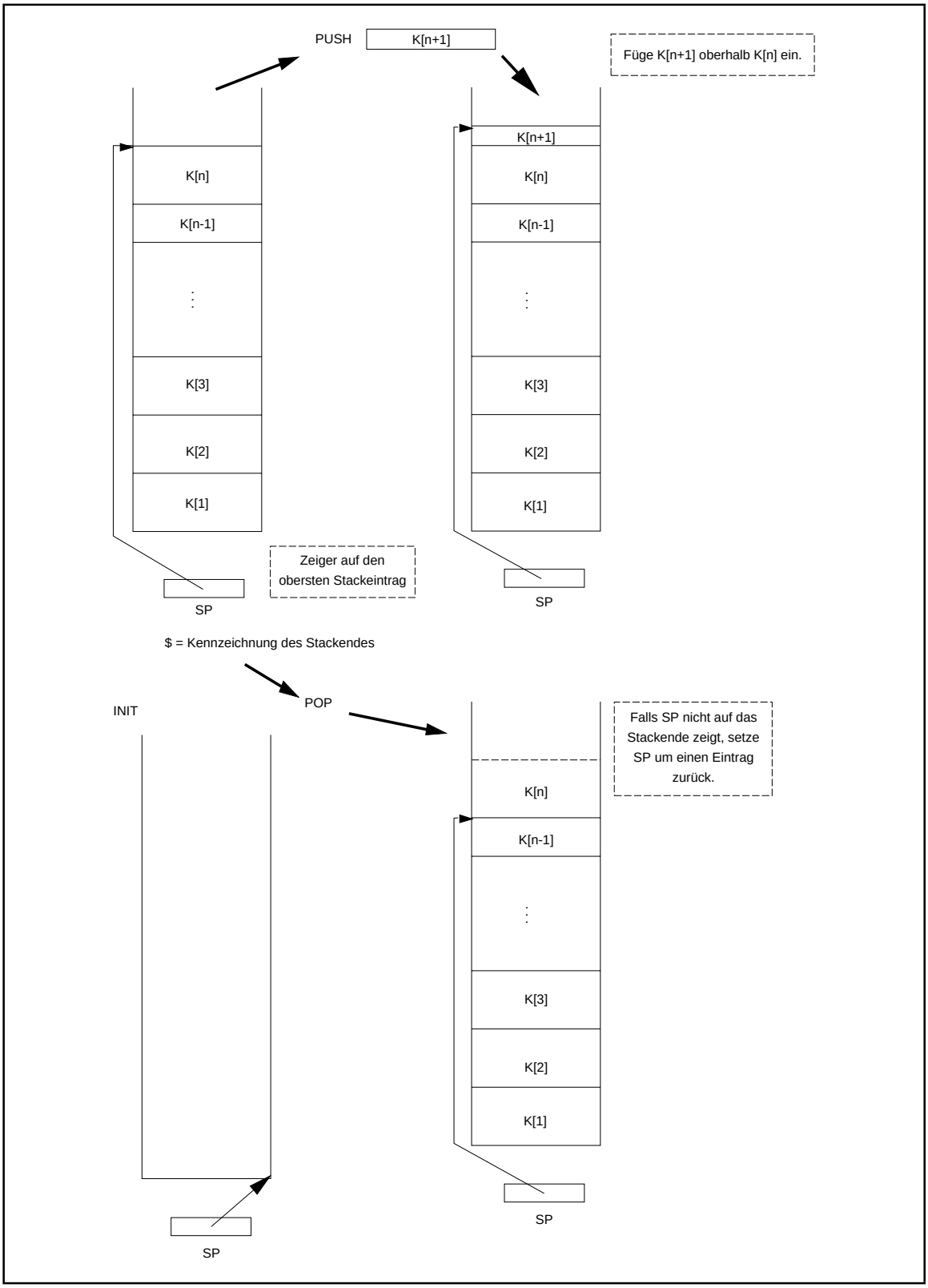


Abbildung 7.2-1: Stack

Mit einem Stack sind im wesentlichen **drei Operationen** mit folgender Wirkung auf den Stack verbunden (Abbildung 7.2-1):

- die Operation **INIT**, die einen leeren Stack initialisiert

- die Operation **PUSH**, die einen zusätzlichen Eintrag im Stack vornimmt und den Inhalt von SP so aktualisiert, daß er nun den neuen Eintrag identifiziert (es soll angenommen werden, daß die Anzahl an Bytes für den neuen Stackeintrag aus dem Stackeintrag selbst ermittelt werden kann)

- die Operation **POP**, die den obersten Stackeintrag vom Stack entfernt und den Inhalt von SP entsprechend verändert, falls der Stack nicht leer ist; bei einem leeren Stack hat die Operation POP keine Wirkung.

In den meisten Implementierungen eines Stacks wächst der belegte Teil des Stacks bei der Operation PUSH von hohen zu niedrigeren Adressen und nimmt bei der Operation POP in Richtung auf hohe Adressen ab. Diese Implementierung ist auch in Abbildung 7.2-1 gewählt. Andere Implementierungen eines Stacks sind jedoch auch möglich, wobei Implementierungsdetails für das Stackprinzip zunächst ohne Belang sind.

Die Operation PUSH stapelt also einen weiteren Eintrag auf den Stack; die Operation POP entfernt einen Eintrag vom oberen Ende des Stapels (**LIFO-Prinzip, last-in-first-out-Prinzip**). Das bedeutet, daß nach einer POP-Operation der Zugriff auf entfernte Einträge nicht mehr besteht und auch nicht wiederherstellbar ist, auch wenn sie physisch nicht gelöscht werden.

Zur Illustration der Anwendung des Stackprinzips im Zusammenhang mit dem Prozedurkonzept dienen wieder in Pascal geschriebene Beispiele. Es muß betont werden, daß es sich bei der im folgenden dargestellten Methode um *eine mögliche Umsetzung des Konzepts* handelt, die je nach Programmiersprache und Compiler variiert.

Die in einem Programm definierten Datenobjekte lassen sich bezüglich ihrer logischen Zugehörigkeit zu Programmteilen unterscheiden in

- ***globale Datenobjekte des Hauptprogramms***, die von allen Blöcken des Programms aus sichtbar, d.h. zugreifbar sind, wenn nicht innerhalb eines Blocks eine Deklaration mit demselben Bezeichner erfolgt (vgl. Kapitel 6.1)

- ***globale Datenobjekte*** aus der Sicht einer Prozedur, nämlich solche, die in Blöcken deklariert sind, die die Prozedur (evtl. über mehrere Hierarchiestufen) einschachteln und in der Prozedur nicht durch Deklarationen mit denselben Bezeichnern überlagert werden

- ***lokale, innerhalb einer Prozedur oder Funktion definierte***, nur dieser Prozedur zugehörige ***Datenobjekte***; diese sind für alle darin eingebettete Prozeduren und Funktionen global, nach außen aber nicht sichtbar

- ***Formalparameter einer Prozedur oder Funktion***, die wie lokale Datenobjekte einer Prozedur betrachtet werden

- ***Aktualparameter für den Aufruf einer Prozedur oder Funktion***, die zum Aufrufer der Prozedur oder Funktion gehören.

Auf lokale Datenobjekte einer Prozedur (Funktionen werden wieder als syntaktisch spezielle Prozeduren angesehen) wird erst zugegriffen, wenn diese Prozedur während der Laufzeit auch aufgerufen wird. Daher braucht Speicherplatz für diese Datenobjekte erst bei Aufruf der Prozedur entsprechend dem Beginn ihrer Lebensdauer zugewiesen werden. Falls die Prozedur also gar nicht aufgerufen wird, braucht auch kein Speicherplatz für lokale Datenobjekte reserviert zu werden. Bei Verlassen der Prozedur endet die Lebensdauer der lokalen Datenobjekte, und damit kann der für sie verwendete Speicherplatz wieder freigegeben werden. Der Speicherplatz für globale Datenobjekte des Hauptprogramms wird beim Laden des Programms reserviert.

Bei Programmstart wird der Stack als leer initialisiert (Operation INIT).

Bei jedem Aufruf einer Prozedur wird ***vom Aufrufer und von der aufgerufenen Prozedur*** mit Hilfe von PUSH-Operationen ein neuer Stackeintrag erzeugt, der als **Aktivierungsrecord (des Aufrufs)** bezeichnetet wird. Der Compiler generiert sowohl beim Aufrufer als auch bei der aufgerufenen Prozedur entsprechenden Code. Der bei Einsprung in eine Prozedur erzeugte Aktivierungsrecord wird beim Rücksprung zum Aufrufer von der gerufenen Prozedur wieder entfernt (Operation POP). Details des Ablaufs werden in Kapitel 7.5 beschrieben.

Der Aktivierungsrecord eines Prozeduraufrufs enthält in einer implementierungsabhängigen Reihenfolge:

- die ***Rücksprungadresse zum Aufrufer***

- die ***Speicherplatzreservierung für jedes lokale Datenobjekt***, d.h. für jeden ***Formalparameter*** und jede in der Prozedur deklarierte ***lokale Variable***; Formalparameter sind bei Start des Prozedurcodes bereits vom Aufrufer mit den Werten der Aktualparameter entsprechend der jeweiligen Parameterübergabemethode initialisiert worden

- das **Display**; hierbei handelt es sich um eine Tabelle aus Verweisen auf diejenigen, vorher im Stack eingetragenen Aktivierungsrecords, die aus Sicht des jeweiligen Prozeduraufrufs globale Datenobjekte enthalten. Die Display-Informationen resultieren aus der Analyse der statischen Blockstruktur, die zur Übersetzungszeit durchgeführt wird. Sie werden zur Laufzeit der aufgerufenen Prozedur erzeugt (siehe Kapitel 7.5)

- ***Einträge zur Verwaltung des Stacks***, insbesondere einen Adreßverweis auf den vorherigen Stackeintrag.

Während also die globalen Variablen des Hauptprogramms in der Regel in einem eigenen Datenmodul liegen, werden die lokalen Datenobjekte einer Prozedur im Stack verwaltet[25].

Während des Prozedurablaufs können evtl. weitere Einträge im Stack vorgenommen werden (Operation PUSH), z.B.

- ***Sicherstellungsbereiche für Registerinhalte***: wenn innerhalb des Prozedurcodes Register verwendet werden, müssen deren Inhalte natürlich zuvor (lokal) gesichert und vor dem Rücksprung zum Aufrufer rekonstruiert werden, damit der Aufrufer dieselben Registerinhalte wie unmittelbar vor Eintritt in die Prozedur sieht

- Speicherplatz für die ***lokale Speicherung von Zwischenergebnissen***.

Sebstverständlich werden auch diese Einträge vor dem Rücksprung zum Aufrufer wieder entfernt.

Am Beispiel der in Abbildung 6.2-1 dargestellten Aufruffolge soll die prinzipielle Belegung des Stacks gezeigt werden (Abbildung 7.2-2). In diesem Fall finden keine rekursiven Prozeduraufrufe statt, und alle Prozeduren sind parameterlos. Ein Aktivierungsrecord wird durch ein Kästchen angedeutet, das lediglich die jeweiligen Datenobjekte mit ihren im Programm verwendeten Bezeichnern enthält; auf allen weiteren Einträge eines Aktivierungsrecords wie Rücksprunggadresse, Display usw. wird in der Darstellung der Übersichtlichkeit halber verzichtet. Die **aktuelle Stackbelegung** beginnt an der Stelle, auf die SP zeigt, und endet in der Darstellung unterhalb am Stackanfang. Wichtig ist, daß über SP immer nur auf den Eintrag (das Kästchen) direkt zugegriffen werden kann, auf das SP zeigt.

[25] Dem Konzept entsprechend könnten die globalen Variablen ebenfalls im Stack als erster Eintrag abgelegt werden.

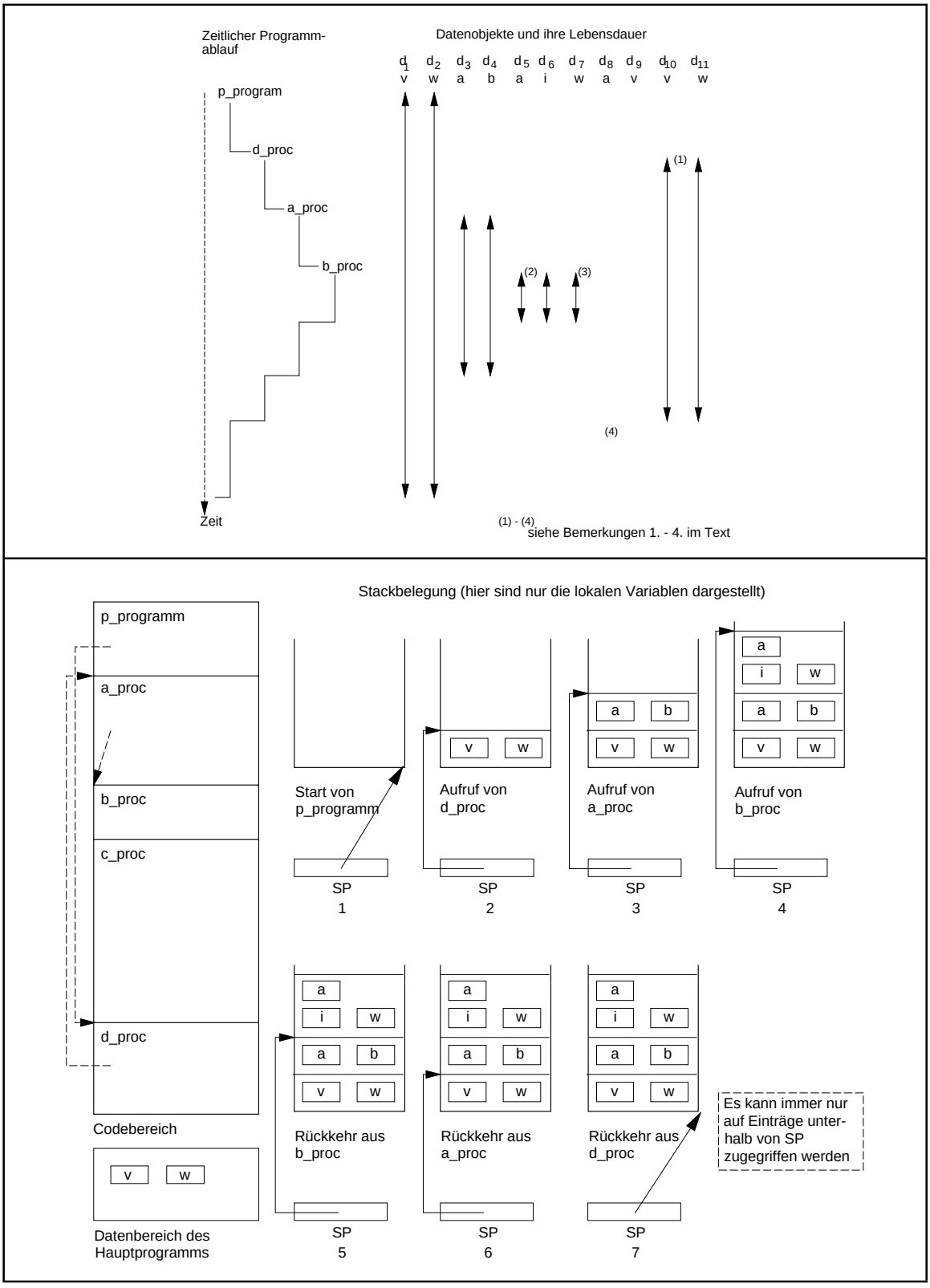


Abbildung 7.2-2: Stackbelegung bei einer Aufruffolge

Der Gültigkeitsbereich eines Bezeichners und das Konzept der Lebensdauer eines Datenobjekts sind hier noch einmal verdeutlicht: Ein Bezeichner bezieht sich immer auf das Datenobjekt, das im Stack in Richtung auf den Stackanfang (in Abbildung 7.2-2) nach unten) mit diesem Bezeichner zuerst auftritt. Gibt es im Stack kein Datenobjekt mit diesem Bezeichner, so wird ein Datenobjekt im Datenbereich des Hauptprogramms benannt. Solange ein Datenobjekt in der aktuellen Stackbelegung oder im Datenbereich des Hauptprogramms vorhanden ist, lebt es. Die Lebensdauer eines Datenobjekts im Stack endet, wenn der Aktivierungsrecord mit der Operation POP "entfernt" wird.

Neben der Zugriffsberechtigung auf ihre lokalen Datenobjekte besteht für eine Prozedur eine Zugriffsberechtigung auf die in diesem Augenblick aus Sicht der Prozedur für sie globalen Datenobjekte. Während eines Prozeduraufrufs können das Datenobjekte im Hauptprogramm bzw. Datenobjekte sein, die zu übergeordneten, noch nicht beendeten Prozeduraufrufen gehören. Im obigen Beispiel hat die Prozedur `b_proc` während ihres Ablaufs die Zugriffsberechtigung auf ihre eigenen lokalen Datenobjekte d_5, d_6 und d_7, die mit `a`, `i` bzw. `w` bezeichnet sind, und (aus ihrer Sicht global) auf die Datenobjekte d_4 mit Bezeichner `b` in `a_proc` und d_1 mit Bezeichner `v` im Hauptprogramm `p_programm`. Die Datenobjekte d_5, d_6 und d_7 stehen im Aktivierungsrecord, auf den SP gerade zeigt, das Datenobjekt d_4 im Aktivierungsrecord, der bei Aufruf von `a_proc` angelegt wurde, und das Datenobjekt d_1 im Datenbereich des Hauptprogramms. Die Anfangsadresse des globalen Datenbereichs für das Hauptprogramm und der Wert von SP, der bei Aufruf von `a_proc` aktuell war, d.h. die Adresse des Aktivierungsrecords für den übergeordneten Aufruf von `a_proc`, sollten `b_proc` also bekannt sein, um auf diese für `b_proc` globalen Datenobjekte zugreifen zu können. Der Aktivierungsrecord beispielsweise, der die lokalen Datenobjekte für den Aufruf von `d_proc` enthält, ist für `b_proc` ohne Belang, da `b_proc` keine globalen Datenobjekte in `d_proc` hat.

7.3 Rekursive Prozeduren: Das Prinzip

Eine Prozedur bzw. Funktion heißt **(direkt) rekursiv**, wenn innerhalb ihres Prozedurrumpfs ein Aufruf der Prozedur selbst erfolgt. Sie heißt **indirekt rekursiv**, wenn sie eine Prozedur aufruft, die evtl. weitere Prozeduren anstößt, und dadurch eine Aufrufkette ineinandergeschachtelte Prozeduraufrufe beginnt, in deren Ablauf die äußere Prozedur selbst wieder aufgerufen wird.

Programmiersprachen wie Pascal, C, C++ lassen grundsätzlich (direkte und indirekte) rekursive Prozeduraufrufe zu, während derartige Aufrufketten, an denen COBOL-Programme beteiligt sind, im allgemeinen während der Laufzeit zu Fehlern führen. Das liegt im wesentlichen daran, daß in COBOL häufig Unterprogrammaufrufe nicht mit Hilfe eines Stacks realisiert werden (siehe dazu [HOF]).

Das Implementierungsprinzip rekursiver Prozeduren ergibt sich aus dem generellen Implementierungsprinzip von Prozeduren und wird im folgenden beschrieben. Kapitel 7.4 schließt mit einigen ***Anwendungen*** rekursiver Prozeduren am Beispiel von Pascal-Prozeduren an.

Programmiersprachen, die rekursive Prozeduraufrufe zulassen, unterscheiden nicht zwischen "normalen" und rekursiven Prozeduren. Das Implementierungsprinzip mittels Stack erfordert diese Unterscheidung auch nicht, so daß hier eine Beschränkung auf direkt rekursive Prozeduren erfolgen kann.

Die **allgemeine Form einer rekursiven Prozedur** geht davon aus, daß der Prozedurablauf vom Wert eines Parameters abhängt, meist ein Formalparameter der Prozedur, der abzählbare Werte annehmen kann. ***Dieser Parameter kontrolliert die Rekursionstiefe.*** Beispiele sind Parameter vom Integer-Typ, die Anzahl der Einträge verketteter Listen, ein Ausschnitt eines ARRAYs usw. Für einen "kleinen" Wert dieses Parameters wird die Prozedur ohne weitere Aufrufe der Prozedur selbst abgearbeitet. Für einen "großen" Wert des Parameters wird innerhalb des Prozedurrumpfs die Prozedur erneut aufgerufen, jetzt aber mit einem um mindestens 1 verminderten Wert des Parameters. Natürlich muß im äußeren Aufruf sichergestellt werden, daß der die Rekursion steuernde Parameter in seinem Wert nicht unterhalb des "kleinen" Werts fällt, sonst kommt es zu einer unendlichen Aufruffolge[26]. ***Die korrekte Handhabung des Parameters zur Kontrolle der Rekursionstiefe ist natürlich Sache der Anwendung.***

Das folgende Beispiel (Abbildung 7.3-1) zeigt eine Prozedur `rek_proc` mit einem call-by-value-Formalparameter mit Bezeichner `n` vom Typ `INTEGER`, der den Verlauf der Rekursion steuert. Die Wirkungsweise von `rek_proc` ist in diesem Beispiel von untergeordneter Bedeutung; es sollen lediglich die Aufrufschachtelung und die Stackbelegung verfolgt werden. Die Prozedur wird in einem Programm mit `rek_proc(2)` aufgerufen. Wieder wird im Stack nicht der komplette Aktivierungsrecord eines Aufrufs gezeigt, sondern nur der Formalparameter `n`, der ja eine lokale Variable von `rek_proc` ist, und die Rücksprungadresse zum Aufrufer. Pro Aufruf von `rek_proc` wird ein eigenes Datenobjekt (mit Bezeichner `n`) für den Formalparameter im Stack eingerichtet. Die Regeln über den Gültigkeitsbereich von Namen besagen, daß innerhalb der Prozedur mit dem Bezeichner `n` immer das zuletzt eingerichtete Datenobjekt (im Stackeintrag, auf den SP zeigt) gemeint ist. Pro Aufruf gibt es also eine eigene "Inkarnation" der Variablen `n` im Stack. Das Datenobjekt lebt jeweils solange, bis der Aufruf, zu dem es gehört, abgeschlossen ist. Selbstverständlich ist der Code der Prozedur `rek_proc` nur einmal vorhanden, so daß die Datenobjekte zur Aufnahme der Rücksprungadresse zum Aufrufer für jeden rekursiven Aufruf denselben Wert beinhalten.

[26] Nach endlich vielen rekursiven Aufrufen läuft das Programm auf einen Speicherplatzfehler (stack overflow).

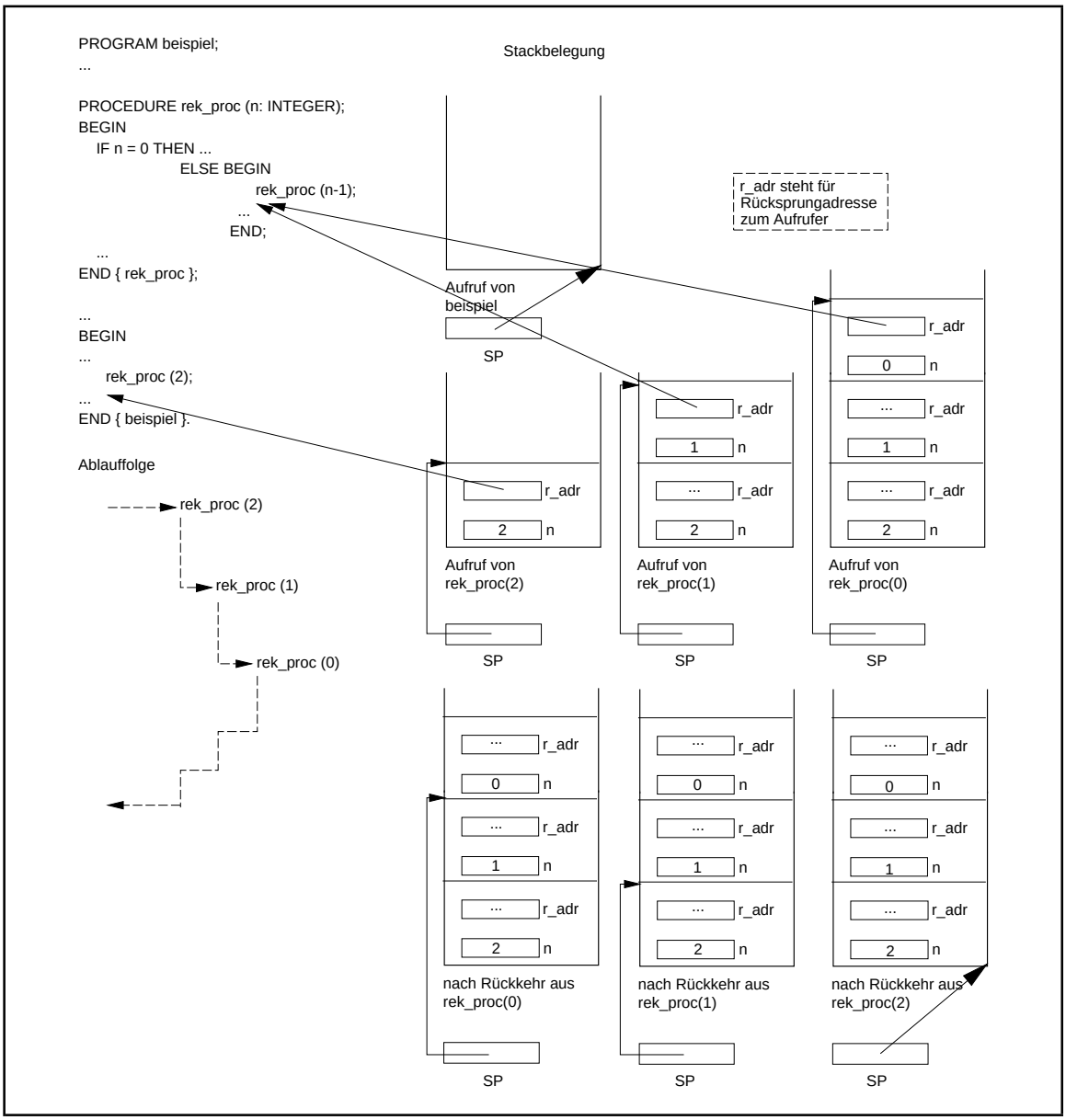


Abbildung 7.3-1: Stackbelegung bei einer rekursiven Prozedur

7.4 Beispiele rekursiver Prozeduren

Typischerweise läßt sich die Berechnung der **Fakultätsfunktion**, die durch

$$f(n) = \begin{cases} 1 & \text{für } n = 0 \\ f(n-1) \cdot n & \text{für } n > 0 \end{cases}$$

definiert ist, direkt in eine rekursive `PROCEDURE fakultaet_rekursiv` bzw. noch kürzer in eine rekursive `FUNCTION f` umsetzen, so wie in Abbildung 7.3-1 beschrieben. Zum Vergleich wird auch eine nicht-rekursive Version der Fakultätsfunktion als `PROCEDURE fakultaet_iterativ` angegeben.

```
PROCEDURE fakultaet_rekursiv (n            : INTEGER;
                              VAR resultat : LongInt);

{ rekursive Version zur Berechnung von n! }

BEGIN { fakultaet_rekursiv }
  IF n = 0
  THEN resultat := 1
  ELSE BEGIN
         fakultaet_rekursiv (n-1, resultat);
         resultat := resultat * n
       END
END { fakultaet_rekursiv } ;
```

bzw.

```
FUNCTION f (n: INTEGER): LongInt;

BEGIN { f }
  IF n = 0 THEN f := 1
           ELSE f := n * f(n-1)
END   { f };
```

bzw.

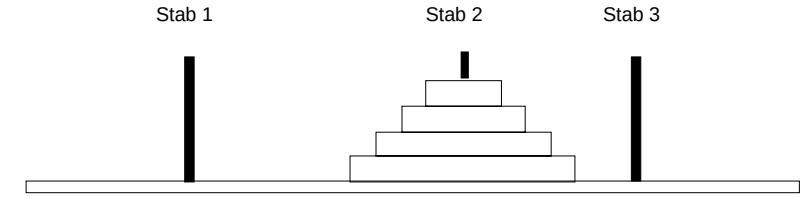
```
PROCEDURE fakultaet_iterativ (n            : INTEGER;
                              VAR resultat : LongInt);

VAR i : INTEGER;
    j : LongInt;

BEGIN { fakultaet_iterativ }
  i := 1;
  j := 1;
  WHILE i <= n DO
    BEGIN
      j := j*i;
      Inc(i);
    END;
  resultat := j
END { fakultaet_iterativ };
```

Natürlich kann jede Programmieraufgabe auch nicht-rekusiv gelöst werden. Die Umsetzung einer rekursiven Prozedur mit Hilfe des Stacks kann prinzipiell eine Anleitung dazu sein, jedoch wird meist ein zur rekursiven Version unterschiedlicher Ansatz zu einer einfacheren nicht-rekursiven Lösungsprozedur führen. Andererseits gibt es Programmieraufgaben, für die es leicht zu überblickende rekursive Lösungen gibt, bei denen nicht-rekursive Lösungsansätze jedoch zumindest "sehr unübersichtlich" sind. Ein Beispiel ist das Spiel **Türme von Hanoi**:

Auf einem Stab befinden sich *n* in der Mitte mit einem Loch versehene Scheiben, deren Durchmesser von oben nach unten von jeder Scheibe zur nächsten zunimmt. Die Aufgabe besteht darin, die Scheiben auf einen zweiten Stab umzusetzen, wobei in jedem Schritt nur eine Scheibe bewegt werden darf und eine Scheibe nicht auf eine andere mit kleinerem Durchmesser gelegt werden darf. Ein dritter Stab kann als Zwischenspeicher verwendet werden.

Der rekursive Ansatz teilt das Problem in zwei "kleinere" Probleme auf. Die Problemgröße, d.h. der Parameter, der die Rekursion kontrolliert, ist hier die Anzahl der zu bewegenden Scheiben. Zunächst werden die *n*-1 oberen Scheiben vom ersten auf den dritten Stab transportiert, wobei der zweite Stab als Zwischenspeicher fungiert. Dann wird die *n*-te Scheibe (mit dem größten Durchmesser) direkt vom ersten auf den zweiten Stab gelegt. Anschließend erfolgt ein Umsetzen des Scheibenstapels vom dritten auf den zweiten Stab, wobei jetzt der erste Stab als Zwischenspeicher verwendet wird. Die Lösung der beiden kleineren Teilaufgaben erfolgt jeweils nach dem gleichen Prinzip.

Ausgangssituation



Schlußsituation

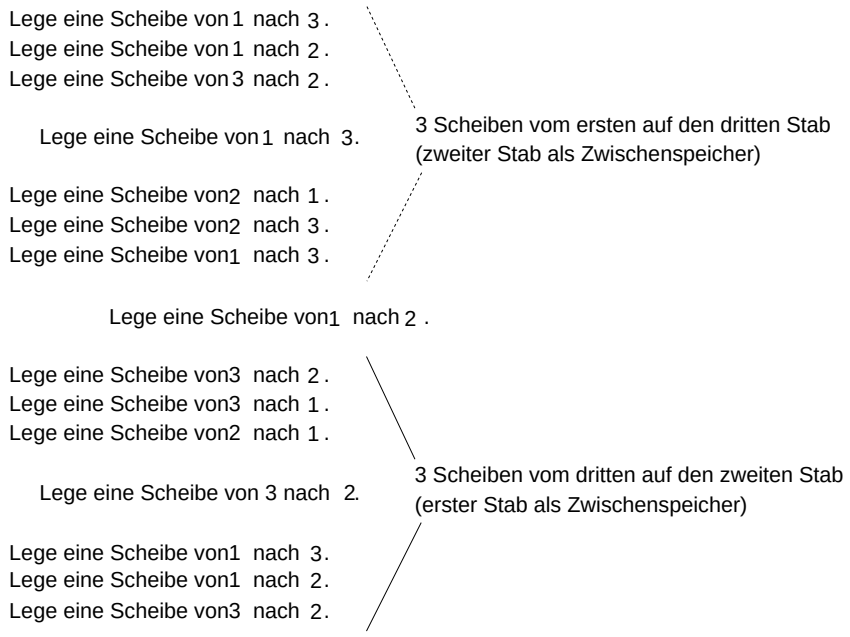
Das Prozedur `move` ermittelt den Handlungsablauf und schreibt ihn mit Hilfe der `Writeln`-Prozedur auf den Bildschirm.

```
...

CONST max = ...;

TYPE disk_typ = 0 .. max;
     stab_typ = 1 .. 3;

PROCEDURE move (anz: disk_tp;
                a  : stab_typ;
                b  : stab_typ;
                c  : stab_typ);
  { move bewegt Scheiben, deren Anzahl in anz angegeben wird,
    vom Stab a zum Stab b, wobei der Stab c als Zwischenspeicher
    verwendet wird                                                }

    BEGIN { move }
      IF anz > 0 THEN BEGIN
                       move (anz - 1, a, c, b);
                       Writeln ('Lege eine Scheibe von ', a,
                                                  ' nach ', b, '.');
                       move (anz - 1, c, b, a)
                      END
    END   { move };

...
```

Das Ergebnis des Programmaufrufs bei Aufruf von

```
move (n, 1, 2, 3);
```

mit $n = 4$ lautet (zur Verdeutlichung der Rekursivität sind die Ausgabezeilen entsprechend eingerückt):



Der Aufwand für die Ermittlung des Handlungsablaufs bei n Scheiben mit Hilfe der Prozedur `move` ist proportional zur Anzahl $T(n)$ der `Writeln`-Aufrufe in `move` (mit erstem Parameter n). Offensichtlich gilt

$T(0) = 0$ und $T(n) = 2 \cdot T(n-1) + 1$ für $n > 0$.

Daraus ergibt sich $T(n) = 2^n - 1$ für $n \geq 0$; es läßt sich zeigen, daß dieser Wert optimal ist.

Es gibt auch eine sehr einfach zu formulierende nicht-rekursive Lösung für die Türme von Hanoi (vgl. [DEW]). Die Korrektheit dieser Lösung ist jedoch nicht so einsichtig wie beim rekursiven Ansatz. Da zu jedem Zeitpunkt die kleinste Scheibe auf einen beliebigen Stab transportiert werden kann, sollte versucht werden, in einem Spielzug möglichst eine andere Scheibe zu transportieren und die kleinste Scheibe nicht direkt zweimal hintereinander zu wählen. Wenn die kleinste Scheibe nicht genommen wird, dann kann nur die zweitkleinste sichtbare (oben auf einem Stapel liegende) Scheibe genommen werden (im folgenden *zweite* Scheibe genannt) und auf einen Stab gelegt werden, der nicht mit der kleinsten Scheibe belegt ist. Man denke sich nun die Stäbe kreisförmig angeordnet. Der Ablauf des Verfahrens wird so eingeschränkt, daß die kleinste Scheibe stets auf den im Uhrzeigersinn nächsten Stab gelegt wird. Dieses Verfahren führt auf den folgenden Algorithmus, dessen Schritte hier nur informell formuliert sind:

```
PROCEDURE hanoi_iterativ;
  BEGIN { hanoi_iterativ }
    REPEAT BEGIN
             transportiere kleinste Scheibe auf den
                   im Uhrzeigersinn nächsten Stab;
             transportiere zweite Scheibe zum verbleibenden Stab;
           END
    UNTIL alle Scheiben sind auf einem Stab;
  END   { hanoi_iterativ };
```

Das Verfahren `hanoi_iterativ` bewegt den Scheibenstapel nicht wie die Prozedur `move` auf einen unter den beiden anderen beliebig wählbaren Stab, sondern auf den im Uhrzeigersinn nächsten oder übernächsten, je nachdem, ob die Anzahl n der Scheiben ungerade oder gerade ist. Auch dazu werden wieder nur $2^n - 1$ Transportbewegungen benötigt.

Nicht immer ist eine rekursive Implementation einer Aufgabenstellung von Vorteil. Als Beispiel dient die Berechnung der **Fibonacci-Folge** $(F_n)_{n \geq 0}$, eine Zahlenfolge, die in vielen Informatikanwendungen von Bedeutung ist: Sie ist definiert durch
$F_0 = 0, F_1 = 1, F_n = F_{n-2} + F_{n-1}$ für $n \geq 2$.

n	0	1	2	3	4	5	6	7	8	9	10	11	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	...

Der rekursive Ansatz zur Berechnung von F_n nutzt die rekursive Definition der Folge:

```
FUNCTION fib_1 (n : INTEGER) : INTEGER;

BEGIN { fib_1 }
  CASE n OF
  0    : fib_1 := 0;
  1    : fib_1 := 1;
  ELSE fib_1 := fib_1 (n-2) + fib_1 (n-1);
  END { CASE };
END   { fib_1 };
```

Eine nicht-rekursive Version beruht auf der expliziten Darstellung von F_n als

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \text{ für } n \geq 0.$$

```
FUNCTION fib_2 (n : INTEGER) : INTEGER;

CONST phi_1 = (1+Sqrt(5))/2;
      phi_2 = (1-Sqrt(5))/2;

VAR x, y : REAL;
    idx  : INTEGER;

BEGIN { fib_2 }
  CASE n OF
  0    : fib_2 := 0;
  1    : fib_2 := 1;
  ELSE BEGIN
         x := phi_1;
         y := phi_2;
         FOR idx := 1 TO n-1 DO
          BEGIN
            x := x * phi_1;
            y := y * phi_2;
          END;
         fib_2 := Trunc( 1/Sqrt(5) * (x - y) );
       END;
  END { CASE };
END  { fib_2 };
```

Ein dritter Ansatz nutzt aus, daß man zur Berechnung von F_n die Folge bei F_0 und F_1 beginnend durchlaufen kann und sich dabei immer nur die letzten beiden Werte der Folge zu merken braucht:

```
FUNCTION fib_3 (n : INTEGER) : INTEGER;

VAR f_n1, f_n2, f_n : INTEGER;
    idx             : INTEGER;

BEGIN { fib_3 }
  CASE n OF
  0    : fib_3 := 0;
  1    : fib_3 := 1;
  ELSE BEGIN
         f_n2 := 0;
         f_n1 := 1;
         FOR idx := n DOWNTO 2 DO
           BEGIN
             f_n  := f_n2 + f_n1;
             f_n2 := f_n1;
             f_n1 := f_n;
           END;
         fib_3 := f_n;
       END;
  END { CASE };
END   { fib_3 };
```

Der algorithmische Aufwand der Lösung `fib_1` ist proportional zur Anzahl der Wertzuweisungen an den Funktionsbezeichner `fib_1`. Bezeichnet $F(n)$ diesen Aufwand zur Berechnung von F_n, so gilt:

$F(0) = F(1) = 1,$

$F(2) = F(1) + F(0) = 2,$

$F(3) = F(2) + F(1) = 3$ usw.

Offensichtlich ist

$F(n) = F_{n-1} = O((1+\sqrt{5})^{n-1}) = O(2^n)$, für $n > 0$.

Es handelt sich um einen exponentiellen Aufwand[27].

Der algorithmische Aufwand der Lösung `fib_2` ist proportional zur Anzahl der Durchläufe der `FOR`-Schleife, d.h. der Aufwand ist von der Ordnung $O(n)$, jedoch sind Rundungsfehler durch den Einsatz der Gleitkommaarithmetik nicht ausgeschlossen.

Der Ansatz `fib_3` berechnet F_n mit einem algorithmischen Aufwand, der proportional zur Anzahl der Durchläufe der dortigen `FOR`-Schleife ist, also ebenfalls mit einem Aufwand der Ordnung $O(n)$. Rundungsprobleme treten hier nicht auf. Da die Zahl F_n selbst von der Größenordnung $O(2^n)$ ist, kann ein geringerer Berechnungsaufwand nicht erwartet werden.

Rekursive Lösungsansätze treten in sehr vielen nicht-numerischen Problemen auf. Ein prominentes Beispiel ist die **Binärsuche**:

Das durch

```
CONST n = ...;
VAR t : ARRAY [1..n] OF INTEGER;
```

deklarierte Feld enthalte aufsteigend sortierte Einträge[28], d.h. es gelte
`t[1]` ≤ ... ≤ `t[n]`.

Die im folgenden beschriebene Prozedur `bin_search` stellt bei Eingabe des Feldes `t` und einer Zahl `a` feststellen, ob `a` unter `t[1]`, ..., `t[n]` vorkommt, und ermittelt gegebenenfalls den Feldindex `i` mit `a = t[i]`. Dazu wird zunächst das mittlere Element `t[middle]` in `t` geprüft (bei einer geraden Anzahl von Elementen ist das mittlere Element das erste Element der zweiten Feldhälfte). Ist es gleich `a`, so ist der gesuchte Feldindex gefunden, und die Suche ist beendet. Andernfalls liegt `a`, wenn es überhaupt im Feld vorkommt, im vorderen Feldabschnitt, falls `a < t[middle]` ist, oder im hinteren Feldabschnitt, falls `a > t[middle]` ist. Die Entscheidung, in welchem Feldabschnitt weiterzusuchen ist, kann jetzt

---

[27] Die hier verwendete ***O-Notation*** bedeutet (vgl. [GKP]):
Eine Funktion $T(n)$ ist von der **Ordnung** $O(f(n))$, geschrieben $T(n) = O(f(n))$, wenn gilt:
Es gibt eine (nicht von n abhängige) Konstante $C > 0$ und eine natürliche Zahl n_0 mit
$|T(n)| \le C|f(n)|$ für jedes $n \ge n_0$.

[28] Eine allgemeinere Voraussetzung ist die Annahme, daß auf den Elementen von `t` eine Ordnungsrelation erklärt ist und die Einträge in `t` gemäß dieser Ordnungsrelation angeordnet sind.

getroffen werden. Gleichzeitig wird durch diese Entscheidung die andere Hälfte aller potentiell auf Übereinstimmung mit `a` zu überprüfenden Feldelement ausgeschlossen. Im Feldabschnitt, der weiter zu überprüfen ist, wird nach dem gleichen Prinzip (also rekursiv) verfahren. Unter Umständen muß die Suche fortgesetzt werden, bis ein noch zu überprüfender Feldabschnitt nur noch ein Feldelement enthält. Die Größe, die hier die Rekursion kontrolliert, ist die Anzahl noch zu überprüfender Feldelemente. In jedem Rekursionsschritt wird diese Kontrollgröße im wesentlichen halbiert.

Eine Implementierung der Binärsuche verwendet

```
CONST  max = ... ;              { maximale Feldgröße          }

TYPE idx_bereich  = 1..max;
     entry        = INTEGER;    { Typ der Feldeinträge        }
     feld_typ     = ARRAY [idx_bereich] OF entry;
```

Die Prozedur `bin_search` wird bei $n \leq$ `max` durch

```
    bin_search (a, t, 1, n, gefunden, index);
```

aufgerufen.

```
PROCEDURE bin_search (    a       : entry;
                          t       : feld_typ;
                          i_min   : idx_bereich;
                          i_max   : idx_bereich;
                      VAR gefunden: BOOLEAN;
                      VAR index   : idx_bereich);

 { gefunden = TRUE, falls a unter t[i_min], ..., t[i_max] vorkommt;
   in diesem Fall ist a = t[index] mit i_min ≤ index ≤ i_max;
   gefunden   = FALSE sonst                                    }

   VAR middle: idx_bereich { Index auf das mittlere Feldelement
                             von t[i_min], ... t[i_max]          };
```

```
BEGIN { bin_search }
  gefunden := FALSE;
  IF i_min < i_max
  THEN BEGIN { der Feldausschnitt t[i_min], ..., t[i_max]
               enthält mindestens 2 Elemente }
         middle := i_min + ((i_max - i_min + 1) DIV 2);
         IF a = t[middle]
         THEN BEGIN
                gefunden := TRUE;
                index    := middle
              END
         ELSE BEGIN
                IF a < t[middle]
                THEN bin_search (a, t, i_min, middle-1,
                                 gefunden, index)
                ELSE bin_search (a, t, middle + 1, i_max,
                                 gefunden, index)
              END
       END
  ELSE BEGIN
         IF a = t[i_min]
         THEN BEGIN
                gefunden := TRUE;
                index    := i_min
              END
       END
END   { bin_search };
```

Zu beachten ist, daß in die Deklaration der Prozedur `bin_search` die die Rekursion kontrollierende Größe nur implizit eingeht, nämlich durch Angabe der Unter- und Obergrenze des noch zu untersuchenden Feldabschnitts.

Das zu durchsuchende Feld `t` enthalte $n \le$ `max` viele Einträge, und zur Darstellung von n benötige man m viele Binärstellen mit führender Stelle 1_2. Dann gilt

$n = 2^{m-1} + l$ mit $0 \le l < 2^{m-1}$ und $m > 0$, d.h. $2^{m-1} \le n < 2^m$.

Der algorithmische Aufwand von `bin_search` ist proportional zur Anzahl $V(n)$ der Vergleiche, in denen das jeweils mittleren Elements im zu untersuchenden Feldabschnitt mit dem "Suchelement" `a` verglichen wird. Denkt man sich den Feldabschnitt `t[1]`, ..., `t[`n`]` um einige Elemente erweitert vor, so daß die entstehende Elementanzahl gleich 2^m ist, so wird die Anzahl durchzuführender Vergleiche sicherlich nicht kleiner (dieses erweiterte Feld enthält ja das ursprüngliche Feld `t[1]`, ..., `t[`n`]`. Es gilt also $V(n) \le V(2^m)$.

Im ersten Aufruf von `bin_search` wird `a` im erweiterten Feld mit 2^m Einträgen mit t$[1 + 2^{m-1}]$ verglichen. Im ungünstigsten Fall ist im vorderen Feldabschnitt mit 2^{m-1} Elementen weiterzusuchen; falls die Suche im hinteren Feldabschnitt weitergeht, bleibt sogar noch ein Element weniger. Für den zweiten Vergleich im rekursiven Aufruf

`bin_search(a, t, 1,` 2^{m-1}`, gefunden, index)`

sind nun noch höchstens 2^{m-1} viele Elemente zu überprüfen. Allgemein halbiert sich im ungünstigsten Fall die Anzahl noch zu überprüfender Feldelemente. Es ergeben sich die Gleichungen

$V(2^m) \le 1 + V(2^{m-1})$ für $m > 0$,

$V(1) = 1$,

deren Lösung $V(2^m) \le m+1$ für $m \ge 0$ lautet. Aus der Voraussetzung $2^{m-1} \le n < 2^m$ folgt nacheinander:

$2^{m-1} < n+1 \le 2^m$,

$m-1 < \log_2(n+1) \le m$ und

$\log_2(n+1) \le m < \log_2(n+1)+1$, d.h.

$m = \lceil \log_2(n+1) \rceil$.

Insgesamt sieht man, daß die Suche nach einem bestimmten Element in einem sortierten Feld mit *n* Elementen mit Binärsuche höchstens

$\lceil \log_2(n+1) \rceil + 1$

viele Elementvergleiche benötigt. Diese Größenordnung ist optimal.

Die Binärsuche ist ein Spezialfall einer allgemeineren Problemlösungsstrategie, die in vielen Basisalgorithmen (auch der systemnahen Programmierung) nutzbringend angewendet werden kann, des **Divide-and-Conquer-Prinzips (Teile-und-Herrsche-Prinzip)**. Hierbei liegt die Verwendung rekursiver Prozeduren auf der Hand. Das Divide-and-Conquer-Prinzip kann wie folgt formuliert werden:

Divide-and-Conquer	
	Gegeben sei ein Problem der Größe *n*. Für kleine Werte von *n* löst man das Problem direkt. Für große Werte von *n* führt man die folgenden Schritte durch:
1.	***Divide:*** Man teilt das Problem der Größe *n* in (wenigstens) zwei kleinere Teilprobleme.
2.	***Conquer:*** Man löst die "kleineren" Teilprobleme nach demselben Prinzip (rekursiv) und erhält für jedes Teilproblem eine Lösung.
3.	***Merge:*** Man fügt die Teillösungen zu einer Gesamtlösung zusammen.

Das Divide-and-Conquer-Prinzip soll an einer der am häufigsten vorkommenden Programmieraufgaben, der Sortierung von Elementen, gezeigt werden. Eine typische Lösung ist der Quicksort (siehe z.B. [O/W]), der jedoch trotz seiner Schnelligkeit in den meisten praktischen Fällen im Rechenaufwand asymptotisch nicht optimal ist. Prinzipiell besser ist der Heapsort in seinen verschiedenen Varianten. Hierbei wird jedoch eine komplexe Datenstruktur verwendet, deren Beschreibung hier zu weit gehen würde.

Eine direkte Umsetzung des Divide-and-Conquer-Prinzip ist der **Mergesort**. Dieses Verfahren zählt nicht zu den internen Sortierverfahren wie z.B. der Quicksort und der Heapsort, die die Sortierung im Feld selbst, das die zu sortierenden Zahlen enthält, durchführen. Bei der Zusammensetzung der Teillösungen im Merge-Schritt wird in einem zusätzlichen Hilfsfeld Speicherplatz (im Heap) bereitgestellt, um Elementverschiebungen im Ausgangsfeld zu vermeiden. Die folgende Implementierung in der Prozedur mergesort verwendet wieder die Deklarationen

```
CONST  max = ... ;              { maximale Feldgröße          }

TYPE idx_bereich  = 1..max;
     entry        = INTEGER;    { Typ der Feldeinträge        }
     feld_typ     = ARRAY [idx_bereich] OF entry;
```

Der Aufruf der Prozedur `mergesort` bei einem Feld `t` mit $n \leq$ `max` Einträgen erfolgt durch

```
mergesort (t, 1, n);
```

```
PROCEDURE mergesort ( VAR t : feld_typ;
                      lower : idx_bereich;
                      upper : idx_bereich);

VAR hilfsfeld_ptr : ^feld_typ;

PROCEDURE sort ( VAR t : feld_typ;
                 lower : idx_bereich;
                 upper : idx_bereich);

  VAR  i      : idx_bereich;
       j      : idx_bereich;
       k      : idx_bereich;
       middle : idx_bereich;

  BEGIN { sort }
    IF lower < upper
    THEN BEGIN
           { Eingabefolge in der Mitte in zwei Teilfolgen teilen }
           middle := lower + ((upper - lower) DIV 2);
           sort (t, lower, middle);
           sort (t, middle+1, upper);

           { Mischen }
           i := lower;
           j := middle + 1;
           k := lower;
           WHILE (i <= middle) AND (j <= upper) DO  { <=== }
           BEGIN
             IF t[i] < t[j]
             THEN BEGIN
                    hilfsfeld_ptr^[k] := t[i];
                    i := i + 1
                  END
             ELSE BEGIN
                    hilfsfeld_ptr^[k] := t[j];
                    j := j + 1
                  END;
             k := k + 1
           END { WHILE };

           { Restfolgen übertragen }
           IF i <= middle
           THEN  WHILE i <= middle DO
                 BEGIN
                   hilffeld_ptr^[k] := t[i];
                   i := i + 1;
                   k := k + 1
                 END
            ELSE WHILE j <= upper DO
                 BEGIN
                   hilfsfeld_ptr^[k] := t[j];
```

```
                    j := j + 1;
                    k := k + 1
                  END;

              { sortierte Folge zurückschreiben }
              FOR k := lower TO upper DO
                t[k] := hilfsfeld_ptr^[k]

            END { IF }

      END { sort };

    BEGIN { mergesort }
      { Hilfsfeld initialisieren }
      New (hilffeld_ptr);

      { Sortieren }
      sort (t, lower, upper);

      { Hilfsfeld freigeben }
      Dispose (hilfsfeld_ptr)

    END   { mergesort };
```

Die die Rekursionen des Mergesorts kontrollierende Größe ist die Anzahl der Elemente im zu sortierenden Feldabschnitt. Es fällt auf, daß `mergesort` Vergleiche der Feldelemente nur im Merge-Schritt ausführt.

Es bezeichne $M(n)$ die Anzahl der Rechenschritte, die `mergesort` bei Eingabe eines Felds mit n Elementen benötigt. Es gelte wieder

$n = 2^{m-1} + l, 2^{m-1} \le n < 2^m$

mit geeigneten Werten m und l. Man kann sogar annehmen, daß das Feld um einige Elemente erweitert wird, so daß es nun 2^m viele Elemente enthält. Die zusätzlichen Elemente haben alle den (fiktiven) Wert ∞. Dann gilt $M(n) \le M(2^m)$.

Für $M(2^m)$ gelten folgende rekursive Gleichungen:

$M(2^m) = c \cdot 2^m + 2 \cdot M(2^{m-1})$ für $m > 0$,

$M(1) = c$

mit einer Proportionalitätskonstanten c>0. Die Lösung lautet

$M(2^m) = c \cdot (m+1)2^m, m \ge 0,$

und insgesamt folgt wegen $m = \lceil \log_2(n+1) \rceil$:

$M(n) = O(n \log n).$

Dies ist asymptotisch optimal.

Zusammenfassend kann man feststellen, daß der rekursive Ansatz häufig zu einer eleganten und übersichtlichen Lösung eines Problems führt, andererseits aber in der Implementierung meist speicherplatzaufwendiger als eine nicht-rekursive Lösung ist. Für jede nicht-abgeschlossene Rekursion müssen ja noch Informationen wie Rücksprungadressen, lokale Variablen usw. vermerkt werden.

7.5 Das Prinzip der Implementierung

Das Thema dieses Kapitels ist die Beschreibung einer ***exemplarischen*** Realisation der Unterprogrammverknüpfung bei Programmiersprachen, in denen rekursive Prozeduraufrufe zugelassen sind. Im konkreten Fall erzeugt der Compiler entsprechenden Code; auf keinen Fall muß sich der Anwender in die Details der Unterprogrammverknüpfung einschalten. Gelegentlich werden die Mechanismen auch hinter Routinen des Laufzeitsystems der Sprache verborgen. Das hier beschriebenen Prinzip entspricht weitgehend der in der INTEL 80x86-Architektur verwendeten Methode.

Auch hier soll wieder eine Beschränkung auf die Behandlung "reiner" Prozeduren (in Pascal `PROCEDURE`) erfolgen, da Funktionen (in Pascal `FUNCTION`) als Prozeduren mit einem weiteren impliziten Rückgabeparameter, nämlich dem Funktionsergebnis, angesehen werden können. Außerdem werden zunächst Formalparameter vom Prozedurtyp ausgeschlossen; am Ende dieses Kapitels werden sie kurz betrachtet.

Für jeden Prozeduraufruf wird ein Aktivierungsrecord (siehe Kapitel 7.2) ***im Stack angelegt***, der im wesentlichen die Rücksprungadresse zum Aufrufer, die lokalen Variablen (einschließlich der Formalparameter) der Prozedur und Adreßverweise enthält, über die auf aus Sicht der Prozedur globalen Datenobjekte zugegriffen werden kann. ***Der Aktivierungsrecord wird teilweise vom Aufrufer und teilweise von der Prozedur selbst erzeugt und am Ende des Prozeduraufrufs von der Prozedur wieder entfernt. Die Größe eines Aktivierungsrecords hängt von der gerufenen Prozedur ab.***

Der Wert des Stackpointers SP identifiztiert den "obersten" Stackeintrag. Diese Identifizierung soll in Form einer Adresse erfolgen. Im folgenden bewegt sich der belegte Teil des Stacks durch eine PUSH-Operation in Richtung niedrigerer Adressen und bei der POP-Operation in Richtung höherer Adressen, d.h. PUSH dekrementiert den Wert von SP, während POP ihn inkrementiert.

Ein Aktivierungsrecord, der komplett erzeugt und im Stack abgelegt ist, besteht selbst aus mehreren Komponenten, deren Anfänge jeweils einen relativen Abstand (Offset) zu der Adresse haben, die in SP steht. Um zu verdeutlichen, daß der Zugriff auf diese Komponenten eine lokale Operation auf der Ebene der jeweiligen Prozedur bzw. des Aufrufers ist, während die Manipulationen des SP-Werts durch die Operationen PUSH und POP global an den Stack gebunden sind, wird ein weiteres globales Datenobjekt verwendet, das **Basispointer (BP)** genannt werden soll. BP enthält eine Adresse, die in den Stack verweist, und zwar im wesentlichen die Adresse, die den Aktivierungsrecord in die beiden Teile teilt, die vom Aufrufer einer Prozedur bzw. von der Prozedur selbst generiert werden. Alle Einträge im Aktivierungsrecord unterhalb des BP-Werts (in Richtung größerer Adressen relativ zum BP-Wert) werden vom Aufrufer erzeugt. Alle Einträge oberhalb des BP-Werts (in Richtung absteigender Adressen relativ zum BP-Wert) werden von der gerufenen Prozedur angelegt (Abbildung 7.5-1).

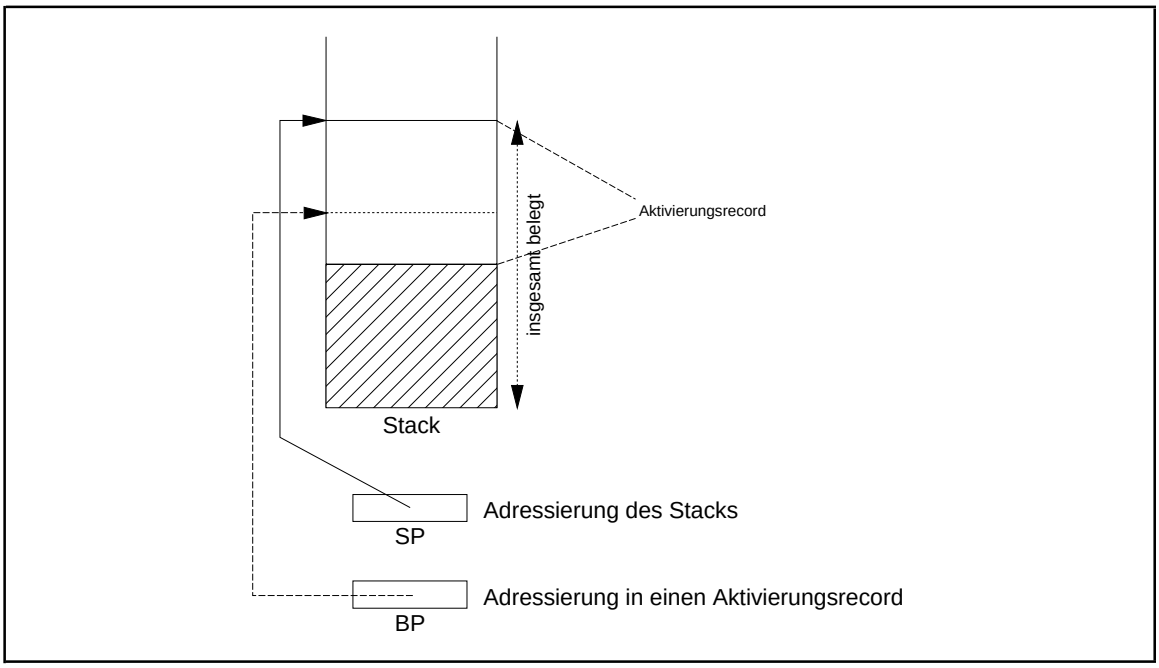


Abbildung 7.5-1: Aktivierungsrecord im Stack

Das Implementierungsprinzip soll zunächst am Layout eines Aktivierungsrecords für einen Aufruf einer Prozedur verdeutlicht werden, die nur auf ihre eigenen lokalen Variablen zugreift, d.h. auf die Variablen, die innerhalb der Prozedur oder als Formalparameter deklariert sind. Die Prozedur

```
PROCEDURE proc (f1 : ...; ...; fn : ...;);
  { lokale Variablen von proc: }
  VAR v1 : ...;
      ...
      vm : ...;
  BEGIN { proc }
    ...
  END   { proc };
```

werde durch

```
proc (a1, ..., an);
```

aufgerufen. Die Aktualparameter a_1, ..., a_n gehören zum Aufrufer. Die Details, die sich aus den Datentypen der einzelnen Datenobjekte und der Parameterübergabemethode ergeben, sollen zunächst nicht betrachtet werden.

Der Compiler hat für den Prozeduraufruf Code erzeugt, der (mittels PUSH) nacheinander die Werte bzw. Adressen der Aktualparameter a_1, ..., a_n (je nach Parameterübergabemethode) und die Rücksprungadresse auf den Stack transportiert. Der Speicherplatz für diese Werte bzw. Adressen der Aktualparameter auf dem Stack ist identisch mit dem Speicherplatz für die Formalparameter f_1, ..., f_n der gerufenen Prozedur. Jetzt erfolgt eine Verzweigung in den Code von `proc` (Abbildung 7.5-2 (a)).

Der Code, der für das `BEGIN`-Statement in der Prozedur `proc` generiert wurde, bewirkt, daß der gegenwärtige BP-Wert auf dem Stack abgelegt wird und der BP mit dem jetzigen SP-Wert überschrieben wird. Dadurch wird eine Rückwärtsverkettung in den Aktivierungsrecord des Aufrufers hergestellt, und zwar an die Stelle, die ihrerseits eine Rückwärtsverkettung zu einem vorherigen BP-Wert enthält. Der neue BP-Wert ist der Anfang dieser Rückwärtsverkettung. Anschließend wird der SP-Wert soweit dekrementiert, wie Platz für die lokalen Variablen v_1, ..., v_m der Prozedur `proc` benötigt wird, d.h. der Speicherplatz für v_1, ..., v_m ist nun zugewiesen (Abbildung 7.5-2 (b)).

Am Ende von `proc` bewirkt der für das `END`-Statement vom Compiler eingesetzte Code, daß v_1, ..., v_m vom Stack wieder entfernt werden (Operation POP), der BP mit der Rückwärtsverkettung wieder restauriert wird und ein Rücksprung an die Adresse erfolgt, die nun als oberster Eintrag auf dem Stack steht. Vorher werden noch die Rücksprungadresse und a_1, ..., a_n bzw. f_1, ..., f_n vom Stack genommen (Abbildung 7.5-2 (c)).

Offensichtlich kann innerhalb der Prozedur ein Zugriff auf ihre lokalen Variablen v_1, ..., v_m und f_1, ..., f_n ***relativ zum BP-Wert erfolgen***: v_1, ..., v_m haben einen negativen, f_1, ..., f_n einen positiven Offset zu BP. Dabei ist es unerheblich, wo genau innerhalb des Stacks der Aktivierungsrecord dieses Prozeduraufrufs liegt oder in welcher Rekursionstiefe der Prozeduraufruf erfolgte.

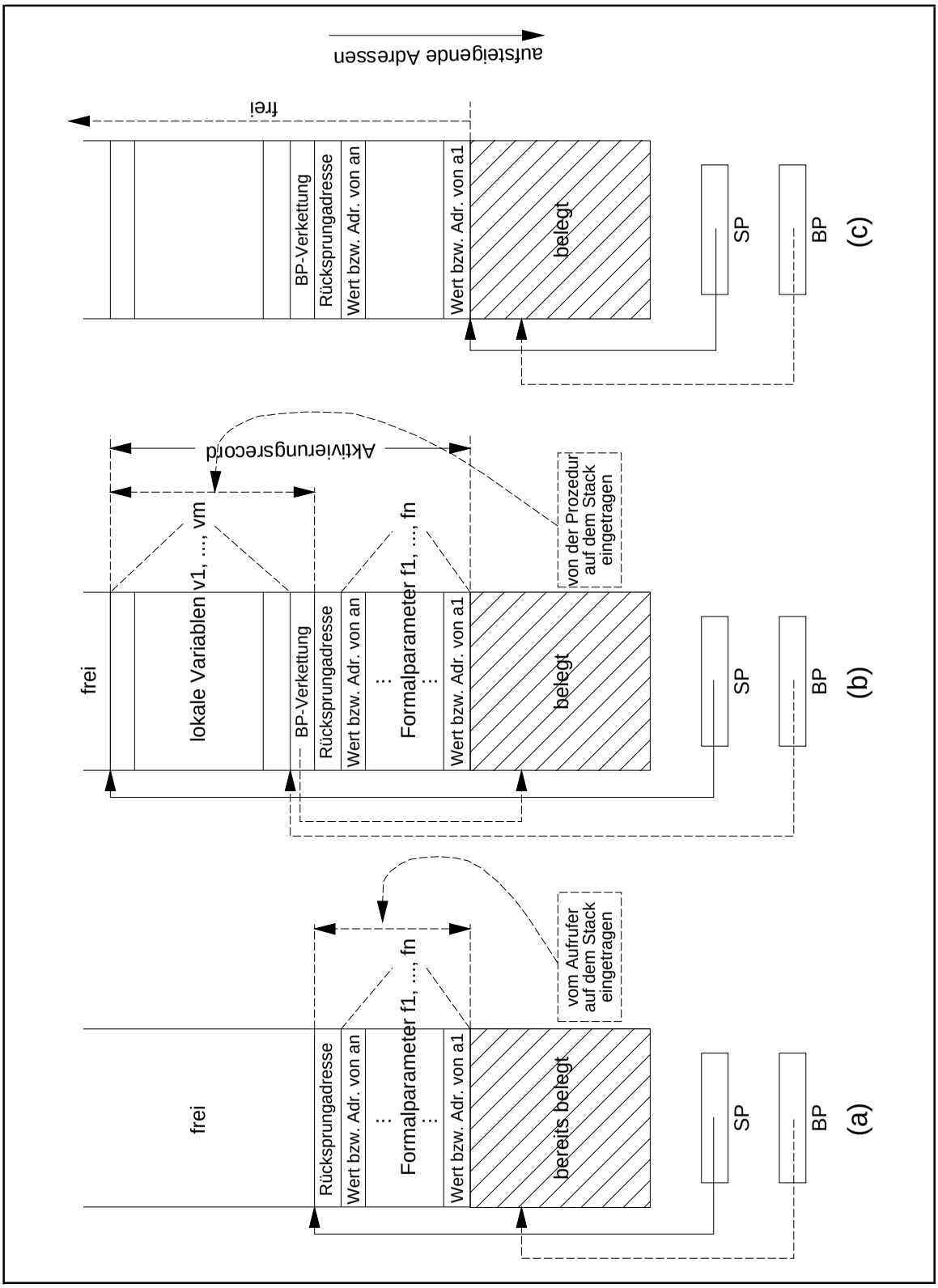


Abbildung 7.5-2: Stackbelegung bei einem Unterprogrammsprung (keine globalen Variablen)

Um die Auswirkung von Datentypen und Parameterübergabemethoden auf das Layout des Aktivierungsrecords deutlich zu machen, werden obige Deklarationen etwas präzisiert und spezialisiert. Die Prozedur `proc` habe nun zwei Formalparameter, nämlich einen call-by-reference-Parameter `f1` und einen call-by-value-Parameter `f2`, und eine lokale Variable `v1` mit den im folgenden angegebenen Datentypen:

```
TYPE feld_typ = ARRAY[1..10] OF INTEGER;

PROCEDURE proc (VAR f1 : feld_typ;
                    f2 : INTEGER);
  { lokale Variablen von proc: }
  VAR v1 : RECORD
             belegt : BOOLEAN;
             wert   : INTEGER;
           END;
      v2 : INTEGER;

  BEGIN { proc }
                                { <== Abbildung 7.5-3 (a) }
    ...
    v1.belegt := TRUE;
    v1.wert   := f2;
    ...
    f1[3] := v1.wert + 2;
    f2    := f1[3]+3;
                                { <== Abbildung 7.5-3 (b) }
    ...
  END   { proc };
                                { <== Abbildung 7.5-3 (c) }
```

Sie werde durch

```
proc (a1, a2);
```

aufgerufen. Dabei seien die Variablen `a1` und `a2` lokal zum Aufrufer, der selbst eine Prozedur ist, so daß diese Variablen innerhalb dessen Aktivierungsrecords liegen. Die Variable `a2` habe zum Aufrufzeitpunkt den Wert 42_{10} = $\texttt{002A}_{16}$. Für die interne Darstellung eines Datenobjekts werden die in Kapitel 4 beschriebenen Darstellungen genommen. Ein Datenobjekt vom Typ `INTEGER` belegt dabei ein Wort mit 2 Bytes. Für eine Rücksprungadresse zum Aufrufer werden hier 4 Bytes genommen (entsprechend einem in der INTEL 80x86-Architektur üblichen FAR-Pointer). Genauso werden Adressen, die Datenobjekte im globalen Datenbereich lokalisieren, mit 4 Bytes festgehalten. Für Adressen, die sich auf den Stack beziehen, also auch für die Werte von BP und SP, genügen 2 Bytes (NEAR-Pointer); ein Datenobjekt vom Typ `BOOLEAN` benötigt intern ein Byte. Eine Ausrichtung soll so stattfinden, daß im Stack ein Datenobjekt immer auf Wortgrenze beginnt. In einem `INTEGER`-Datenobjekt wird intern das niedrigwertige Byte an der kleineren Adresse gespeichert (little-ending, siehe Kapitel 4.1). Das bedeutet, daß in der bildlichen Darstellung, bei der aufsteigende Adressen von links nach rechts und oben nach unten wiedergegeben werden, ein Datenobjekt vom Datentyp `INTEGER` und (binärem) Wert $[b_{15}\ldots b_8 b_7\ldots b_0]_2$ als $b_7\ldots b_0 b_{15}\ldots b_8$ erscheint.

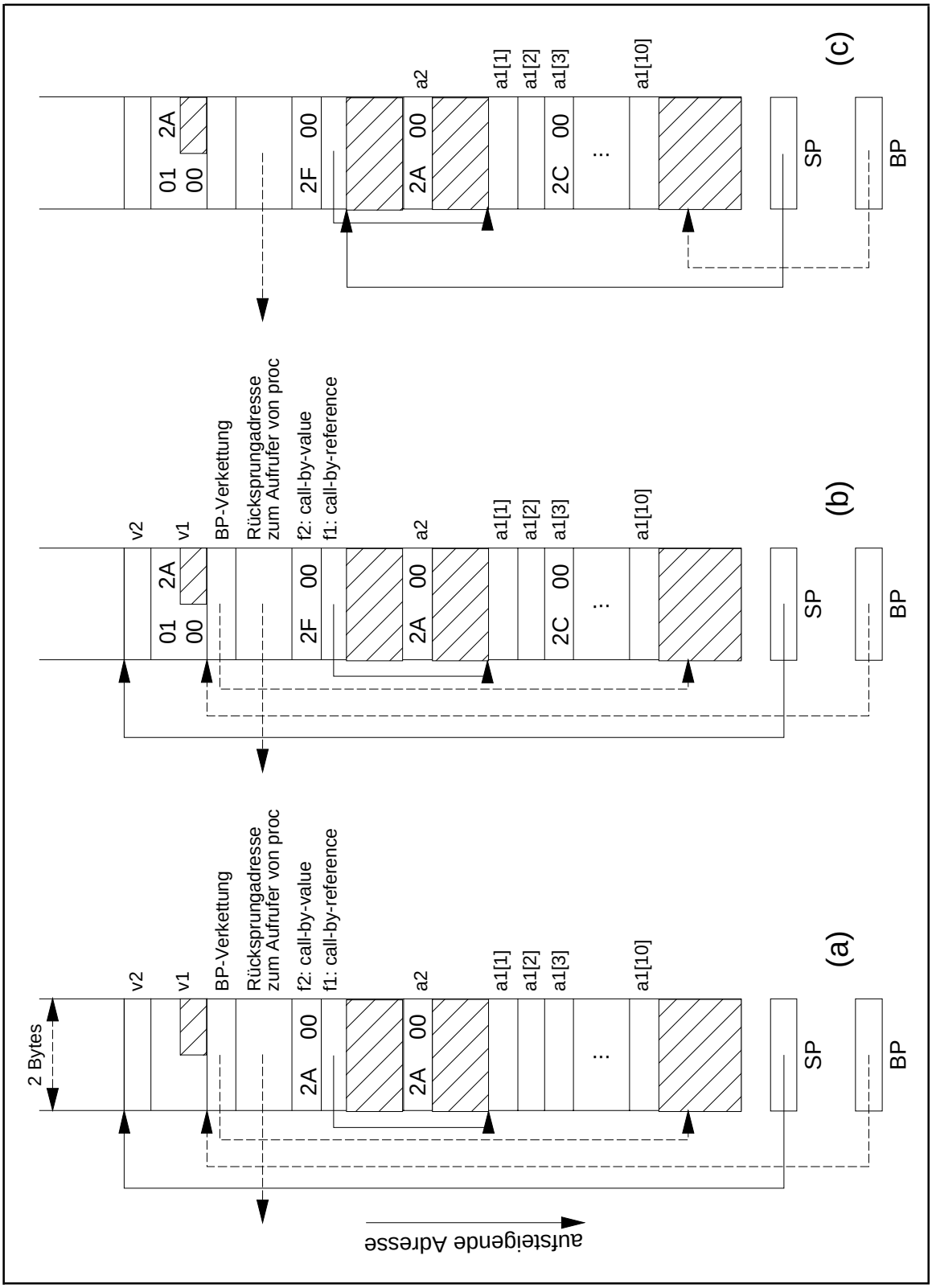


Abbildung 7.5-3: Stackbelegung bei einem Unterprogrammsprung (Beispiel)

Die Stackbelegung unmittelbar nach Abarbeitung des `BEGIN`-Statements in `proc` zeigt Abbildung 7.5-3 (a). Die Teile (b) bzw. (c) zeigen die Situation unmittelbar nach den Wertzuweisungen bzw. unmittelbar vor dem Sprung aus `proc` zurück zum Aufrufer.

Die lokale Variable `v1` beginnt mit ihrer Komponente belegt am Offset -4 relativ zum aktuellen BP-Wert; die Komponente `wert` hat einen Offset von -3. Durch die Vereinbarung der Ausrichtung aller Datenobjekte auf Wortgrenze innerhalb eines Aktivierungsrecords wird 1 Byte "verschenkt". Der Speicherplatz für die lokale Variable `v2` wird nach der Speicherplatzreservierung für `v1` festgelegt, so daß `v2` oberhalb von `v1` mit dem Offset -6 relativ zum BP-Wert auf dem Stack liegt. Der Formalparameter `f2` hat den Offset +6 relativ zum BP-Wert. Die Adresse des zu `f1` korrespondierenden Aktualparameters ist in `f1` am Offset +8 relativ zum BP-Wert eingetragen.

Um die Umsetzung der Zuweisungsstatements in `proc` in Maschinencode näher zu beschreiben, wird die Pascal-ähnliche Notation

```
v^ := arithmetischer_ausdruck
```

verwendet. Sie soll ausdrücken, daß der Wert von `v` als Adresse interpretiert und der Wert des Datenobjekts an dieser Adresse durch den Wert des arithmetischen Ausdrucks ersetzt wird (die Anzahl benötigter Bytes ergibt sich implizit aus dem Datentyp des arithmetischen Ausdrucks). Anstelle des arithmetischen Ausdruck kann auch eine syntaktische Konstruktion der Form

```
Datentyp(adr^)
```

stehen, die bedeutet, daß dem Datenobjekt an der durch `adr` bestimmten Adresse der angegebene Datentyp aufgeprägt wird. Außerdem soll eine einfache Adreßarithmetik zugelassen sein: der Wert eines Datenobjekts, das eine Adresse enthält, darf um einen konstanten Wert verändert werden. Beispielsweise bedeutet `INTEGER((adr-2)^)` das Datenobjekt vom Typ `INTEGER`, das 2 Bytes vor der Adresse beginnt, die in `adr` steht. Die Anweisung `(adr+4)^ := INTEGER((adr-2)^)` weist den beiden Bytes an der Adresse, die 4 Bytes nach dem Wert von `adr` beginnen, das Datenobjekt vom Typ `INTEGER` zu, das 2 Bytes vor der Adresse in `adr` beginnt.

Die Wirkung des für die Wertzuweisung
```
v1.belegt := TRUE;
```

generierten Maschinencodes kann man durch

```
(BP-4)^ := 01
```

umschreiben (`TRUE` wird in die Konstante 01 übersetzt). Die folgende Tabelle stellt den Anweisungen des Pascalcodes die generierten Maschineninstruktionen in der obigen Notation gegenüber. Man sieht, daß die Adressierungen aller lokalen Datenobjekte von `proc` relativ zu BP erfolgen können.

Pascal-Anweisung	Wirkung der zugehörigen Maschineninstruktionen	Bemerkung
`v1.belegt := TRUE;`	`(BP-4)^ := 01`	
`v1.wert := f2;`	`(BP-3)^ := INTEGER((BP+6)^)`	`f2` ist ein call-by-value-Parameter, enthält also den Wert des Aktualparameters
`f1[3] := v1.wert + 2;`	`((BP+8)^+4)^ := INTEGER((BP-3)^)+2`	`f1` ist ein call-by-reference-Parameter, enthält also die Adresse des Aktualparameters; das Feldelement mit dem Index 3 hat einen Abstand 4 = (3-1)*2 vom Feldanfang
`f2 := f1[3] + 3;`	`(BP+6)^ := INTEGER(((BP+8)^+4)^)+3`	nur der Wert der lokalen Kopie des Aktualparameters wird geändert

Der Aktivierungsrecord eines Prozeduraufrufs wird etwas komplexer, wenn eine Prozedur auf globale Datenobjekte zugreifen kann, die in Prozeduren liegen, die die aufgerufene Prozedur einschachteln (auf Variablen des globalen Datenbereichs des Hauptprogramms besteht grundsätzlich Zugriffsberechtigung, falls diese innerhalb der Prozedur überhaupt benannt werden können). ***Die aus Sicht einer Prozedur globalen Variablen sind in der Menge der globalen und lokalen Variablen des sie umschließenden Blocks enthalten. Diese globalen Variablen befinden sich also im statischen Datenbereich oder in Aktivierungsrecords*** (zu umschließenden Prozeduraufrufen)***, die zum Aufrufzeitpunkt der Prozedur bereits im Stack liegen.*** Dort haben sie jeweils einen festen Offset zum Anfang des statischen Datenbereichs bzw. relativ zum BP-Wert des zugehörigen Aktivierungsrecords.

Als Beispiel wird wieder das Programm in Abbildung 6.1-1 betrachtet. Dabei soll jetzt zusätzlich angenommen werden, daß `p_program` selbst eine Prozedur und nicht ein Hauptprogramm ist, so daß die lokalen Variablen von `p_program` im Stack und nicht im globalen Datenbereich verwaltet werden. Die statische Blockstruktur der Prozeduren kann man dann auf unterschiedliche Weise darstellen, z.B. als ineinandergeschachtelte Blöcke oder als gerichteten Graph, dessen Kantenrichtung die mögliche Aufrufreihenfolge anzeigt (Abbildung 7.5-4). Eine gerichtete Kante dieses Graphen in vertikaler Richtung beschreibt dabei einen möglichen Aufruf einer eingeschachtelten Prozedur, in horizontaler Richtung einen Aufruf auf derselben Schachtelungstiefe. Der untere Teil von Abbildung 7.5-4 zeigt die Stackbelegung, wenn aus `p_program` heraus `a_proc` und von dort `b_proc` aufgerufen werden (diese Aufrufreihenfolge vereinfacht etwas die in Abbildung 6.1-1 angegebene Aufrufreihenfolge, in der ein Aufruf von `a_proc` aus `d_proc` heraus erfolgt; für den Augenblick soll diese etwas vereinfachte Aufrufreihenfolge angenommen werden). In Abbildung 7.5-4 sind exemplarisch Variablen wiedergegeben, auf die beim Aufruf von `b_proc` zugegriffen werden kann: die Variable `w` (exakter: das mit `w` bezeichnete Datenobjekt d_7) ist lokal in `b_proc`, also im Aktivierungsrecord, der über den gegenwärtigen BP-Wert angesprochen wird; die Variable `b` ist global für `b_proc` und lokal für `a_proc`; sie liegt im Aktivierungsrecord des `a_proc`-Aufrufs. Entsprechend befindet sich die Variable `v` im Aktivierungsrecord des `p_program`-Aufrufs.

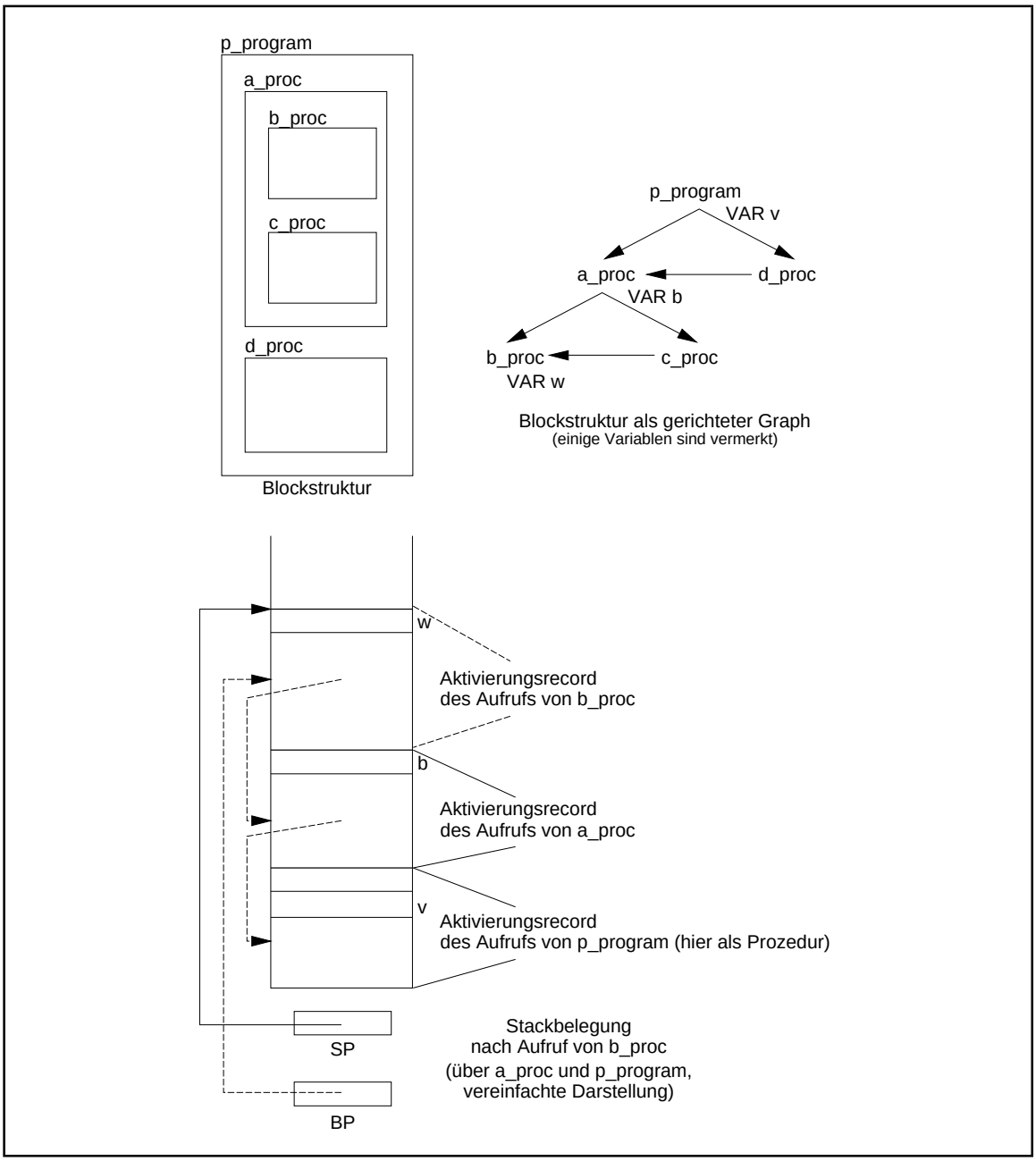


Abbildung 7.5-4: Blockstruktur (Beispiel aus Abbildung 6.1-1)

Für jede Variable ist zur Übersetzungszeit aufgrund der Regeln über die Gültigkeitsbereich von Bezeichnern klar, über wieviele Stufen in der Schachtelungstiefe der Blockstruktur nach außen gegangen werden muß, um sie zu erreichen: 0 Stufen für lokale Variablen, *d* Stufen, falls die Variable im *d*-ten umfassenden Block liegt; hierbei wird von innen nach außen gezählt. Beispielsweise befindet sich für `b_proc` die Variable `w` 0 Stufen, die Variable `b` 1 Stufe und die Variable `v` 2 Stufen weiter außen in der Blockstruktur. Für `a_proc` liegt die Variable `b` 0 Stufen und die Variable `v` nur 1 Stufe weiter außen. Als offensichtliche Regel gilt: Liegt ein Datenobjekt für eine Prozedur innerhalb der Blockstruktur $d \geq 0$ Stufen weiter außen, so liegt es für eine direkt eingebettete Prozedur $d + 1$ Stufen weiter außen, falls überhaupt eine Zugriffsmöglichkeit auf das Datenobjekt besteht.

Es sei noch einmal hervorgehoben, daß die im folgenden beschriebene Implementierung nur eine Möglichkeit aufzeigt, um das Konzept lokaler und globaler Variablen umzusetzen (beispielsweise handhabt der vom Borland Pascal Compiler erzeugte Code den Zugriff auf die nichtlokale Umgebung einer Prozedur auf eine etwas andere Weise).

Für eine während der Laufzeit effiziente Handhabung des Zugriffs auf Datenobjekte in umfassenden Blöcken generiert der Compiler Code, der bei Eintritt in eine Prozedur (BEGIN-Statement) ausgeführt wird und ***der im Aktivierungsrecord der Prozedur zusätzliche Adreßverweise auf diejenigen Aktivierungsrecords erzeugt, die aus Sicht der gerufenen Prozedur globale Datenobjekte in anderen Aktivierungsrecords enthalten.*** Die Menge dieser Adreßverweise heißt **Display**. Alle Display-Informationen liegen im Aktivierungsrecord zwischen der BP-Rückwärtsverkettung und den lokalen Variablen. Wenn sich mehrere globale Datenobjekte in demselben umfassenden Block befinden, wird nur eine einzige Display-Information für die Datenobjekte dieses Blocks festgehalten. Es wird jeweils ein Verweis auf die BP-Rückwärtsverkettung innerhalb eines Aktivierungsrecords als jeweilige Display-Information genommen. Die Display-Informationen einer gerufenen Prozedur können während der Laufzeit bei Eintritt in die Prozedur aus der Display-Information des Aufrufers und der statischen Blockstruktur gewonnen werden. Da alle Blöcke eines Programms Zugriff auf den globalen Datenbereich des Programms besitzen, ist ein Verweis auf diesen Bereich als Teil des Displays überflüssig. Den so **erweiterten Aufbau eines Aktivierungsrecords** für einen Prozeduraufruf zeigt Abbildung 7.5-5.

Zur dynamischen Erzeugung der Display-Information während der Laufzeit wird jedem Prozeduraufruf als zusätzlicher verdeckter Parameter der BP-Wert mitgegeben, der die Lage desjenigen Aktivierungsrecords im Stack bestimmt, der zum nächsten umfassenden Prozeduraufruf (also in der Blockstruktur um eine Stufe weiter nach außen) gehört; er wird im folgenden als *direkt übergeordneter Aktivierungsrecord* bezeichnet. Dieser BP-Wert wird, falls auf dortige lokale Variablen zugegriffen wird, oberhalb der BP-Verkettung im Aktivierungsrecord der gerufenen Prozedur abgelegt (Position 1 in Abbildung 7.5-5). Falls zusätzlich die lokalen Variablen in der nächsten Stufe nach außen innerhalb der Blockstruktur verwendet werden, erfolgt ein weiterer Display-Eintrag (Position 2 in Abbildung 7.5-5). Dieser kann aus dem direkt übergeordneten Aktivierungsrecord genommen werden: er liegt dort an der in Abbildung 7.5-5 mit Position 1 bezeichneten Stelle. Entsprechend werden eventuell weitere Display-Einträge aus dem direkt übergeordneten Aktivierungsrecord erzeugt.

Das gleiche Verfahren wird auch verwendet, wenn die gerufene Prozedur ihrerseits eine weitere Prozedur aufruft, die auf die nichtlokale Umgebung der gerufenen Prozedur zugreift.

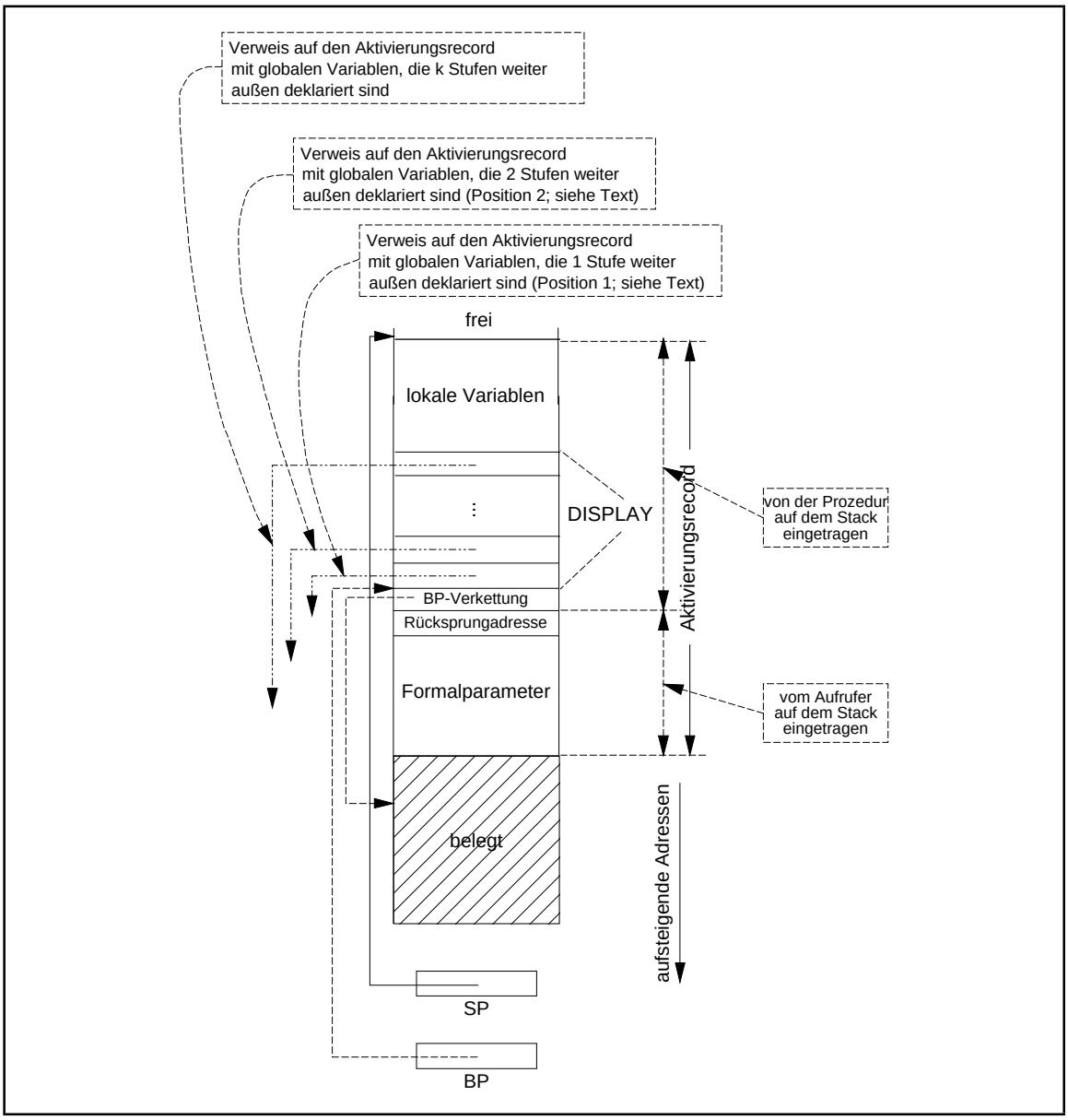


Abbildung 7.5-5: Aktivierungsrecord

Abbildung 7.5-6 zeigt die Stackbelegung, einschließlich der Display-Informationen, nach Eintritt in die Prozedur `b_proc` für das Beispiel der Abbildung 6.1-1. Auch hier wird wieder das Hauptprogramm als Prozedur behandelt. Es wird jetzt wieder die ursprüngliche, etwas komplexere Aufrufreihenfolge `p_program - d_proc - a_proc - b_proc` (vgl. Abbildungen 6.1-1 und 6.2-1) angenommen. In den folgenden Abbildungen werden Einzeleinträge innerhalb des Stacks wieder als gleichgroße Kästchen gezeichnet, ohne die interne Speicherdarstellung der einzelnen Datentypen zu berücksichtigen.

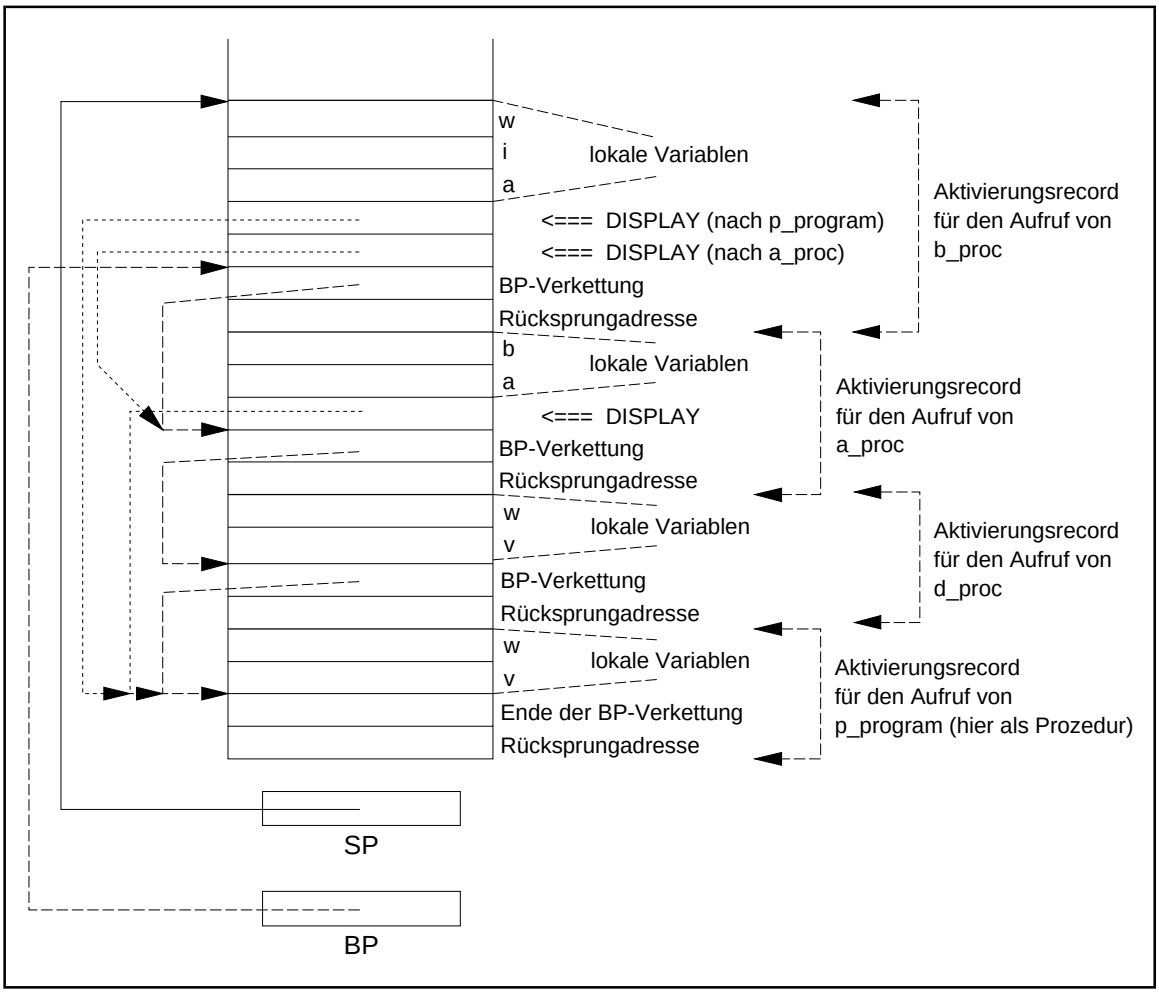


Abbildung 7.5-6: Beispiel der Stackbelegung mit Display-Information

Da alle lokalen Variablendeklarationen in `d_proc` die gleichlautenden Variablendeklarationen von `p_program` überlagern, wird für `d_proc` keine Display-Information festgehalten. Das Statement

```
v := w;
```

im Prozedurrumpf von `d_proc` bezieht sich auf die lokalen Variablen von `d_proc`. Hingegen sind die Variablen der gleichlautenden Anweisung im Prozedurrumpf von `a_proc` global zu `a_proc`, so daß im Aktivierungsrecord des `a_proc`-Aufrufs eine Display-Information auf `p_program` benötigt wird. In der oben eingeführten Maschinensprachen-nahen Notation lautet die Übersetzung dieser Anweisung in `d_proc` (die Längen aller Einträge seien hier mit 2 Bytes angenommen):

```
(BP-2)^ := INTEGER((BP-4)^)
```

Derselbe Code würde für das mögliche gleichlautende Statement in `p_program` erzeugt. Dagegen wird für die Übersetzung der gleichlautenden Anweisung im Prozedurrumpf von `a_proc` ein indirekter Zugriff über die Display-Information verwendet:

```
((BP-2)^-2)^ := INTEGER(((BP-2)^-4)^)
```

Befindet sich das Statement in `b_proc`, so wird es dort in

```
((BP-4)^-2)^ := INTEGER(((BP-4)^-4)^)
```

umgesetzt. Die relativen Abstände der Variablen `v` und `w` zum BP-Wert ist im Aktivierungsrecord von `p_program` jeweils gleich (-2 für `v` bzw. -4 für `w`). Die Prozedur `b_proc` benötigt sowohl eine Display-Information für `a_proc` als auch für `p_program`, da die Variable `v` in `p_program` (und nicht etwa in `d_proc`!) und die Variable `b` in `a_proc` liegen.

Das folgende Beispiel verwendet globale Variablen, von denen eine als call-by-reference-Parameter an eine rekursive Prozedur übergeben wird. Der die Rekursion kontrollierende Parameter ist ein call-by-value-Parameter mit dem Wert 1, so daß auch wirklich ein rekursiver Prozeduraufruf erfolgt. Für den call-by-reference-Parameter wird die in Kapitel 7.1 angeführte Besonderheit bei der Übergabe von call-by-reference-Parametern als Aktualparameter für untergeordnete Prozeduraufrufe berücksichtigt.

Abbildung 7.5-7 zeigt wieder die Stackbelegung zu verschiedenen Augenblicken im Ablauf des Programms, dieses Mal unmittelbar *vor* einem Prozeduraufruf. Die Werte von `INTEGER`-Datenobjekten erscheinen als Dezimalzahlen.

```
PROGRAM h_prog;
  ...
  VAR g : INTEGER;
      h : INTEGER;

  PROCEDURE proc;
    VAR p : INTEGER;

    PROCEDURE a_proc (    v : INTEGER;
                            { call-by-value }
                      VAR r : INTEGER
                            { call-by-reference });
      VAR l : INTEGER;

      BEGIN
        l := v + 5;
        r := l + v + g;
                  { g ist global, im Hauptprogramm }
        p := r;   { p ist global, in proc }
                        { <=== Abbildung 7.5-7 (b) }
                        { <=== Abbildung 7.5-7 (c) }
        IF v > 0
        THEN a_proc (v-1, r);  { rekursiver Prozeduraufruf }
        { Rücksprungadresse a_proc : }
      END { a_proc };

    BEGIN { proc }
      ...
      g := 2;
      h := 10;
                          { <=== Abbildung 7.5-7 (a) }
      a_proc (1, h);
                { nicht-rekursiver Prozeduraufruf }
```

```
    { Rücksprungadresse zu proc : }
    ...
  END   { proc };

BEGIN { h_prog }
  ...
  proc;
  { Rücksprungadresse zu h_prog : }
  ...
END   { h_prog }.
```

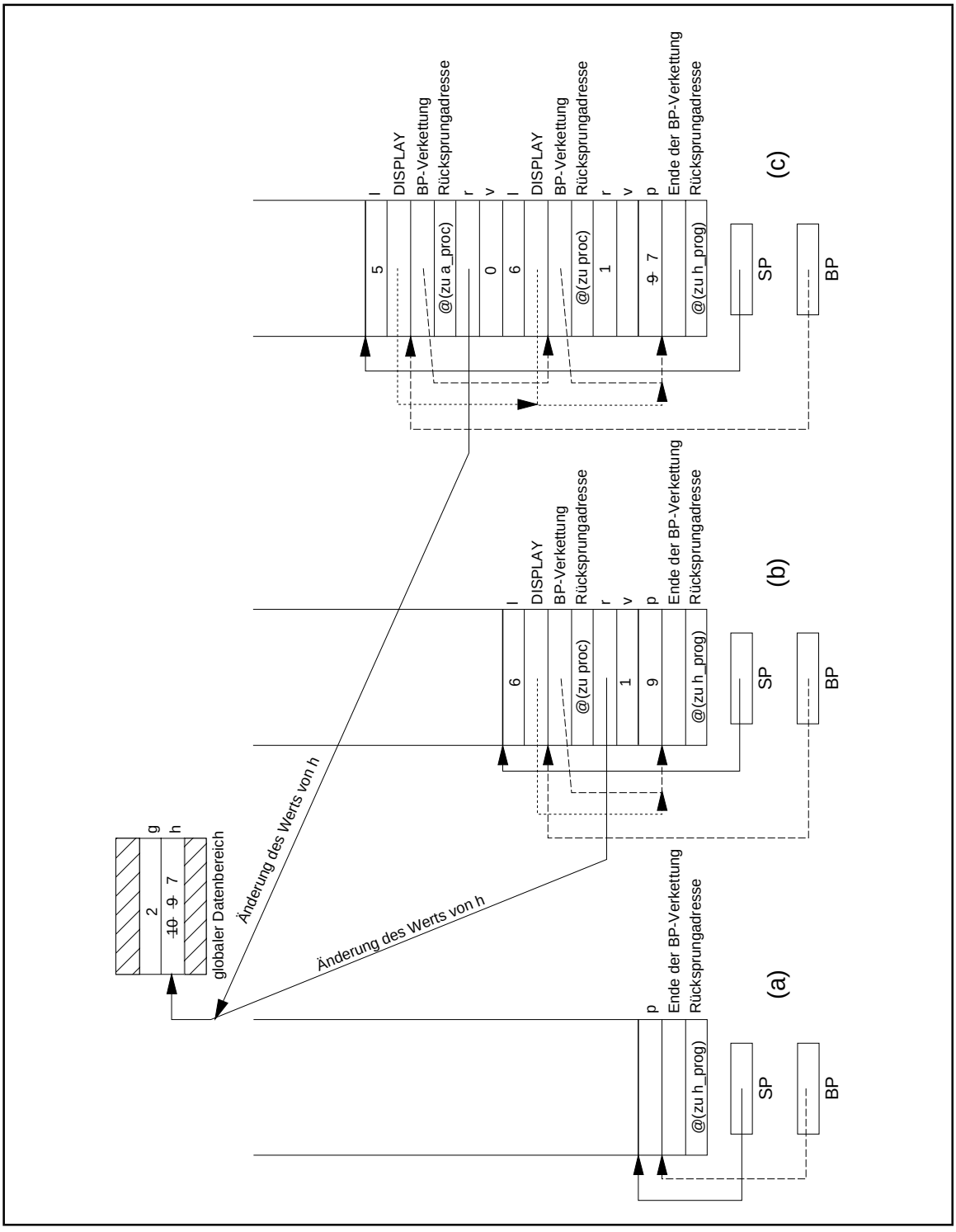


Abbildung 7.5-7: Stackbelegung beim Aufruf einer rekursiven Prozedur (Beispiel)

Die Behandlung von Formal- bzw. Aktualparametern vom Prozedurtyp ordnet sich vollständig in das Konzept ein. Dabei ist zu beachten, daß der Wert eines Aktualparameters vom Prozedurtyp die *Adresse* (innerhalb des Codeteils) der benannten Prozedur ist. Für das folgende Beispiel aus [G/J] wird die Stackbelegung zu unterschiedlichen Ablaufzeitpunkten in Abbildung 7.5-8 gezeigt (Zahlenangaben sind wieder als Dezimalzahlen zu lesen). Zu

sehen ist eine Prozedur `main`, die neben einer parameterlosen Prozedur `a` eine Prozedur `b` enthält, die einen Formalparameter vom Prozedurtyp besitzt. Aus `main` heraus wird zunächst `b` mit Aktualparameter `a` und innerhalb des Prozedurrumpfs von `b` sie selbst rekursiv mit Aktualparameter `c` aufgerufen. Der Bezeichner `y` benennt entweder eine lokale Variable in `b` oder in `a`. Innerhalb des Blocks der Prozedur `c` ist `y` ein globaler Bezeichner von `b`. Zu beachten ist, welche durch `y` bzw. `u` bezeichneten Datenobjekte jeweils verwendet wurden.

```
PROCEDURE main;
  VAR u, v : INTEGER;

  PROCEDURE a;
    VAR y : INTEGER;
    BEGIN { a }
      ...
      y := u + 1;
                  { <=== Abbildung 7.5-8 (b) }
      ...
    END   { a };

  PROCEDURE b (x : PROCEDURE);
    VAR u, v, y : INTEGER;

    PROCEDURE c;
       BEGIN { c };
         ...
         y := u + 1;
                   { <=== Abbildung 7.5-8 (c) }
         ...
       END   { c };

    BEGIN { b }
      u := 1;
      y := 5;
                  { <=== Abbildung 7.5-8 (a), beim ersten Mal }
      x;
      ...
      b(c);
      ...
    END   { b };

  BEGIN { main }
    ...
    u := 10;
    b(a);
    ...
  END   { main };
```

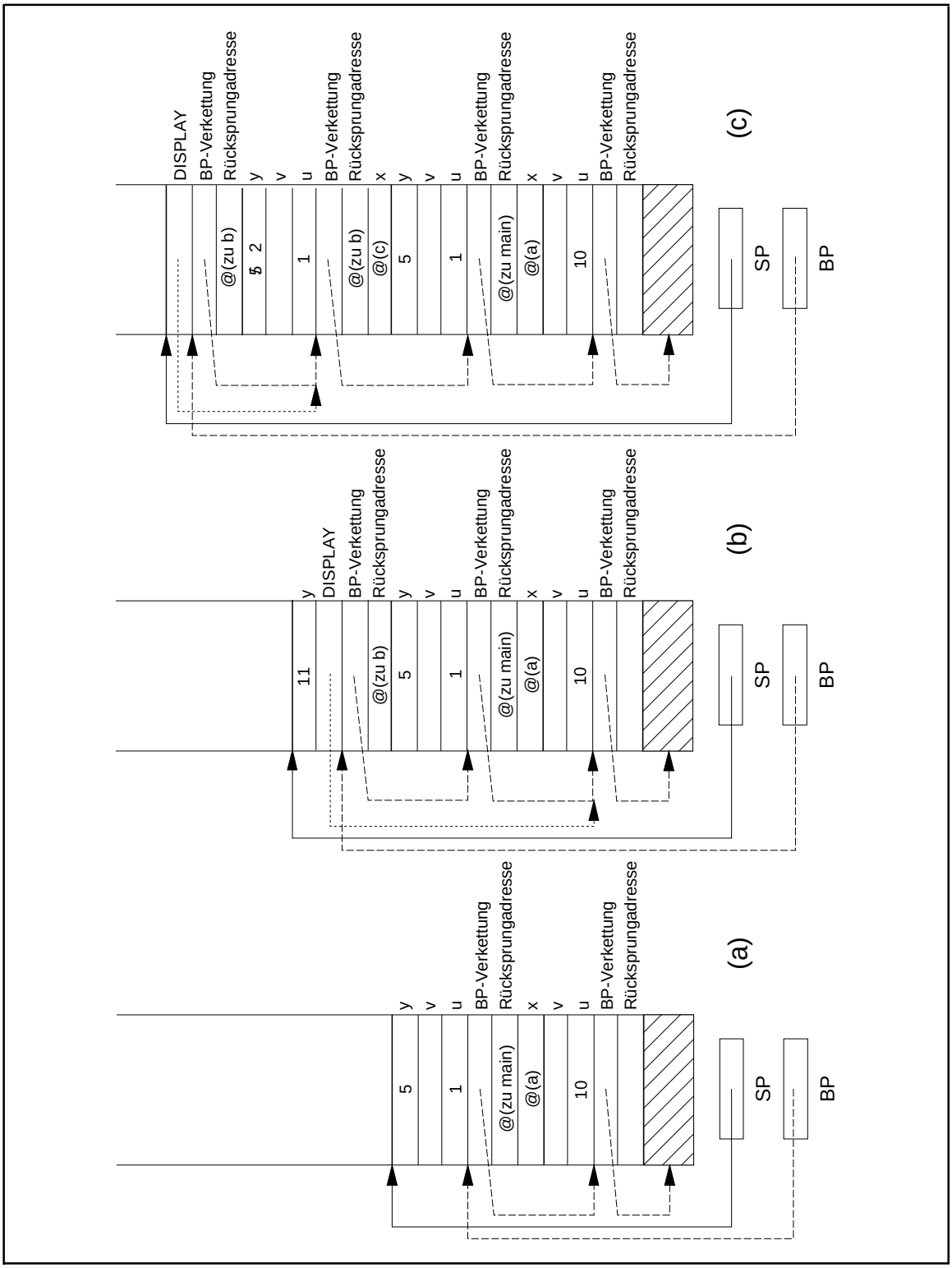


Abbildung 7.5-8: Stackbelegung mit Aufrufparametern vom Prozedurtyp (Beispiel)

7.6 Mehrfachverwendbarkeit von Prozeduren

In dem in Kapitel 3.2 beschriebenen Prozeßkonzept kann eine Prozedur logisch in unterschiedlichen virtuellen Adreßräumen liegen, und die zugehörigen Prozesse laufen parallel durch die Prozedur, ohne daß es zu einer gegenseitigen Beeinflussung kommt. Dafür sorgt ja gerade die Kontrolle durch das Betriebssystem. Das beschriebene Prozedurkonzept gewährleistet darüber hinaus, daß bei Einhaltung der im folgenden beschriebenen Programmierregeln auch unterschiedliche Threads innerhalb desselben Prozesses parallel durch den Code derselben Prozedur laufen können. Hierbei befindet sich die Prozedur nur ein einziges Mal im virtuellen Adreßraum des Prozesses, zu dem die Threads gehören. Mit Parallelität ist hier gemeint, daß ein Thread in den Code einer Prozedur gelaufen ist und ein weiterer Thread den Code der Prozedur betritt, bevor der erste Thread die Prozedur wieder verlassen hat. Es muß sichergestellt sein, daß auch hier keine (ungewollte) gegenseitige Beeinflussung der Threads entsteht.

Unter folgenden Voraussetzung ist es unterschiedlichen Threads erlaubt, eine im virtuellen Adreßraum des Prozesses nur ein einziges Mal vorhandene Prozedur parallel zu durchlaufen:

- ***Jeder Thread verfügt*** neben seinem virtuellen Registersatz ***über einen eigenen Stack*** (für die Einrichtung eines "privaten" Stacks für jeden Thread sorgt das Betriebssystem); als Stack bei der Umsetzung des Prozedurkonzepts wird immer der private Stack des gerade aktiven Threads genommen

- ***Die Prozedur verwendet*** keine globalen Daten, sondern ***ausschließlich lokale Datenobjekte*** (Formalparameter, innerhalb der Prozedur deklarierte lokale Variablen)

Code, der von mehreren Threads (auch unterschiedlicher Prozesse) parallel durchlaufen werden kann, bezeichnet man als **parallel mehrfach-benutzbar (ablauf-invariant, reentrant)**.

Die Forderung, daß parallel mehrfach-benutzbarer Code keine globalen Datenobjekte verwendet, liegt auf der Hand: Setzt ein Thread den Wert einer globalen Variablen, so kann er nicht sicher sein, daß diese Variable beim nächsten Zugriff den von ihm gesetzten Wert noch enthält. Inzwischen kann ja ein Threadwechsel stattgefunden haben, und ein anderer Thread hat den Variablenwert verändert.

Der vom Compiler erzeugte Code, der bei Eintritt in die Prozedur durchlaufen wird, generiert einen Aktivierungsrecord mit den lokalen Datenobjekten der Prozedur auf dem Stack des aktiven Threads. Der Zugriff auf alle diese Datenobjekte erfolgt relativ zum Basispointer BP. Betritt ein weiterer Thread mit seinem eigenen Stack die Prozedur, wird ein weiterer Aktivierungsrecord nun im Stack dieses Threads erzeugt. Analog der Erzeugung einer weiteren Inkarnation lokaler Variablen bei jedem rekursiven Prozeduraufruf, erhält jeder Thread daher seinen eigenen Satz lokaler Variablen (Abbildung 7.6-1). Natürlich muß das Betriebssystem bei einem Threadwechsel neben den jeweiligen virtuellen Registersätzen auch die privaten Stacks der Threads umschalten.

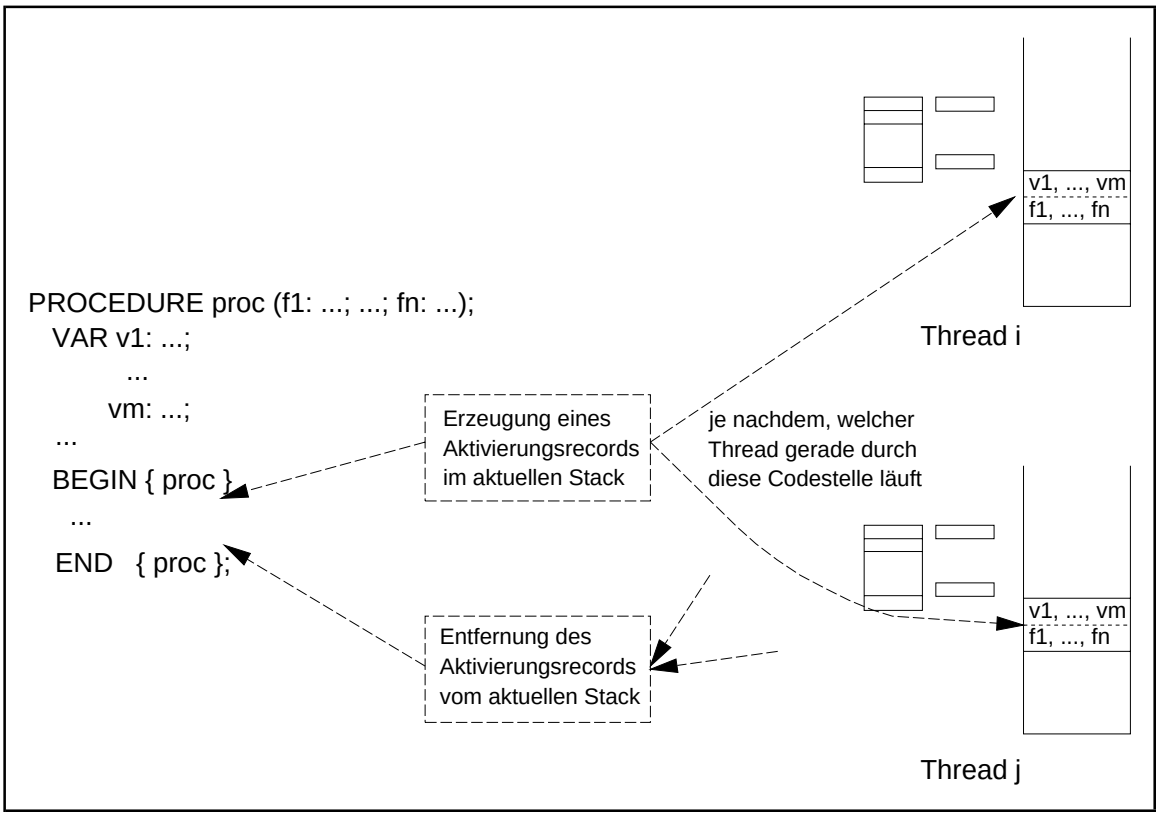
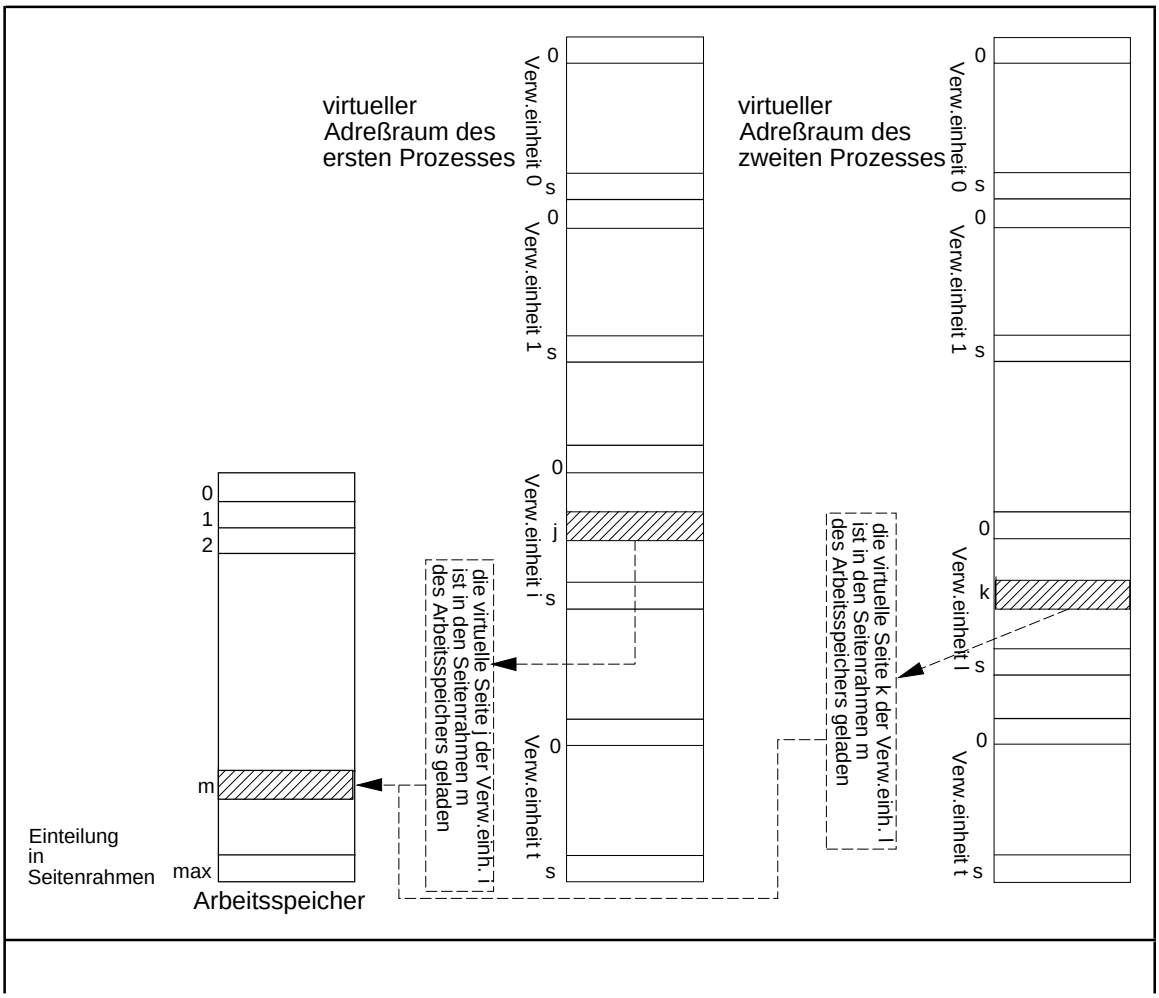


Abbildung 7.6-1: Parallel mehrfach-benutzbarer Code

Prinzipiell ist es unerheblich, ob die Threads, die durch den parallel mehrfach-benutzbaren Code einer Prozedur laufen, demselben oder unterschiedlichen Prozessen angehören. Es ist daher dem Betriebssystem möglich, einen derartigen Code physisch auch nur ein einziges Mal in den Arbeitsspeicher zu laden, obwohl er logisch gleichzeitig eventuell in unterschiedlichen virtuellen Adreßräumen und dort an unterschiedlichen virtuellen Adressen liegt. Man spricht hier von **Code-Sharing**. Um den Code nur einmal im Arbeitsspeicher zu halten und zugreifbar von unterschiedlichen Prozessen zu haben, werden spezielle Funktionen des Betriebssystems benötigt. Es muß beispielsweise verhindern, daß der von mehreren Prozessen benutzte Code aus dem Arbeitsspeicher des Rechners ausgelagert wird, solange noch ein Prozeß auf ihn zugreift, auch wenn ein Prozeß, der den Code mit anderen zusammen benutzt hat, beendet wird.

Je nach eingesetzter virtueller Speicheradressierung und Adreßumsetzung wird der Zugriff auf derartigen Code (in der Speicherverwaltung des Betriebssystems) realisiert. Bei Verwendung virtueller Speichertechnik mit Paging (Kapitel 3.3.3) ist die vom Betriebssystem verwaltete Einheit des virtuellen Adreßraums eine Seite. Eine Prozedur belegt eine oder mehrere Seiten in den virtuellen Adreßräumen der am Code-Sharing beteiligten Prozesse, wobei sie dort durchaus jeweils in unterschiedlichen Seiten liegen kann. ***Es geht beim Code-Sharing also darum, einzelne Seiten mit identischem Inhalt verschiedener virtueller Adreßräume auf dieselben physikalischen Seitenrahmen im Arbeitsspeicher abzubilden.***

Abbildung 7.6-3 zeigt das Prinzip, nach dem für zwei unterschiedliche Prozesse Seiten ihres jeweiligen virtuellen Adreßraums (mit ein und demselben Inhalt) demselben Seitenrahmen im Arbeitsspeicher zugeordnet werden kann. Die in Frage kommende Seite befindet sich in den Prozessen auf unterschiedlichen virtuellen Adressen: im ersten Prozeß trägt die Seite die Seitennummer *j* in der Verwaltungseinheit *i*, im anderen Prozeß die Seitennummer *k* in der Verwaltungseinheit *l*. Ist die Seite in den Arbeitsspeicher abgebildet, so enthält der Eintrag für Seite *j* in der Seitentabelle für Verwaltungseinheit *i* des ersten Prozesses den gleichen Verweis auf den Arbeitsspeicher wie der Eintrag für Seite *k* in der Seitentabelle für Verwaltungseinheit *l* des zweiten Prozesses.



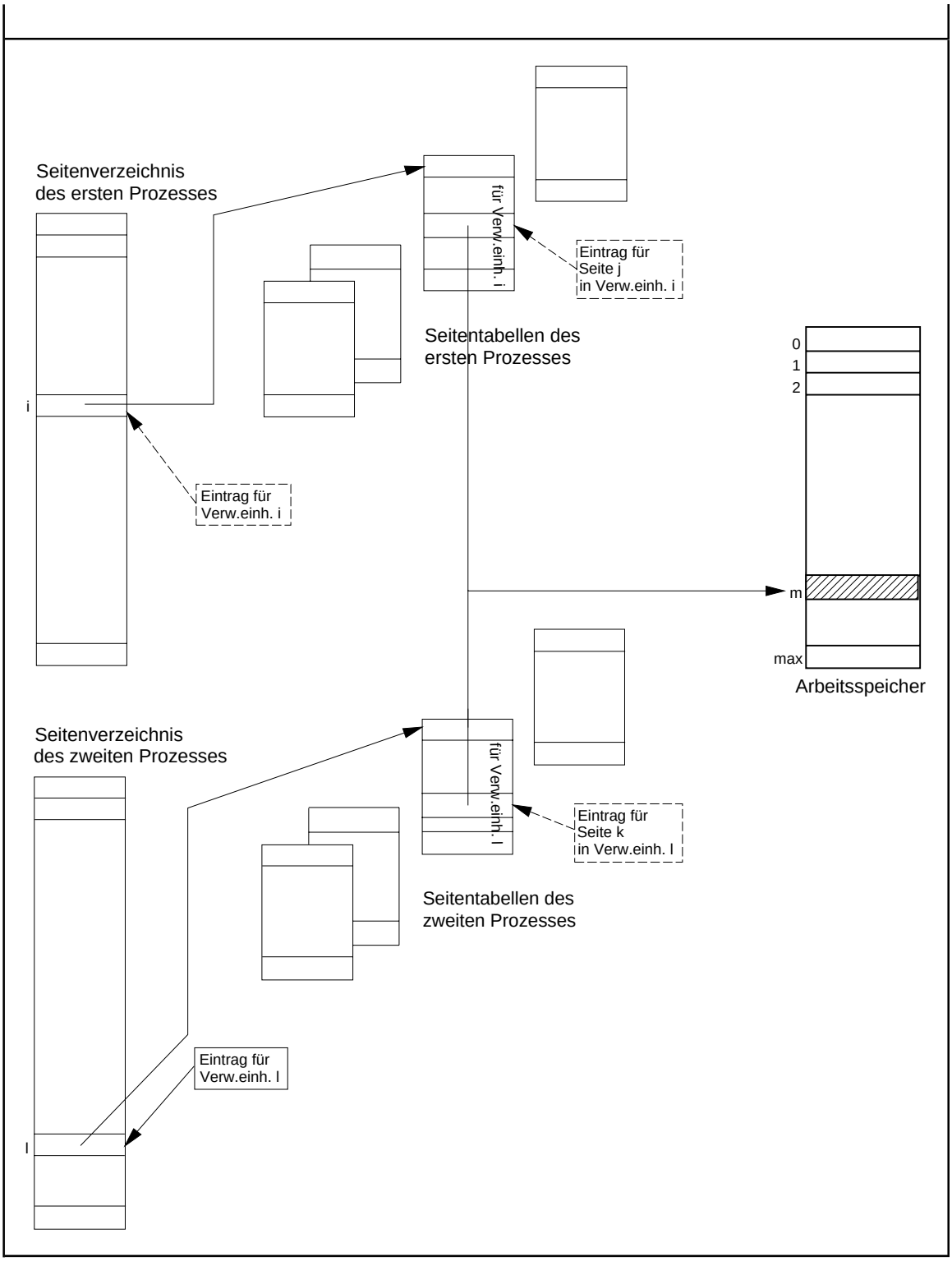


Abbildung 7.6-3: Sharing auf Seitenebene beim Paging

Bei der Verwendung einer Speichertechnik mit Segmentierung, wie sie in Kapitel 3.3.2 beschrieben wird, bei der die vom Betriebssystem verwaltete Grundeinheit ein ganzes Codesegment ist, wird ein ähnliches Verfahren eingesetzt. Jedes im Arbeitsspeicher geladene Segment wird ja durch einen Deskriptor beschrieben, der sich in der lokalen Deskriptortabelle

eines Prozesses (die allein nur von diesem Prozeß benutzt wird) oder in der globalen Deskriptortabelle (für alle Segmente, die von allen Prozessen gemeinsam benutzt werden) befindet. Ein von mehreren Prozessen benutzbares und im Arbeitsspeicher geladenes Segment erfordert einen Eintrag in der globalen Deskriptortabelle.

8 Modularisierung

Das Prozedurkonzept einer höheren Programmiersprache unterstützt die Aufteilung einer (komplexen) Programmieraufgabe in einzelne Teilaufgaben. Die Einbettung einzelner Prozeduren in andere Prozeduren ermöglicht eine Kapselung rein lokaler Teilaufgaben nach außen und eine strenge Top-Down-Modularisierung einer größeren Programmieraufgabe. Im bisher beschriebenen Konzept sind alle Prozeduren **interne Unterprogramme**, da sie innerhalb eines Hauptprogramms deklariert, d.h. im Quellcode des Hauptprogramms angegeben werden. In Standard-Pascal ist keine andere Modularisierungsmöglichkeit vorgesehen. Programmiersprachen wie C unterstützen noch nicht einmal ein Top-Down-Vorgehen, da C keine eingebetteten Funktionen[29] zuläßt. Die Anforderungen, die Software-Engineering-Konzepte an eine Programmiersprache stellen (Top-down-Design, objektorientiertes Vorgehen mit Vererbung und Kapselung von Datenobjekten, Wiederverwendbarkeit von Code, Programmiersprachenunabhängigkeit von Modulen auf Objektcode-Ebene, automatische Versionskontrolle usw.), verlangen weitgehende Modularisierungsmöglichkeiten.

8.1 Automatisches Einfügen von Quellcode

Quellcodeteile, die in mehreren Programmieraufgaben identisch vorkommen, müssen nur einmal erstellt werden und können dann, nachdem sie in einer eigenen Datei abgelegt wurden, ***vor der Übersetzung automatisch in Quellcodes verschiedener Programme hineinkopiert werden***. Beispielsweise könnten mehrere Programme eine Reihe identischer Deklarationen verwenden; einzelne Prozeduren/Funktionen könnten identisch in mehreren Programmen vorkommen; größere durch `BEGIN` und `END` geklammerte Codeteile könnten in mehreren Programmen verwendet werden.

In PASCAL fügt beispielsweise der in einem Quelltext vorkommende Compiler-Befehl (das ist eine Steueranweisung an das Übersetzungsprogramm, eine definierte Funktion auszuführen)

```
{$I dateiname}
```

automatisch vor der Übersetzung den Inhalt der mit `dateiname` benannten Datei genau an dieser Stelle ein. Diese Einfügung ist im Quelltext nicht zu sehen, aber im übersetzten Programm ist der entsprechende Code enthalten. Durch weitere Steuerparameter kann bestimmt werden, in welchem Verzeichnis der Übersetzer nach der benannten Datei suchen soll.

[29] Prozeduren im Sinne des `PROCEDURE`-Konzepts in Pascal gibt es nicht.

8.2 Aufruf externer Prozeduren

Gelegentlich ist es notwendig, Programmteile (Prozeduren) aufzurufen, die spezielle Hardware-Eigenschaften der Maschine berücksichtigen und nur mit Maschinensprachbefehlen realisierbar sind. Dazu gehören z.B. Basisroutinen zum Prozeßwechsel in einer Multitasking-Umgebung oder einige Interruptroutinen.

In PASCAL erfolgt der Anschluß dieser (in Maschinensprache geschriebener) **externen Unterprogramme**, die vorher mit Hilfe des Assemblers in INTEL-Object-Format übersetzt und in einem benannten Modul (.OBJ-Datei) abgelegt wurden, mit der **EXTERNAL**-Anweisung. Im Quellcode wird lediglich das Aufrufformat (Prozedurkopf) angegeben zusammen mit dem Schlüsselwort EXTERNAL und dem Compiler-Anweisung (`{$L ... }`), die mitteilt, in welcher externen Objektdatei die anzubindenden Prozeduren zu finden sind (Details findet man in [BP7]).

Das folgende Beispiel ermöglicht den nachfolgenden Aufruf externer Prozeduren, deren Objektcode sich in der Datei mit Namen BLKSTUFF.OBJ befindet:

```
PROCEDURE MoveWord (VAR Quelle, Ziel; Zahl : LongInt); EXTERNAL;
PROCEDURE MoveLong (VAR Quelle, Ziel; Zahl : LongInt); EXTERNAL;

PROCEDURE FillW (VAR Ziel; Data: INTEGER; Z : LongInt); EXTERNAL;
PROCEDURE FillL (VAR Ziel; Data: LongInt; Z : LongInt); EXTERNAL;

{$L BLKSTUFF.OBJ}
```

8.3 Laufzeitbibliotheken

Eine Reihe von Prozeduren/Funktionen, die in vielen Programmen vorkommen, deren Implementierung häufig aber Rechner-abhängig ist, wird in Form einer **Laufzeitbibliothek** bereitgestellt. Die Laufzeitbibliothek ist Teil des Entwicklungssystems der Programmiersprache. Im Quelltext eines Programms können die Prozeduren der Laufzeitbibliothek verwendet werden, ohne sie explizit zu deklarieren. Die Handbücher der Programmiersprache geben über Bezeichner, Aufrufformate, Bedeutung der Parameter und Randbedingungen Auskunft. Bei der Übersetzung wird der benötigte Code automatisch angebunden. Programmiersprachen wie C oder C++ haben standardisierte Laufzeitbibliotheken, die auf allen Rechnersystemen verfügbar sein müssen.

Die Inhalte der Laufzeitbibliothek realisieren zum einen standardisierte Sprachelemente, z.B. mathematische Funktionen und Ein/Ausgabefunktionen, zum anderen stellen sie funktionale Spracherweiterungen, z.B. Schnittstellen zur Bildschirmsteuerung und eine Reihe "nützlicher" Rechner-spezifischer Konstanten und Variablen bereit, die auf speziellen Betriebssystem- oder Rechnerfunktionen aufbauen.

8.4 Das Unit-Konzept in PASCAL

Die Aufteilung eines Programms in getrennt übersetzbare Programmeinheiten (Moduln) und das anschließende Binden zu einem ablauffähigen Programm bietet eine Reihe offensichtlicher ***Vorteile***. Diese werden insbesondere dann sichtbar, wenn einige Programmteile in unterschiedlichen Programmiersprachen formuliert werden *müssen*, weil beispielsweise Anschlüsse an Betriebssystemdienste eine Maschinensprachenprogrammierung erfordern oder in einem Unterprogramm spezielle Hardwareeinrichtungen angesteuert werden sollen. Bei der Aufteilung einer größeren Programmieraufgabe in Teilaufgaben bietet es sich an, jede dieser Teilaufgaben in eine für die Teilaufgabe spezifische Menge getrennt übersetzter Moduln zu legen. Alle Moduln sind dann meist in derselben Programmiersprache geschrieben.

Ein ***Nachteil*** des Einsatzes getrennt übersetzter Moduln besteht (neben der eventuellen Komplexität des gegenseitigen Programmanschlusses) darin, daß es dem Compiler nicht möglich ist, die Datentypen der aktuellen Parameter in einem Aufruf eines getrennt übersetzten Unterprogramms daraufhin zu überprüfen, ob sie mit den Datentypen der korrespondierenden Formalparameter übereinstimmen. An dieser Stelle ist strenge Programmierdisziplin vom Anwender gefordert; sonst kann es zu Laufzeitfehlern oder logischen Programmfehlern kommen.

Um die Vorteile der Datentypüberprüfung mit den Vorteilen getrennt übersetzter Programmteile auf Sprachebene zu verbinden, sind zusätzliche Konzepte erforderlich. In PASCAL ist dazu das **Unit-Konzept** realisiert; Java enthält ein ähnliches Modularisierungskonzept. Diese Sprachkonzepte genügen zusätzlich der Forderung des **Information Hidings**, des Verbergens implementierungstechnischer Details einzelner externer Unterprogramme gegenüber dem Anwender.

Eine **Unit** besteht (auf Sprachebene) aus drei Teilen:

- dem **Interfaceteil** als "öffentlichen" Teil der Unit; er enthält die Deklarationen von Datenobjekten (Datentyp- und Variablendeklarationen), Prozedur- und Funktionsköpfen, die von außen zugreifbar sind

- dem **Implementierungsteil** als "privaten" Teil der Unit; er enthält den Programmcode zu den im Interfaceteil aufgeführten Prozedur- und Funktionsköpfen und eventuell zusätzliche nach außen nicht verfügbare Daten-, Prozedur- und Funktionsdeklarationen; auf keines der nur im Implementierungsteil deklarierten Datenobjekte und auf keine der nur hier deklarierten Prozeduren oder Funktionen kann von außen zugegriffen werden

- dem optionalen **Initialisierungsteil**, der Programmcode enthält, der beim Start des Programms, das diese Unit benutzt (siehe unten), durchlaufen wird und im wesentlichen zur Initialisierung der Datenstrukturen der Unit dient; hier sind beliebige Operationen im Rahmen des PASCAL-Sprachumfangs erlaubt, die sowohl die "öffentlich" als auch die "privat" deklarierten Datenobjekte verwenden.

Im Interfaceteil werden die Schnittstellen (Aufrufformate) für Prozeduren und Funktionen, die die Unit bereitstellt, und "öffentliche" Daten der Unit deklariert. Die Realisierung einer Prozedur bzw. Funktion, der Prozedur- bzw. Funktionsrumpf also, wird im Implementierungsteil nach außen vollständig verborgen.

Den syntaktischen Aufbau einer Unit lautet:

```
UNIT name;

INTERFACE

  USES <Liste weiterer von dieser UNIT "öffentlich"
        verwendeter UNITs                          (*)>
       { optional } ;

  <"öffentliche" Deklarationen>;

IMPLEMENTATION

  USES <Liste weiterer von dieser UNIT "privat"
        verwendeter UNITs                (*)>
       { optional } ;

  <"privaten" Deklarationen                (*)>;

  <Prozedur- und Funktionsrümpfe der im Interfaceteil
   aufgeführten Prozeduren und Funktionen          (*)>

BEGIN

  <Programmcode zur Initialisierung der
   Datenobjekte                      (*)>
  { optional; der Initialisierungsteil umfaßt alle
    Anweisungen von BEGIN (einschließlich) bis
    END (ausschließlich)                          }

END.
```

(*) Der Text in den spitzen Klammern (< ... >) beschreibt hier eine syntaktische Einheit.

Eine Unit stellt ein eigenständiges Programm dar und führt zu einem eigenen (bezüglich anderer Programme getrennt übersetzten) Modul. Alle im Interface aufgeführten Deklarationen werden nach außen bekannt gegeben. Ein Programm kann eine Unit (mit Bezeichner name) als externes Programm durch Angabe von

```
USES name;
```

einbinden. Alle Bezeichner des Interfaceteils (aber nicht des Implementierungsteils) der angesprochenen Unit sind von dieser Programmstelle aus für das einbindende Programm global verfügbar und werden so behandelt, als wären sie an dieser Stelle im Programm deklariert worden (mit den entsprechenden Auswirkungen auf die Gültigkeit der Bezeichner). Insbesondere kann der Compiler prüfen, ob eine Prozedur, deren Prozedurkopf im Interfaceteil einer Unit deklariert ist und die von einem Programm, das diese Unit verwendet,

aufgerufen wird, bezüglich der Datentypen der korrespondierenden Aktual- und Formalparameter richtig versorgt wird. Alle Bezeichner, die im Implementierungsteil einer Unit deklariert werden, sind wie der Code des Implementierungsteils nach außen abgekapselt (Information Hiding).

Das folgende Beispiel erläutert Abhängigkeiten im Unit-Konzept:

Ein Programm mit Bezeichner `main` verwendet Prozeduren `high_proc` und `middle_proc`, die getrennt übersetzt in den Units `high` bzw. `middle` liegen. Innerhalb der Unit `high` wird ebenfalls die Prozedur `middle_proc` (aus der Unit `middle`) verwendet, die für die Unit `high` also extern ist. Die Unit `middle` greift auf eine extern in der Unit `low` definierte Prozedur `low_proc` zu. Die entsprechenden PASCAL-Programmteile und ihre gegenseitigen Abhängigkeiten sind in Abbildung 8.4-1 dargestellt. Zu beachten ist, daß der im Interfaceteil einer Unit aufgeführte Prozedurkopf einer Prozedur im Implementierungsteil wiederholt wird.

```
PROGRAM main;

  USES high, middle
             { Einbindung der "öffentlichen"
               Deklarationen der UNITs high und middle };

  VAR i: INTEGER;
      t : REAL;

BEGIN
  ...
  high_proc (i)  { in UNIT high   };
  ...
  middle_proc (t)  { in UNIT middle };
  ...
END.


UNIT high;

INTERFACE   { "öffentliche" Deklarationen }

  USES middle  { Zugriff auf die "öffentlichen"
                 Deklarationen der UNIT middle };

  PROCEDURE high_proc (p : INTEGER);

IMPLEMENTATION { "private" Deklarationen }

  PROCEDURE high_proc (p: INTEGER);

    VAR r_var : REAL { nach außen verborgen };

    BEGIN
     ...
     middle_proc (r_var);
     ...
    END  { high_proc };
  ...
END.
```

```
UNIT middle;

INTERFACE

  PROCEDURE middle_proc (r : REAL);

IMPLEMENTATION

  USES low;

  PROCEDURE privat_proc
              { "private, nach außen nicht
                 bekannte Prozedur der UNIT middle };
    BEGIN
    ...
    END  { privat_proc };

  PROCEDURE middle_proc (r: REAL);

    VAR in_char : CHAR;
    BEGIN
    ...
    low_proc (in_char);
    ...
    privat_proc  { Aufruf einer für die UNIT "privaten"
                   Prozedur innerhalb einer nach außen
                   bekannt gegebenen Prozedur           };
    ...
    END  { middle_proc };
  ...
END.


UNIT low;

INTERFACE

  PROCEDURE low_proc (VAR zeichen : CHAR);

IMPLEMENTATION

  PROCEDURE low_proc (VAR zeichen : CHAR);

    BEGIN
     ...
     zeichen := ...;
     ...
    END  { low_proc };
   ...
END.
```

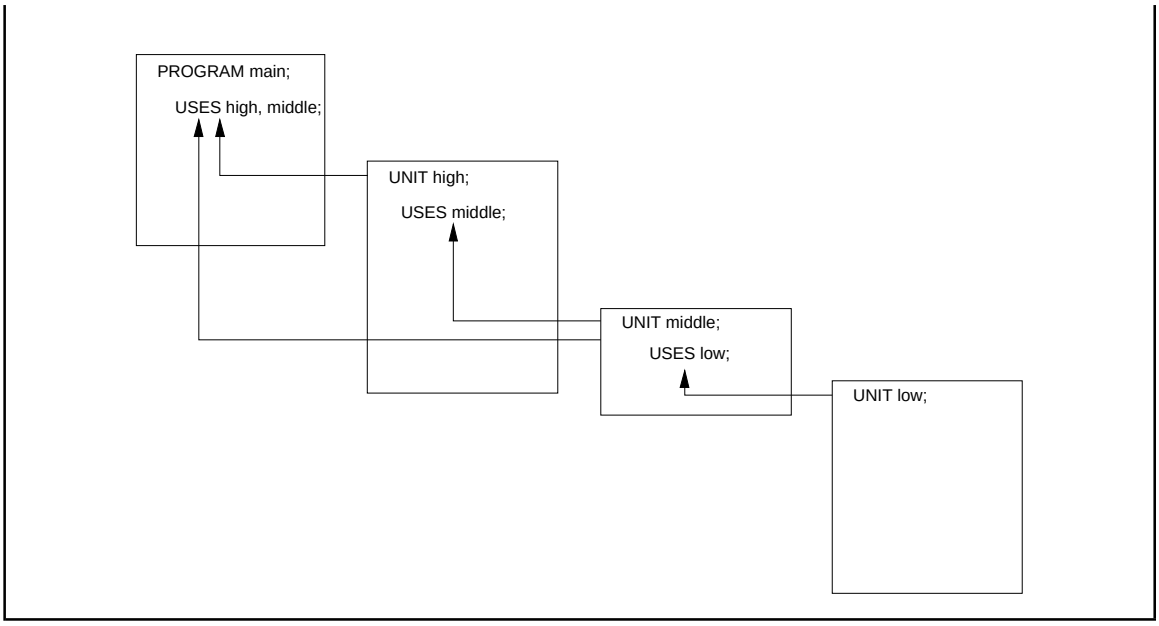


Abbildung 8.4-1: Units in PASCAL

Bei derartigen Abhängigkeiten ist die Übersetzungsreihenfolge der einzelnen Programmteile (`PROGRAM` und `UNIT`) von Bedeutung: Verwendet ein Programmteil mit der `USES`-Anweisung andere Units, d.h. ist dieser Programmteil vom Vorhandensein der Deklarationen der so einzubindenden Units ***abhängig***, so müssen diese Units zuvor übersetzt worden sein. In obigem Beispiel lautet die Reihenfolge:

```
UNIT low,
UNIT middle,
UNIT high,
PROGRAM main.
```

Wird der Interfaceteil einer Unit geändert, so müssen alle Programmteile, die diese Unit mittels `USES`-Anweisung benutzen, neu kompiliert werden. Die integrierte Entwicklungsumgebung von PASCAL unterstützt die **automatische Recompilierung abhängiger Programmteile nach Quellcodeänderungen**. Änderungen einer Unit im Implementierungs- oder Initialisierungsteil erfordern keine Rekompilation der abhängigen Programmteile. Nach Änderung und Rekompilation einer Unit und eventuell ihrer abhängigen Programmteile ist ein erneuter Bindevorgang notwendig. Selbstverständlich wird der Objektcode einer Unit, die in mehreren anderen Units per `USES`-Anweisung angesprochen wird, wie es im Beispiel mit der Unit `middle` geschieht, nur einmal angebunden. Außerdem bindet der Compiler nur diejenigen Teile einer Unit an, die auch verwendet werden.

8.5 Dynamische Bindebibliotheken

Die bisher beschriebene Methode wird als **statisches Binden** bezeichnet, weil das Anbinden auch externer Codeteile *vor* der Laufzeit des Programms erfolgt. Eine alternative Methode stellt das **dynamische Binden** dar. Dabei werden externe Referenzen erst zur Laufzeit aufgelöst und die durch externe Referenzen angesprochenen Objektmoduln *nach Bedarf* angebunden. Dabei wird angegeben, wo die einzelnen Objektmoduln zu finden sind, und

zwar durch Angabe entsprechender Bibliotheken (**dynamische Bindebibliothek, DLL, dynamic link library**). Wird auf eine externe Referenz Bezug genommen, so wird das entsprechende Objektmodul in der Bibliothek gesucht und angebunden (Auflösung externer Adreßreferenzen). Der dynamische Bindevorgang kann entweder im Augenblick des Programmladens ablaufen (**dynamisches Binden zur Ladezeit**, **load-time dynamic linking**) oder erst zur Laufzeit, wenn auf die anzubindende externe Refererenz auch wirklich Bezug genommen wird (**dynamisches Binden zur Laufzeit**, **run-time dynamic linking**).

Mit dem dynamischen Binden sind einige ***Vorteile*** verbunden:

- Ein Objektmodul wird erst dann angebunden, wenn es während der Laufzeit auch wirklich angesprochen wird. Das ablauffähige Programm als gebundenes Programm ist vom Umfang her kleiner.
- Das System kann dynamisch rekonfiguriert bzw. erweitert werden, ohne daß das komplette System neu übersetzt oder gebunden zu werden braucht; es wird nur der veränderte Teil in der DLL ersetzt bzw. eine neue DLL angebunden. Dies ist besonders wichtig bei Änderung des Rechners und zur Garantie der Portierbarkeit von Softwaresystemen.

Das Konzept der DLLs kommt in vielen Betriebssystemen vor. Ihr Aufbau und die Details des Umgangs mit einer DLL ist natürlich vom jeweiligen Betriebssystem abhängig. Exemplarisch soll im folgenden der Aufbau einer **DLL unter MS-DOS/WINDOWS** im Protected bzw. Flat Mode beschrieben werden.

Die Erstellung einer DLL erfolgt mit Hilfe des PASCAL-Compilers. Das Format der DLL ist jedoch sprachübergreifend, d.h. eine mit C oder C++ erstellte DLL kann in einem PASCAL-Programm angesprochen werden, und eine in PASCAL generierte DLL in Programmen, die in anderen Programmiersprachen geschrieben wurden.

Eine DLL ist ein ausführbares Modul, dessen Code und Ressourcen von anderen Anwendungen und DLLs verwendet werden können. Eine DLL ähnelt im Konzept einer Unit: beide stellen einem Programm Dienstleistungen in Form von Prozeduren und Funktionen zur Verfügung. Der Unterschied besteht im Zeitpunkt des Einbindens dieser Dienstleistungen in das verwendende Programm. Der Code einer Prozedur oder Funktion aus einer Unit wird zum Bindezeitpunkt *statisch* in das verwendende Programm eingebunden. Benötigen zwei unterschiedliche Programme dieselbe Prozedur einer Unit, so arbeitet das System mit zwei Kopien dieser Prozedur in den jeweiligen Programmen.

Verwendet ein Programm Code aus einer DLL, wird dieser im Gegensatz zum Code einer Unit ***nicht physisch*** in das Programm eingebunden. Vielmehr befindet sich der Code einer DLL in einer separaten ausführbaren Datei (mit Namenserweiterung .DLL), die während der Ausführung des Programms zugänglich sein muß und in den Arbeitsspeicher geladen wird. Die Prozedur- und Funktionsaufrufe im Programm werden ***dynamisch*** mit den vorgesehenen Einsprungpunkten in der verwendeten DLL verknüpft.

Units und DLLs unterscheiden sich außerdem dadurch, daß Units neben Prozeduren und Funktionen Typen, Konstanten, Daten und Objekte exportieren können, DLLs dagegen nur Prozeduren und Funktionen. Natürlich kann eine DLL eigene (lokale) Datenobjekte definieren, die beim Ablauf einer exportierten Prozedur verwendet werden.

Um eine Prozedur aus einer DLL in einem PASCAL-Programm zu verwenden, ist die Angabe des Prozedurkopfs notwendig, der durch die Benennung der DLL und einen optionalen Einsprungpunkt (Index) in die DLL ergänzt wird. Ein Beispiel, in der eine DLL namens KERNEL verwendet wird, die die Prozedur GlobalAlloc unter dem Index 15 enthält, ist

```
FUNCTION GlobalAlloc (Flags : WORD; Bytes : LongInt) : THandle;
         FAR; EXTERNAL 'KERNEL' INDEX 15;
```

Fehlt die **INDEX**-Anweisung, wird die Prozedur in der DLL über ihren Bezeichner (im Beispiel GlobalAlloc) gesucht, ein Vorgang, der mehr Zeit in Anspruch nimmt als die Suche über einen Index. Zusätzlich kann man einer Prozedur aus einer DLL im eigenen Programm mit Hilfe der **NAME**-Anweisung einen neuen Bezeichner geben:

```
FUNCTION GlobalAlloc (Flags : WORD; Bytes : LongInt) : THandle;
         FAR; EXTERNAL 'KERNEL' NAME 'eigenerName';
```

Die syntaktische Struktur einer DLL ist mit der Struktur eines Programms bis auf einige Schlüsselwörter identisch. Eine DLL beginnt mit dem Schlüsselwort **LIBRARY** (anstelle von PROGRAM). Jede Prozedur und Funktion, die anderen Programmen zur Verfügung gestellt werden soll, muß in ihrem Prozedurkopf mit dem Schlüsselwort **EXPORT** als zu exportierende Prozedur gekennzeichnet werden. Der Export selbst erfolgt in einer sich anschließenden **EXPORTS**-Klausel, in der auch mehrere zu exportierende Bezeichner aufgeführt werden können (die vorher durch EXPORT gekennzeichnet wurden) und in der für jeden Bezeichner ein Index (INTEGER-Konstante zwischen 1 und 32.767) angegeben werden kann. Des weiteren verfügt eine DLL über einen möglicherweise leeren **Initialisierungsteil**, in der z.B. Variablen der DLL mit Anfangswerten versehen werden. Beispielsweise kann auch eine Fehlerbedingung gemeldet werden, indem die Variable **ExitCode**, die in der Unit System deklariert ist, auf 0 gesetzt wird. Die Standardeinstellung von ExitCode ist 1; wird sie auf 0 gesetzt, so wird die DLL aus dem Arbeitsspeicher entfernt und die aufrufende Anwendung darüber informiert, daß die DLL nicht geladen werden konnte.

Zu beachten ist, daß alle Variablen einer DLL für diese privat sind, d.h. nicht exportiert werden können.

Das folgende Beispiel erstellt eine DLL mit Bezeichner MinMax, in der zwei Funktionen Min bzw. Max mit Indizes 1 bzw. 2 bereitgestellt werden:

```
LIBRARY MinMax;

FUNCTION Min (x, y: INTEGER) : INTEGER; EXPORT;
  BEGIN { Min }
    IF x < y THEN Min := x ELSE Min := y;
  END   { Min };

FUNCTION Max (x, y: INTEGER) : INTEGER; EXPORT;
  BEGIN { Max }
    IF x < y THEN Max := y ELSE Max := x;
  END   { Max };

EXPORTS
  Min INDEX 1;
  Max INDEX 2;

BEGIN { MinMax }
  { Initialisierungsteil, hier leer }
END   { MinMax }.
```

Im Anweisungsteil einer DLL befindet sich der Initialisierungscode, der bei der ersten Verwendung einer DLL einmalig durchlaufen wird. Es wird ein **Verwendungszähler** mitgeführt, der angibt, wieviele Programme gerade Prozeduren der DLL verwenden. Durch den Aufruf einer Prozedur in einer DLL, die bereits in Verwendung ist, wird der Initialisierungsteil nicht erneut durchlaufen, sondern lediglich der Verwendungszähler inkrementiert. Die DLL bleibt solange im Arbeitsspeicher geladen, wie der Verwendungszähler größer als 0 ist. Erst wenn der Verwendungszähler auf 0 geht, wird die DLL aus dem Speicher entfernt. Dabei wird zunächst in der DLL eine exportierte Funktion mit Bezeichner WEP gesucht und, falls vorhanden, aufgerufen. Eine PASCAL-DLL exportiert automatisch eine sogenannte **WEP-Funktion**. Diese ruft nun die in der Variablen `ExitProc` der Unit `System` gespeicherte Adresse solange auf, bis `ExitProc` den Wert `NIL` enthält. Innerhalb der DLL kann die Variable `ExitProc` mit einer geeigneten Adresse versehen werden, um damit ein gezieltes Terminierungsverhalten der DLL zu erzwingen.

Für weitere Details und Besonderheiten wie das Fehlen eines Stacksegments bei einer DLL und die sich daraus ergebenden Konsequenzen bei der Programmierung, die Behandlung von Laufzeitfehlern usw. wird auf die angegebene Literatur verwiesen.

9 Einführung in die objektorientierte Programmierung

In der **objektorientierten Programmierung (OOP)** ([COX]) wird ein Problem dadurch zu lösen versucht, daß man ein ***Modell von miteinander kommunizierenden Objekten*** aufstellt. Dazu identifiziert man aus dem jeweiligen Realitätsbereich **Objekte**, ihre gegenseitigen Abhängigkeiten und die mit den Objekten verknüpften **Operationen**, die in der OOP **Methoden** genannt werden. Die Formulierung des Modells erfolgt in einer geeigneten Programmiersprache.

OOP ist bedingt in "klassischen" Programmiersprachen möglich. Das Gesamtkonzept wird aber durch spezielle Sprachen (Smalltalk, Flavor, Eifel usw.) anwendungsorientiert unterstützt ([W/C]). Einen Ansatz bildet die Verwendung abstrakter Datentypen, mit deren Hilfe jedoch nicht alle Forderungen der objektorientierten Programmierung wie beispielsweise die multiple dynamische Vererbung modellierbar sind. Andere Ansätze erweitern bekannte Sprachen. Beispiele sind C++, Java, das aus C++ hervorgegangen ist, Borland Pascal (ab Version 5.5) und der Pascal-Dialect Object Pascal (in Delphi, siehe [WAR]). Die Erweiterungen in Borland Pascal ähneln dem Ansatz in C++, ohne jedoch auf die sonstigen Pascal-Konzepte zu verzichten. Außerdem sind nicht alle Konzepte der OOP in die Sprache eingeflossen, wodurch eine mehr oder weniger hybride Sprache entstanden ist (dies gilt sowohl für C++ als auch für PASCAL). Allein eine Programmierung mit einer dieser Sprachen macht noch nicht die OOP aus. Die Verwendung von Java zwingt jedoch schon eher zur Einhaltung der OOP-Vorgehensweise.

9.1 Konzepte der objektorientierten Programmierung

Ein Charakteristikum der Objekte vieler Modelle ist die Tatsache, daß Objekte gemeinsame Eigenschaften aufweisen und zu einer **(Objekt-) Klasse** zusammengefaßt werden. ***Alle Objekte einer Objektklasse haben daher dieselben Eigenschaften; insbesondere gleichen sich alle Methoden auf (Operationen mit) diesen Objekten.*** Objekte unterschiedlicher Objektklassen können durchaus gemeinsame (Teil-) Eigenschaften besitzen; mindestens eine Eigenschaft trennt jedoch die Objekte der verschiedenen Objektklassen. Es können **Hierarchien von Objektklassen** aufgebaut werden: Eine Objektklasse übernimmt alle Eigenschaften einer **übergeordneten Objektklasse** und fügt weitere Eigenschaften hinzu, spezialisiert also dadurch eine in der Hierarchie weiter oben stehende Objektklasse. Anwendbar sind alle Operationen, die in einer übergeordneten Objektklasse definiert sind, auch auf die Objekte einer dieser Klasse untergeordneten Objektklasse, falls die Operationen in der untergeordneten Klasse nicht neu definiert werden. Eine Objektklasse übernimmt ja von einer übergeordneten Objektklasse alle Eigenschaften; die dort definierten Operationen beziehen sich nur auf diese übernommenen Eigenschaften. Der Vorgang der Übernahme von Eigenschaften übergeordneter Objektklassen nennt man **Vererbung (inheritance)**. Die Art und Weise der Vererbung, z.B. die Vererbung aus einer einzigen Oberklasse (**einfache Vererbung**) oder mehreren Oberklassen (**multiple Vererbung**), und der Zeitpunkt der Vererbung, z.B. bei der Deklaration eines Objekts oder erst bei einer Operationsanwendung (**dynamische Vererbung**), prägen die OOP.

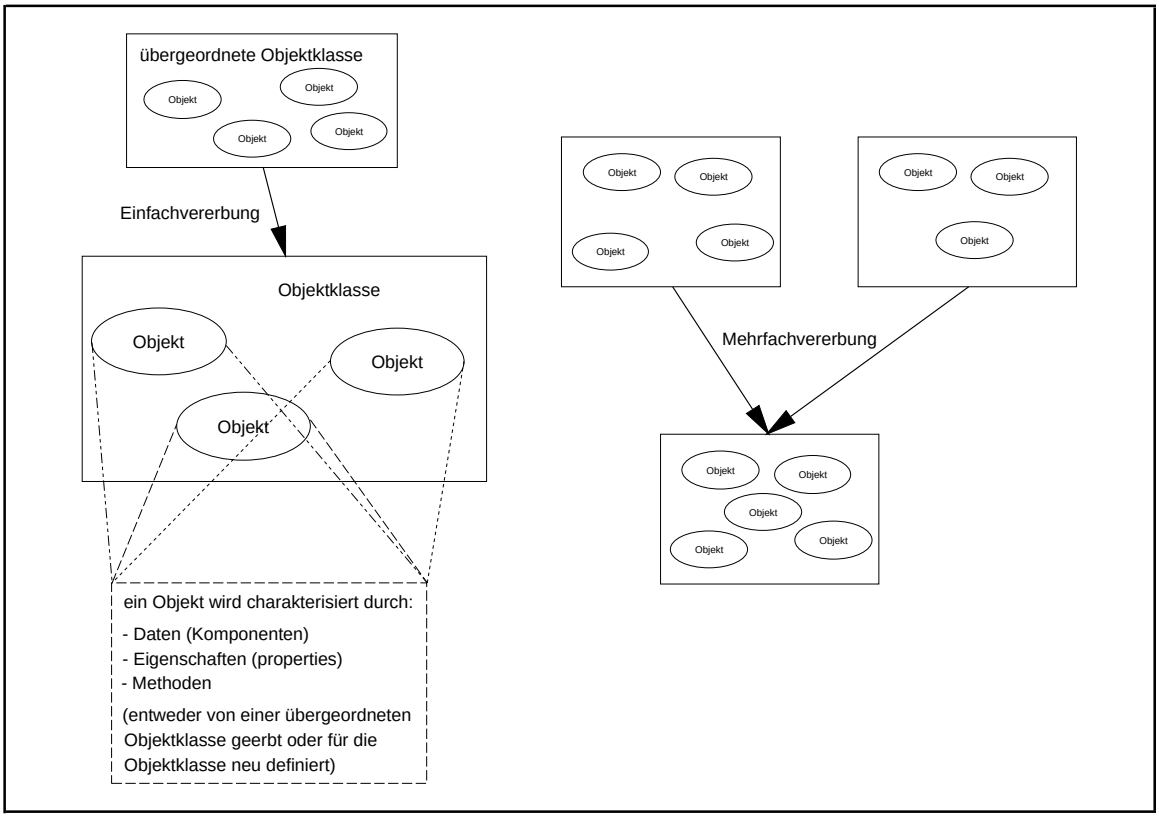


Abbildung 9-1: Objektkonzept der objektorientierten Programmierung

Wie Abbildung 9-1 zeigt, werden alle Objekte einer Objektklasse durch dieselben Charakteristika beschrieben. Jedes Objekt besteht aus:

- **Daten**, auch **Komponenten** genannt: sie besitzen einen Datentyp und können Werte annehmen, die während der Lebensdauer des Objekts veränderbar sind; die Komponenten entsprechen den Komponenten eines durch einen `RECORD`-Typ definierten Datenobjekts im klassischen Pascal

- **Eigenschaften (properties)**: sie gleichen den Daten des Objekts, erlauben es aber zusätzlich, definierte Aktionen (Prozedurabläufe) mit dem Lesen einer Eigenschaft bzw. dem Verändern einer Eigenschaft zu verbinden

- **Methoden**: Prozeduren oder Funktionen, die auf die Komponenten und Eigenschaften des Objekts zugreifen und deren Werte eventuell verändern

Zusätzlich wird jedes Objekt durch einen eindeutigen **Objektbezeichner** identifiziert. Die gegenwärtigen Werte der Daten (Komponenten) und Eigenschaften eines Objekts bestimmen seinen **Objektzustand**.

Prinzipiell besteht eine vollständige **Kapselung** der Daten und Eigenschaften eines Objekts nach außen hin, d.h. nur die für ein Objekt definierten Methoden können auf die einzelnen Komponenten des Objekts zugreifen und deren Werte verändern. Gerade dieser Aspekt ist

jedoch in den Programmiersprachen, die erst später um die Konzepte der OOP erweitert wurden wie C++ oder PASCAL, nicht streng realisiert, so daß die Einhaltung der Kapselung dem Programmierer eine gewisse Programmierdisziplin auferlegt.

Die **Interaktion zwischen Objekten** erfolgt durch **Austausch von Nachrichten**: Ein Objekt ruft eine Methode eines anderen Objekts auf und verändert dadurch dessen Objektzustand (und folglich damit den Zustand des Gesamtsystems). Das Zielobjekt reicht mit dem gleichen Mechanismus eventuell ein Resultat zurück. Das Versenden einer Nachricht ist damit mit dem Prozeduraufruf in einer konventionellen Programmiersprache vergleichbar. In der OOP ist eine Nachricht der Bezeichner eines Objekts zusammen mit dem Bezeichner einer seiner Methoden. Der Methodenaufruf kann auch mit Parametern versehen sein. Dabei ist zu beachten, daß der gleiche Methodenbezeichner bei unterschiedlichen Datenobjekten auch unterschiedliche Reaktionen hervorrufen kann und dies sogar in "kontrolliertem Maß" bei unterschiedlichen Objekten derselben Objekthierarchie. Dieses Konzept wird als **Polymorphie** bezeichnet. In Kapitel 9.2 wird darauf näher eingegangen.

In diesem Interaktionskonzept ist der Empfänger einer Nachricht dem Absender der Nachricht bekannt. Einige Systeme (z.B. Delphi oder PASCAL mit Turbo Vision) erlauben zusätzlich das Versenden von **Botschaften**: eine Botschaft wird durch einen **Botschaftsbezeichner** identifiziert und kann weitere Parameter enthalten. Eine Objektklasse hat zu diesem Botschaftsbezeichner eine Methode, die **Botschaftsbehandlungsmethode**, deklariert. Beim Absenden einer Botschaft durch ein Objekt wird zunächst in der eigenen Objektklasse nach der Botschaftsbehandlungsmethode zu dieser Botschaft gesucht und bei einer nicht erfolgreichen Suche durch die Objektklassenhierarchie gegangen, bis eine entsprechende Botschaftsbehandlungsmethode gefunden wurde. In diesem Konzept kennt das Objekt, das die Botschaft senden, den Empfänger der Botschaft nicht.

In den folgenden Kapiteln werden die beschriebenen Konzepte der OOP an Beispielen, die in Borland Pascal formuliert sind, erläutert und dabei Definitionsmöglichkeiten von Datenobjekten in der OOP aus dem systemnahen Programmierumfeld aufgezeigt.

9.2 Realisierung der Konzepte in PASCAL

PASCAL erlaubt die Anwendung der wichtigsten Prinzipien der OOP. Einschränkungen bestehen in der Form der Vererbung: es ist nur die einfache Vererbung definiert, wodurch eine Reihe von Problemen (z.B. die Auflösung von Namenskonflikten bei der Vererbung) auf Kosten der Flexibilität umgangen wird. Des weiteren wird die Kapselung von Datenobjekten nicht standardmäßig kontrolliert; es ist ohne weiteres möglich, Werte von Komponenten eines Datenobjekts, die nicht durch das Schlüsselwort `PRIVATE` (siehe unten) geschützt sind, zu ändern, ohne die dazu definierten Methoden zu verwenden. Auch das Konzept der Kommunikation zwischen Objekten mit Hilfe von Botschaften ist nicht realisiert; die Klassenbibliothek Turbo Vision enthält dieses Konzept und nennt die Botschaften Events. Botschaften sind als Sprachkonzept in Object Pascal enthalten ebenso wie die "Eigenschaften (properties)" genannten Charakteristika eines Datenobjekts, die man ebenfalls in PASCAL nicht findet. Die wichtigsten Merkmale der OOP, nämlich die Definition

von Objekten und Objektklassen, die (einfache) Vererbung und die Polymorphie von Methoden, findet man als Sprachkonzepte in PASCAL in sehr "handlicher" Weise. Diese sollen im folgenden an Beispielen erläutert werden.

Das erste Beispiel ([HOF]) behandelt ein Modell[30], dessen Objekte ***Blöcke*** genannt werden. Jeder Block besteht aus zwei Teilen, einem ***Kopf*** und einem ***Nutzteil***, der ***Einzeleinträge*** enthält. Der Kopf selbst ist in vier Komponenten gegliedert. Sie enthalten Informationen, die Auskunft über die Anzahl von Einzeleinträgen im Nutzteil geben (in der Komponente `anz`) bzw. die Adreßverweise auf weitere Blöcke darstellen (in den Komponenten `vorg`, `last` und `next`). Es gibt zwei Typen von Blöcken, die sich durch das Layout ihres Nutzteils unterscheiden; der Kopf ist bei allen Blöcken gleichartig strukturiert. Ein Block vom Typ ***Datenblock*** enthält in seinem Nutzteil bis zu 2*v* Einträge der Form (*k*, *data*); dabei ist *v* ein fester Parameter. Der Datentyp und die Bedeutung der Einträge *k* und *data* spielen an dieser Stelle keine Rolle. Ein Block vom Typ ***Indexblock*** enthält in seinem Nutzteil einen Verweis (in Form einer Adresse) auf einen anderen Block und bis zu 2*u* Einträge der Form (*k*, *p*); auch hier ist *u* ein fester Parameter. Die Teilkomponente *k* hat denselben Datentyp wie die Teilkomponente *k* in einem Datenblock; bei *p* handelt es sich um einen Adreßverweis auf einen anderen Block (Daten- oder Indexblock). Die genaue Bedeutungen der Einträge *k* und *p* spielen hier ebenfalls keine Rolle. Die Größen eines Daten- und eines Indexblocks können sich unterscheiden; gemeinsam ist ihnen nur, daß sie einen Kopfteil und einen Nutzteil besitzen.

Es ist also sinnvoll, Objektklassen `datenblock` und `indexblock` zu definieren, die einer Objektklasse `block` untergeordnet sind. In der Objektklasse `block` wird der Kopfteil eines Blocks definiert; die unterschiedlichen Layouts der Daten- bzw. Indexblöcke werden zusätzlich in der Objektklasse `datenblock` bzw. `indexblock` festgelegt. Die beiden Typen von Blöcken sind im oberen Teil von Abbildung 9.2-1 zu sehen.

Datenblöcke und Indexblöcke sind über die Adreßverweise im Kopfteil logisch miteinander verknüpft (die Gesamtstruktur zeigt der untere Teil von Abbildung 9.2-1; sie ist für das Beispiel hier jedoch ohne Bedeutung).

[30] Das Modell beschreibt die Speicherung von Datensätzen in Form höhenbalancierter Bäume (vgl. Kapitel 10.2.3). Auf die Datensätze sind Zugriffe über Primärschlüssel in der Weise möglich, daß die Zugriffszeit auf jeden Datensatz die gleiche Zeit in Anspruch nimmt.

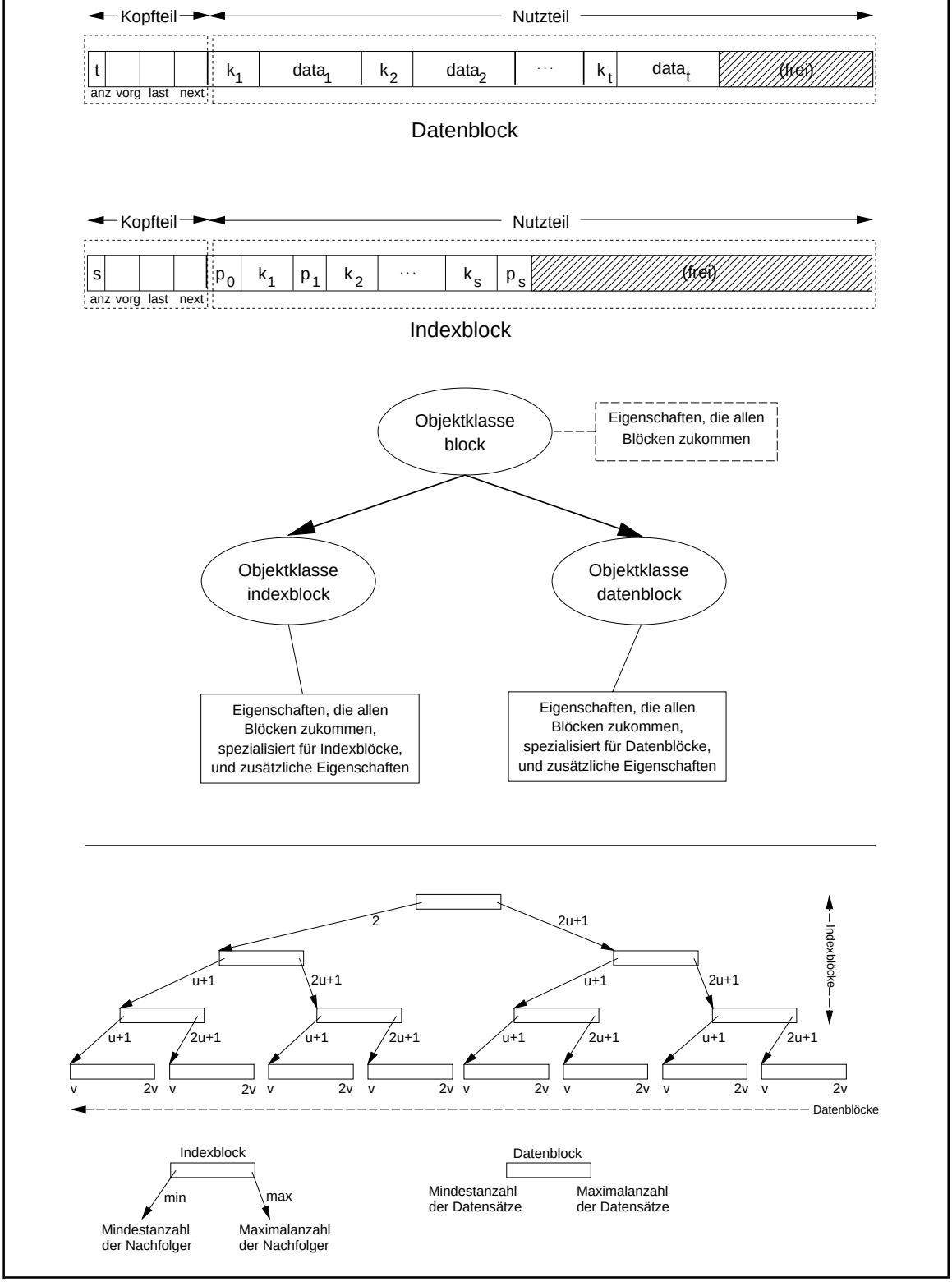


Abbildung 9.2-1: Blöcke (Beispiel)

Neue Blöcke können erzeugt und in die logische Struktur eingefügt bzw. aus der logischen Struktur entfernt und vernichtet werden. Einzeleinträge können in einen Datenblock eingefügt bzw. aus ihm entfernt werden. Das gleiche ist für Einzeleinträge in Indexblöcken möglich. Zusätzlich sollen die Inhalte eines Blocks (Kopfteil und Einzeleinträge in Datenblöcken bzw. Indexblöcken) angezeigt werden können.

In der Objektklasse `block` werden also die Komponenten des Kopfteils und die Methoden

- `init` zur Erzeugung eines neuen Blocks und Initialisierung des Kopfteils
- `insert` zum Einfügen eines Blocks (Daten- oder Indexblock) in die Gesamtstruktur
- `delete` zum Entfernen eines Blocks aus der Gesamtstruktur und zur "Auflösung" des Blocks
- `display` zum Anzeigen der Gesamtstruktur

definiert. Für die Objektklassen `datenblock` bzw. `indexblock` gibt es neben den Komponenten des jeweiligen Nutzteils die Methoden

- `init` zur Initialisierung des jeweiligen Nutzteils
- `insert_entry` zum Einfügen eines Eintrags in den jeweiligen Nutzteil
- `delete_entry` zum Entfernen eines Eintrags aus dem jeweiligen Nutzteil
- `display` zum Anzeigen des jeweiligen Nutzteils

Ein Datenobjekt wird in der OOP in PASCAL wieder in zwei Stufen vereinbart. Zunächst muß die Objektklasse, der das Datenobjekt angehören soll, deklariert werden. Dies geschieht durch Definition des **Objekttyps** (der Objektklasse), der die Struktur (Komponenten) eines Datenobjekts dieser Klasse und die Methoden zur Bearbeitung eines Datenobjekts in dieser Objektklasse bestimmt. Anschließend wird das Datenobjekt mit diesem Objekttyp deklariert und dadurch eine **Instanz** (Exemplar) des Objekttyps erzeugt. Alle Datenobjekte (Instanzen) mit demselben Objekttyp bilden die Objektklasse.

PASCAL verwendet zur Definition eines Objekttyps das Sprachkonstrukt **OBJECT**[31]. Es ähnelt dem `RECORD`-Sprachkonstrukt, definiert aber zusätzlich die Methoden, die mit einem Datenobjekt dieses Objekttyps verbunden sind. Außerdem können alle Eigenschaften einer Klasse an andere Klassen vererbt werden. Dazu wird der Bezeichner des Objekttyps, der die vererbende Klasse beschreibt, in Klammern an das Schlüsselwort `OBJECT` des Objekttyps angehängt, der die erbende Klasse charakterisiert. Die **syntaktische Form einer Objekttypdeklaration** lautet:

```
TYPE objekttyp_bezeichner = OBJECT (bezeichner_des_Vorfahren)
                              < Komponenten des Objekttyps>;
                              < Methodenköpfe des Objekttyps >
                            END;
```

[31] Besser wäre hier ein anderslautendes Schlüsselwort, um zu verdeutlichen, daß der Datentyp einer Objektklasse deklariert wird und nicht etwa ein Objekt. Object Pascal verwendet an dieser Stelle das Schlüsselwort CLASS, wodurch jedoch ein erweitertes Objektmodell definiert wird (vgl. Kap. 9.3 und [BDE]).

Da alle Eigenschaften der Datenobjekte einer Klasse durch den zugehörigen Objekttyp festgelegt sind, kann man den Begriff der Vererbung auf die beschreibenden Objekttypen beziehen. Ein Objekttyp (genauer: die durch den Objekttyp beschriebene Objektklasse) erbt die Eigenschaft eines übergeordneten Objekttyps (genauer: der durch den Objekttyp beschriebenen übergeordneten Klasse). Die Hierarchie der Klassen spiegelt sich in der Hierarchie der die Klassen charakterisierenden Objekttypen wieder. ***Im folgenden werden daher die Begriffe Objektklasse und Objekttyp synonym verwendet.***

Die Methoden in einer `OBJECT`-Deklaration werden lediglich durch ihren Prozedurkopf (Methodenkopf) benannt. Sie sind implizit **`FORWARD`**-Deklarationen; das bedeutet, daß die komplette Deklaration des Methodenrumpfs erst später innerhalb des Moduls erfolgt. Dabei wird dann zur Unterscheidung gleichlautender Methodenbezeichner der Bezeichner des Objekttyps dem Methodenbezeichner vorangestellt und durch eine Punkt von ihm getrennt.

Im Beispiel könnten die Deklarationen wie folgt in einem Programm namens OOP vorkommen (einige Programmteile sind lediglich durch Punkte angedeutet). Hierbei ist noch zu beachten, daß Datenobjekte mit den jeweiligen Objekttypen wie Variablen deklariert werden.

Um eine Methode auf ein Datenobjekt anzuwenden, wird der Datenobjektname mit der zugehörigen Methode durch einen Punkt verbunden.

```
PROGRAM oop;

CONST u               = ...;
      v               = ...;
      k_init_value    = ...;
      data_init_value = ...;

TYPE data_typ        = ... ;
     key_typ         = ...;
     data_entry_typ  = RECORD
                         k    : key_typ;
                         data : data_typ
                       END;
     data_sect_typ   = ARRAY [1..2*v] OF data_entry_typ;

     index_entry_typ = RECORD
                         k : key_typ;
                         p : block_ptr
                       END;
     index_sect_typ  = ARRAY [1..2*u] OF index_entry_typ;

     block_ptr = ^block;
     block     = OBJECT
                   anz  : INTEGER;
                   vorg : block_ptr;
                   last : block_ptr;
                   next : block_ptr;
                   { Methoden von block }
                   PROCEDURE init;
                   PROCEDURE insert;
                   PROCEDURE delete;
                   PROCEDURE display
                 END;
```

```
      datenblock = OBJECT (block)
                     data : data_sect_typ;
                     { Methoden von datenblock }
                     PROCEDURE init;
                     PROCEDURE insert_entry
                                  (new_entry : data_entry_typ);
                     PROCEDURE delete_entry
                                  (del_entry : data_entry_typ);
                     PROCEDURE display
                   END;

      indexblock = OBJECT (block)
                     p_0   : block_ptr;
                     index : index_sect_typ;
                     { Methoden von indexblock }
                     PROCEDURE init;
                     PROCEDURE insert_entry
                                  (new_entry: index_entry_typ);
                     PROCEDURE delete_entry
                                  (del_entry: index_entry_typ);
                     PROCEDURE display
                   END;

{ Implementierung der Methoden von block }

PROCEDURE block.init;
{ Initialisierung des Kopfteils }
  BEGIN { block.init }
    anz  := 0;
    vorg := NIL;
    last := NIL;
    next := NIL
  END   { block.init };

PROCEDURE block.insert;

  BEGIN { block.insert }
    { den Block in die Gesamtstruktur einfügen: }
    ...
  END   { block.insert };

PROCEDURE block.delete;

  BEGIN { block.delete }
    { den Datenblock aus der Gesamtstruktur entfernen: }
    ...
  END   { block.delete };

PROCEDURE block.display;

  BEGIN { block.display }
    { Anzeigen der Komponenten anz, vorg, last und next
      des Kopfteils:                                    }
    ...
  END   { block.display };

{ Implementierung der Methoden von datenblock }

PROCEDURE datenblock.init;

  VAR idx : INTEGER;

  BEGIN { datenblock.init }
    { Kopfteil initialisieren: }
    block.init;
    { Nutzteil initialisieren: }
    FOR idx := 1 TO 2*v DO
```

```
      BEGIN
        data[idx].k    := k_init_value;
        data[idx].data := data_init_value
      END
  END   { datenblock.init };

PROCEDURE datenblock.insert_entry
              (new_entry : data_entry_typ);

  BEGIN { datenblock.insert_entry }
    IF anz < 2*v
    THEN BEGIN
           { new_entry in den Nutzteil einfügen : }
           anz            := anz + 1;
           data[anz].k    := new_entry.k;
           data[anz].data := new_entry.data
         END
    ELSE BEGIN
           { Überlauf des Datenblocks behandeln : }
           ...
         END
  END   { datenblock.insert_entry };

PROCEDURE datenblock.display;

  VAR idx : INTEGER;

  BEGIN { datenblock.display }
    { Kopfteil anzeigen: }
    block.display;
    FOR idx := 1 TO anz DO
      BEGIN
        { data[idx].k und data[idx].data anzeigen: }
        ...
      END
  END   { datenblock.display };

PROCEDURE datenblock.delete_entry
              (del_entry : data_entry_typ);

  VAR idx : INTEGER;

  FUNCTION pos (entry: data_entry_typ) : INTEGER;
    { pos(entry) ermittelt die Position im Nutzteil des Datenblocks,
      an der entry steht; ist entry nicht im Nutzteil enthalten, so
      ist pos(entry) = 0                                            }
    BEGIN { pos }
      ...
    END   { pos };

    BEGIN { datenblock.delete_entry }
      idx := pos(del_entry);
      IF idx <> 0
      THEN BEGIN { del_entry entfernen und alle hinteren
                   Einträge nach vorn schieben:          }
             idx := idx+1;
             WHILE idx <= anz DO
               BEGIN
                 data[idx-1].k    := data[idx].k;
                 data[idx-1].data := data[idx].data;
                 idx              := idx+1
               END;
             anz := anz-1;
             IF anz = 0
             THEN { datenblock entfernen }
```

```
            block.delete
          END
    END { datenblock.delete_entry };

{ Implementierung der Methoden von indexblock }

PROCEDURE indexblock.init;

  VAR idx : INTEGER;

  BEGIN { indexblock.init }
    { Kopfteil initialisieren: }
    block.init;
    { Nutzteil initialisieren: }
    p_0 := NIL;
    FOR idx := 1 TO 2*u DO
      BEGIN
        index[idx].k := k_init_value;
        index[idx].p := NIL
      END
  END   { indexblock.init };

PROCEDURE indexblock.insert_entry
              (new_entry : index_entry_typ);

  BEGIN { indexblock.insert_entry }
    { den Einzeleintrag new_entry in den Nutzteil einfügen;
      dabei einen Indexblock-Überlauf abfangen:            }
    ...
  END   { indexblock.insert_entry };

PROCEDURE indexblock.delete_entry
              (del_entry : index_entry_typ);

  BEGIN { indexblock.delete_entry }
  { den Einzeleintrag del_entry aus dem Nutzteil entfernen;
    einen dabei eventuell entstehenden leeren Indexblock aus
    der Gesamtstruktur entfernen:                          }
    ...
  END { indexblock.delete_entry };

PROCEDURE indexblock.display;

  BEGIN { indexblock.display }
    { Kopfteil und Nutzteil anzeigen: }
    ...
  END   { indexblock.display };


VAR d_block_1 : datenblock;
    d_block_2 : datenblock;
    i_block   : indexblock;

    d_entry : data_entry_typ;

BEGIN { oop }
  { Initialisierung der Datenobjekte d_block_1
    und d_block_2:                              }
  d_block_1.init;
  d_block_2.init;

  { Ausfüllen und Einfügen des Einzeleintrags d_entry
    in den Datenblock d_block_1:                      }
  WITH d_entry DO
    BEGIN
      k    := ...;
```

```
      data := ...
    END;
  d_block_1.insert_entry (d_entry);

  { Anzeigen des Datenblocks d_block_1: }
  d_block_1.display;

  { Anzeigen des Datenblocks d_block_2: }
  d_block_2.display;

  { Initialisierung des Datenobjekts (Indexblock) i_block: }
  i_block.init;

  ...
  { Anzeigen des Indexblocks i_block: }
  i_block.display;
  ...
END   { oop }.
```

Innerhalb der Definition einer Methode sind die Bezeichner für Komponenten des zugehörigen Objekttyps bekannt: Beispielsweise versorgt die Methode `block.init` die im Objekttyp `block` definierten Komponenten eines Datenobjekts mit diesem Datentyp mit Anfangswerten. In der Methode `datenblock.insert_entry` wird außerdem auf eine Komponente des übergeordneten Objekttyps `block`, nämlich auf die Komponente `anz`, zugegriffen; die Definition des übergeordneten Objekttyps `block` ist an den Objekttyp `datenblock` vererbt worden. Die Definition einer Methode und des Objekttyps, der diese Methode beinhaltet, liegen im selben **Gültigkeitsbereich**. Den Gültigkeitsbereich eines Bezeichners kann man auf den Modul beschränken, der die Objekttypdeklaration enthält (siehe [BP7]): Eine Objekttypdeklaration und ihre Komponenten und Methoden sind in anderen Units bekannt, wenn die Objekttypdeklaration im `INTERFACE`-Teil einer Unit steht. Um die Gültigkeit einiger Bezeichner innerhalb einer Objekttypdeklaration nur auf den Modul (Programm oder Unit) zu beschränken, der die Deklaration enthält, kann man diesen Bezeichnern das Schlüsselwort **`PRIVATE`** voranstellen. Dadurch sind diese `PRIVATE`-Bezeichner in anderen Moduln nicht bekannt. Folgende Bezeichner, die wieder allgemein öffentlich sein sollen, müssen mit dem Schlüsselwort **`PUBLIC`** eingeleitet werden. Von dieser Möglichkeit wird in den Beispielen in den folgenden Kapiteln Gebrauch gemacht.

Ein Vorteil der Vererbung wird deutlich: Wird das Layout bzw. eine Methode des Objekttyps block geändert, so wird diese Änderung sofort auch in `datenblock` und `indexblock` wirksam, ohne daß dort die Methoden angepaßt zu werden brauchen. Die Änderungen des allgemeinen Objekttyps werden also an die spezielleren Nachkommen vererbt. Die Definition eines neuen Blocktyps, der ebenfalls aus Kopfteil und Nutzteil besteht, der sich von denjenigen der Daten- und Indexblöcke unterscheidet, ist ebenfalls ohne Änderung des bisherigen Quellcodes möglich. Ist der Name des neuen Objekttyps z.B. `neu_block`, so definiert

```
TYPE neu_block = OBJECT (block)
                   { neuer Nutzteil: }
                   ...
                 { Methoden von neu_block, die sich auf
                   den neuen Nutzteil beziehen; für den
                   Zugriff auf den Kopfteil werden die
```

```
    Methoden von block genommen:       }
    ...
END;
```

einen neuen Objekttyp.

Die Methode `display` wird für jeden der drei vorkommenden Objekttypen `block`, `datenblock` und `indexblock` getrennt definiert, wobei die Methoden von `datenblock` und `indexblock` zur Anzeige des Kopfteils die Methode von `block` verwenden. Würden keine eigenen Methoden mit Bezeichner `display` für die Objekttypen `datenblock` und `indexblock` definiert werden, so würden durch die Aufrufe

```
d_block_1.display;
d_block_2.display;
```

und

```
i_block.display;
```

die Methode `display` des übergeordneten Objekttyps `block` appliziert und lediglich die Komponenten des Kopfteils angezeigt. Die Methode `display` würde also an die dem Objekttyp `block` untergeordneten Objekttypen `datenblock` und `indexblock` vererbt.

Diese Form der Vererbung nennt man **statische Vererbung**. Der Compiler kann zur Übersetzungszeit bereits alle so weitergegebenen Eigenschaften von Objekttypen auflösen. Insbesondere wird jetzt bereits beim Aufruf einer Methode geprüft, ob diese für den entsprechenden Objekttyp definiert ist. Falls es zutrifft, erfolgt ein Unterprogrammsprung in den Objektcode der Methode. Ist keine Methode für den entsprechenden Objekttyp definiert, geht der Compiler zum direkt übergeordneten Objekttyp (der in Klammern hinter dem Schlüsselwort `OBJECT` steht) über und sucht hier nach der Definition der Methode. Ist die Suche auch hier erfolglos, wird die Suche nach dem Code der Methode auf den weiteren übergeordneten Objekttyp ausgedehnt, falls es ihn gibt. Erst wenn die Methode bis in die oberste Hierarchiestufe nicht gefunden wird, bricht der Compiler die Suche mit einer Fehlermeldung ab. Die Auflösung des Zugriffs auf Methoden während der Übersetzungszeit wird als **frühe Bindung** bezeichnet.

Ein wichtiger Randeffekt der statischen Vererbung von Methoden ist zu beachten. Wird in einem Objekttyp O_1 eine Methode mit Namen `xyz` definiert und ebenso in einem übergeordneten Objekttyp O_2, so wird bei Aufruf der Methode `xyz` mit einem Datenobjekt mit Objekttyp O_1 die Methode `xyz` von O_1 genommen. Ist andererseits eine Methode namens `meth` im übergeordneten Objekttyp O_2, aber nicht in O_1 definiert, so wird bei Aufruf von `meth` mit einem Datenobjekt mit Objekttyp O_1 die Methode `meth` aus O_2 aktiviert. Verwendet `meth` seinerseits die Methode `xyz`, so wird bei Aufruf von `meth` die Methode `xyz` aus O_2 genommen, auch wenn `meth` von einem Datenobjekt mit Objekttyp O_1 aufgerufen wurde. Die Situation wird an folgendem Programmbeispiel und in der Abbildung 9.2-2 illustriert.

```
PROGRAM oop_stat;

TYPE o_2 = OBJECT
             { Komponenten von o_2:}
             flag : BOOLEAN;
             i    : INTEGER;
             { Methoden von o_2: }
             PROCEDURE init (wert : BOOLEAN;
                             x    : INTEGER);
             PROCEDURE del;
             PROCEDURE xyz;
             PROCEDURE meth;
           END;

     o_1 = OBJECT (o_2)
             { zusätzliche Komponenten von o_1:}
             j : INTEGER;
             { Methoden von o_1: }
             PROCEDURE init (f : BOOLEAN;
                             x : INTEGER;
                             y : INTEGER);
             PROCEDURE erase;
             PROCEDURE xyz;
           END;

{ Implementierung der Methoden von o_2: }

PROCEDURE o_2.init (wert : BOOLEAN;
                    x    : INTEGER);
  BEGIN { o_2.init }
    flag := wert;
    i    := x
  END   { o_2.init };

PROCEDURE o_2.xyz;

  BEGIN { o_2.xyz }
    Writeln ('Methode xyz in o_2');
    Write ('     o_2-Objekt: ');
    IF flag = TRUE
    THEN Writeln ('flag = TRUE , i = ', i)
    ELSE Writeln ('flag = FALSE, i = ', i)
  END   { o_2.xyz };

PROCEDURE o_2.meth;
  BEGIN { o_2.meth }
    Writeln ('Methode meth');
    Writeln;
    xyz   { Aufruf der Methode xyz (aus o_2) };
  END   { o_2.meth };

PROCEDURE o_2.del;
  BEGIN { o_2.del }
    ...
  END   { o_2.del };

{ Implementierung der Methoden von o_1: }

PROCEDURE o_1.init (f : BOOLEAN;
                    x : INTEGER;
                    y : INTEGER);
  BEGIN { o_1.init }
    flag := f;
    i    := x;
    IF flag
```

```
    THEN j := x
    ELSE j := x + y
  END   { o_1.init };

PROCEDURE o_1.erase;
  BEGIN { o_1.erase }
    ...
  END   { o_1.erase };

PROCEDURE o_1.xyz;
  BEGIN { o_1.xyz }
    Writeln ('Methode xyz in o_1');
    Writeln ('     o_1-Objekt: j = ', j);
    IF flag
    THEN Writeln( '                 flag = TRUE , i = ', i)
    ELSE Writeln( '                 flag = FALSE, i = ', i);
  END   { o_1.xyz };


VAR o_1_objekt_a : o_1;
    o_1_objekt_b : o_1;
    o_2_objekt_1 : o_2;
    o_2_objekt_2 : o_2;


BEGIN { oop_stat }
  o_1_objekt_a.init (TRUE, 1, 100);
  o_1_objekt_b.init (FALSE, 2, 200);
  o_2_objekt_1.init (TRUE, 1000);
  o_2_objekt_2.init (FALSE, 2000);

  o_1_objekt_a.xyz  { <== Aufruf der Methode xyz aus o_1 };
  Writeln;
  o_1_objekt_b.xyz  { <== Aufruf der Methode xyz aus o_1 };
  Writeln;
  o_1_objekt_b.meth { <== Aufruf der Methode meth aus o_2
                          und dort der Methode xyz aus o_2};

  Writeln;
  o_2_objekt_1.xyz  { <== Aufruf der Methode xyz aus o_2 };
  Writeln;
  o_2_objekt_2.meth { <== Aufruf der Methode meth aus o_2
                          und dort der Methode xyz aus o_2};
  Writeln('====================')
END   { oop_stat }.
```

Ergebnis:

```
Methode xyz in o_1
     o_1-Objekt: j = 1
                 flag = TRUE , i = 1

Methode xyz in o_1
     o_1-Objekt: j = 202
                 flag = FALSE, i = 2

Methode meth

Methode xyz in o_2
     o_2-Objekt:      flag = FALSE, i = 2

Methode xyz in o_2
     o_2-Objekt:      flag = TRUE , i = 1000
```

```
Methode meth

Methode xyz in o_2
     o_2-Objekt:      flag = FALSE, i = 2000
```

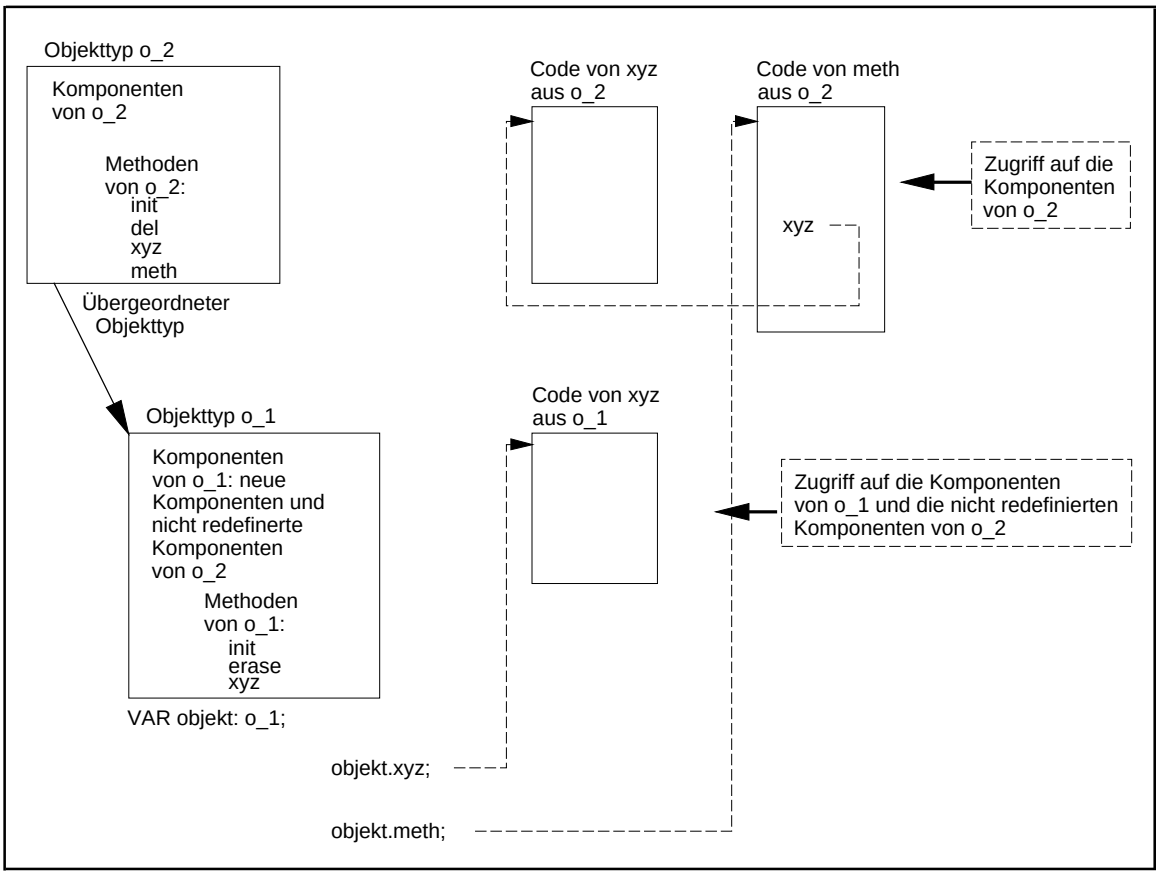


Abbildung 9.2-2: Statische Vererbung

Durch spezielle Mechanismen ist es jedoch möglich, eine Methode fest an ein Datenobjekt eines Objekttyps zu binden, der die Methode definiert. In obigem Beispiel heißt das, daß bei Anwendung der Methode `xyz` auf ein Datenobjekt vom Objekttyp `o_1` immer die Methode `o_1.xyz` genommen wird, bei Anwendung der Methode `xyz` auf ein Datenobjekt vom Objekttyp `o_2` immer die Methode `o_2.xyz` - und das auch, wenn `xyz` aus `meth` heraus aufgerufen wird. Dieses Konzept wird in Kapitel 9.1 als **Polymorphie** beschrieben: Eine Methode bekommt eine einzige Bezeichnung, die in der gesamten Objekttyphierarchie verwendet wird. Für jeden Objekttyp wird eine eigene Implementierung der Methode definiert. Wird die Methode mit einem Datenobjekt angestoßen, so wird zum Ablaufzeitpunkt dann jeweils diejenige Implementation der Methode genommen, die zu dem Objekttyp des Datenobjekts gehört. Die Bindung einer Methode an ein Datenobjekt erfolgt also erst zur Laufzeit und wird daher **späte Bindung** genannt.

Zur Realisierung der späten Bindung beinhaltet PASCAL das Konzept der **virtuellen Methoden**. Die Methoden eines Objekttyps, die an ein Datenobjekt dieses Objekttyps erst

zur Laufzeit angebunden werden, müssen als virtuell (Schlüsselwort **VIRTUAL**) gekennzeichnet werden. Der Compiler erzeugt für jeden Objekttyp, der virtuelle Methoden verwendet, eine eigene Tabelle mit den Adressen der Implementierungen der virtuellen Methoden, die **virtuelle Methodentabelle des Objekttyps**. Die virtuelle Methodentabelle eines Objekttyps liegt im Datensegment, ist aber der Anwendung nicht zugänglich. ***Jedes*** Datenobjekt (Instanz) dieses Objekttyps muß vor Verwendung einer ***virtuellen Methode initialisiert werden***, indem es eine für den Objekttyp spezifische Methode, die als **Konstruktor** (Schlüsselwort **CONSTRUCTOR** anstelle von PROCEDURE) bezeichnet wird, aktiviert. Dabei wird mit dem Datenobjekt ein Verweis auf die virtuelle Methodentabelle seines Objekttyps verbunden. Jedes einzelne Datenobjekt dieses Objekttyps muß zur Initialisierung den Konstruktor aufrufen, um die Verbindung mit der virtuellen Methodentabelle zu erhalten; eine einfache Wertzuweisung eines bereits initialisierten Datenobjekts auf ein noch nicht mit dem Konstruktor initialisierten Datenobjekts überträgt diese Verbindung nicht. Die Syntax eines Konstruktors unterscheidet sich bis auf das Schlüsselwort CONSTRUCTOR nicht von der einer Prozedur.

```
TYPE o_2 = OBJECT
             { Komponenten von o_2:}
             flag : BOOLEAN;
             i    : INTEGER;
             { Methoden von o_2: }
             CONSTRUCTOR init (wert : BOOLEAN;
                               x    : INTEGER);
             PROCEDURE del;
             PROCEDURE xyz; VIRTUAL;
             PROCEDURE meth;
           END;

     o_1 = OBJECT (o_2)
             { zusätzliche Komponenten von o_1:}
             j : INTEGER;
             { Methoden von o_1: }
             CONSTRUCTOR init (f : BOOLEAN;
                               x : INTEGER;
                               y : INTEGER);
             PROCEDURE erase;
             PROCEDURE xyz; VIRTUAL;
           END;
```

Entsprechend müssen auch die Prozedurköpfe bei den Implementierungen der Prozeduren o_1.init und o_2.init geändert werden:

```
CONSTRUCTOR o_2.init (wert : BOOLEAN;
                      x    : INTEGER);
```

und

```
CONSTRUCTOR o_1.init (f : BOOLEAN;
                      x : INTEGER;
                      y : INTEGER);
```

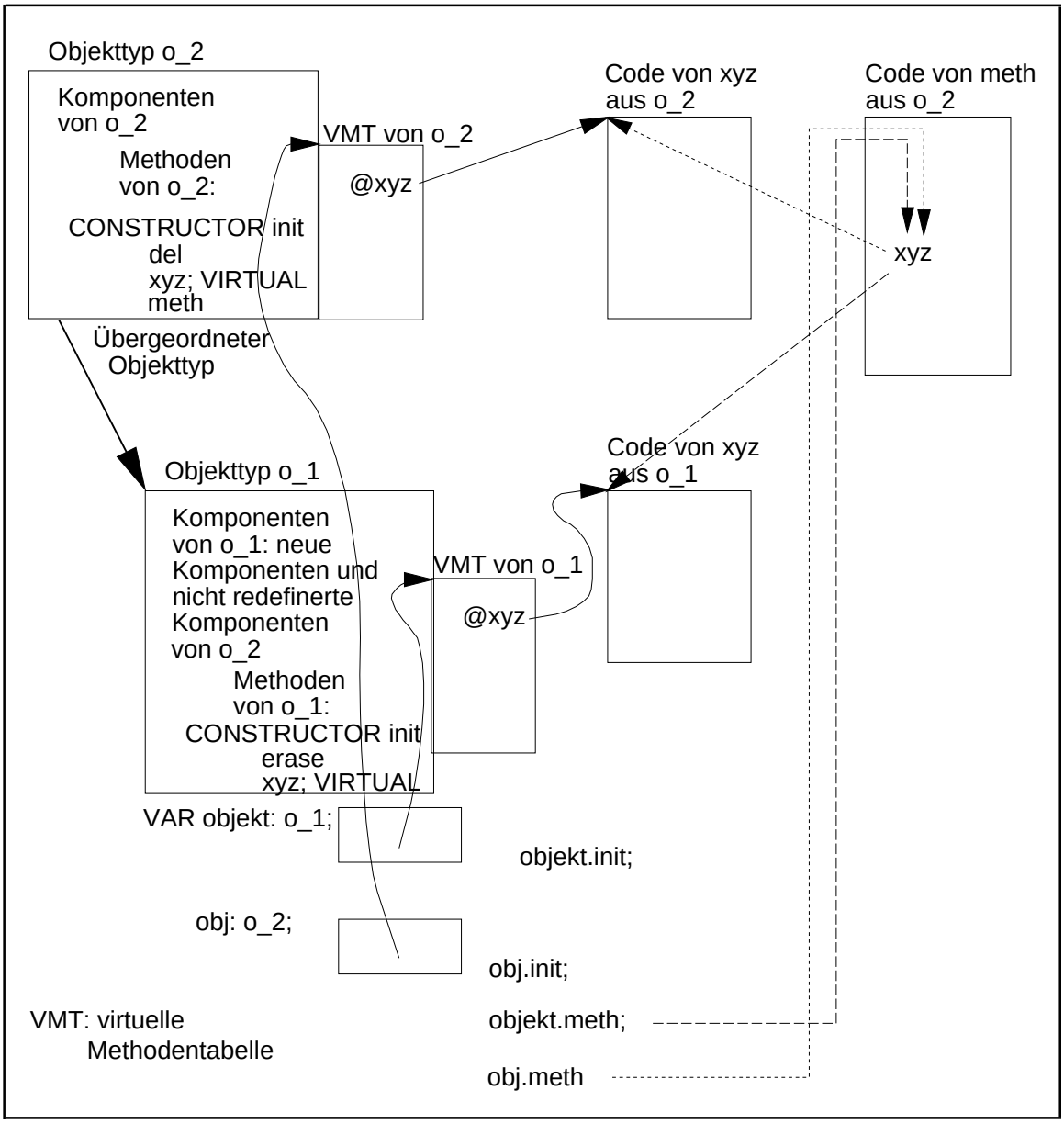


Abbildung 9.2-3: Virtuelle Methoden

Werden zwei Datenobjekt

```
VAR objekt: o_1;
    obj   : o_2;
```

deklariert und durch ihren Objekttyp-spezifischen Konstruktor initialisiert, z.B.:

```
objekt.init (TRUE, 1000, 2000);
obj.init (FALSE, 10);
```

so ruft die Methode `meth` für das Datenobjekt `objekt` die Methode `o_1.xyz` und für das Datenobjekt `obj` die Methode `o_2.xyz` auf (siehe Abbildung 9.2-3). Die Adresse der dem

Objekttyp entsprechenden Methode `xyz` kann ja nun, auch im Aufruf von `meth`, während der Laufzeit aus der virtuellen Methodentabelle des Objekttyps genommen werden, die über den Verweis im Datenobjekt erreichbar ist.

Zusätzlich soll jetzt die Methode `del` von `o_2` als virtuelle Methode vereinbart werden:

```
TYPE o_2 = OBJECT
             { Komponenten von o_2:}
             flag : BOOLEAN;
             i    : INTEGER;
             { Methoden von o_2: }
             CONSTRUCTOR init (wert : BOOLEAN;
                               x    : INTEGER);
             PROCEDURE del; VIRTUAL;
             PROCEDURE xyz; VIRTUAL;
             PROCEDURE meth;
           END;
```

Das prinzipielle Format der internen Darstellung der Datenobjekte `objekt` und `obj` und des Datenobjekts `o_1_objekt_a` mit Objekttyp `o_1` zeigt Abbildung 9.2-4. Insbesondere ist die Methode `del` auch an `o_1` vererbt worden, d.h. es gibt einen entsprechenden Eintrag in der virtuellen Methodentabelle von `o_1`. Die Adressen nicht-virtueller Methoden tauchen nicht in den virtuellen Methodentabellen auf.

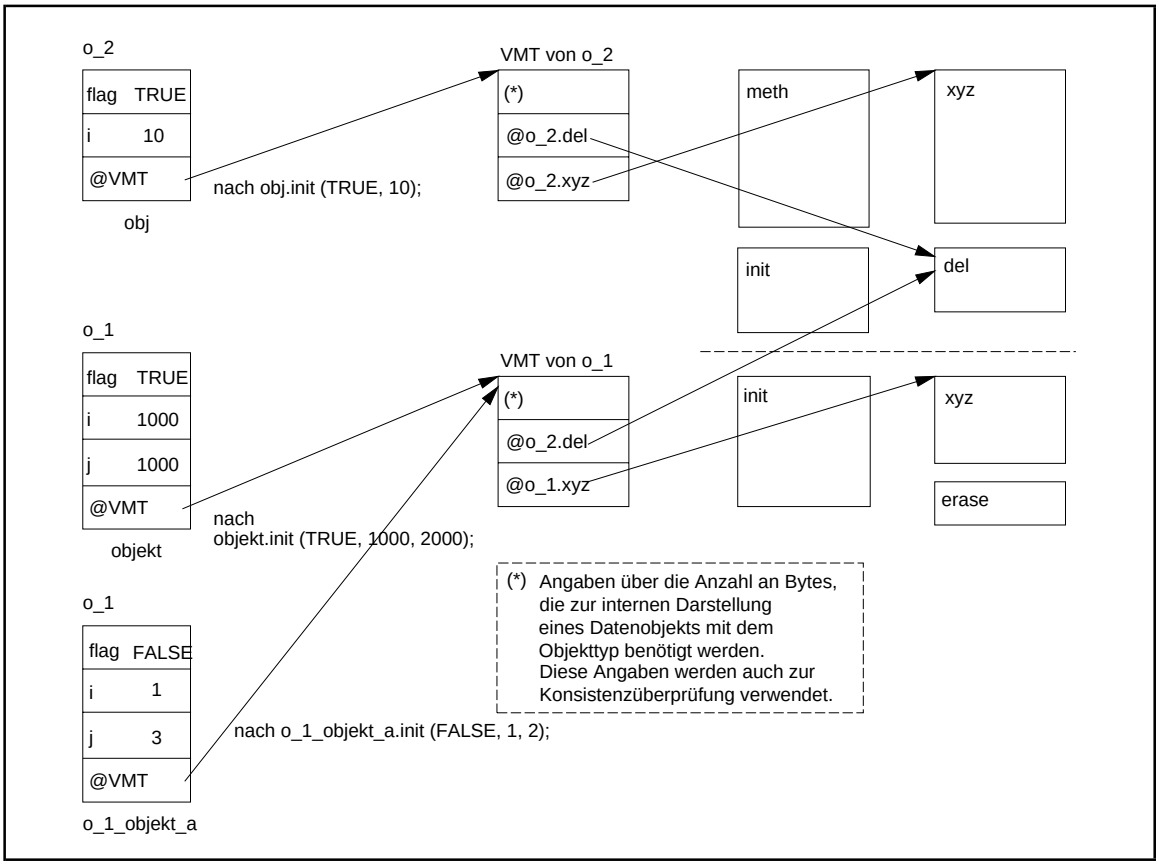


Abbildung 9.2-4: Interne Darstellung von Objekten mit virtuellen Methoden

Die Methodik kann verwendet werden, um eine einheitliche Methode zum Anzeigen der verschiedenen Blocktypen im einführenden Beispiel zu implementieren. Die Methode `show` wird in den übergeordneten Objekttyp `block` eingebaut und ruft die virtuelle Methode `display` auf, für die es für jeden dem Objekttyp `block` untergeordneten Objekttyp, `datenblock` und `indexblock`, eine eigene Implementierung gibt. Die Methode `display` selbst wird aus dem Objekttyp `block` entfernt. In `show` werden die allgemeinen Komponenten (`anz`, `vorg`, `last` und `next`) bearbeitet; für die speziellen Komponenten der nachfolgenden Objekttypen werden die jeweiligen Methoden aufgerufen. Die Änderungen der einzelnen Objekttypen ergeben:

```
 block = OBJECT
           anz  : INTEGER;
           vorg : block_ptr;
           last : block_ptr;
           next : block_ptr;
           { Methoden von block }
           CONSTRUCTOR init;
           PROCEDURE insert;
           PROCEDURE delete;
           PROCEDURE show
         END;

datenblock = OBJECT (block)
               data : data_sect_typ;
               { Methoden von datenblock }
               CONSTRUCTOR init;
               PROCEDURE insert_entry
                        (new_entry : data_entry_typ);
               PROCEDURE delete_entry
                        (del_entry : data_entry_typ);
               PROCEDURE display; VIRTUAL
             END;
indexblock = OBJECT (block)
               p_0   : block_ptr;
               index : index_sect_typ;
               { Methoden von indexblock }
               CONSTRUCTOR init;
               PROCEDURE insert_entry
                        (new_entry: index_entry_typ);
               PROCEDURE delete_entry
                        (del_entry: index_entry_typ);
               PROCEDURE display; VIRTUAL
             END;
```

mit der neuen Prozedur

```
PROCEDURE block.show;

  BEGIN { block.show }
    { Anzeigen der Komponenten anz, vorg, last und next
      des Kopfteils:                                   }
    ...;
    { Anzeigen der speziellen Komponenten der
      Datenobjekte, deren Objekttyp sich aus block
      herleitet:                                      }
    display
  END   { block.show };
```

Alle Komponenten der Datenblöcke

```
VAR d_block_1 : datenblock;
    i_block   : indexblock;
```

werden nach

```
  d_block_1.init;
  i_block.init;
```

durch

```
d_block_1.show;
i_block.show;
```

angezeigt.

Auch wenn ein Objekttyp keine virtuellen Methoden enthält, kann ein Konstruktor definiert werden. Sein Aufruf entspricht dann einem normalen Methodenaufruf. Sobald ein Objekttyp oder einer seiner Vorfahren in der Vererbungshierarchie virtuelle Methoden enthält, ***muß*** vor der ersten Verwendung eines Datenobjekts mit diesem Objekttyp der Konstruktor aufgerufen werden. Prinzipiell können auch mehrere Konstruktoren für einen Objekttyp deklariert werden.

In vielen Anwendungen belegen Datenobjekte aufgrund der großen Zahl ihrer Komponenten, einschließlich der aus übergeordneten Objektklassen eventuell über viele Stufen geerbten Komponenten, in ihrer internen Darstellung eine großen Speicherbereich. Daher ist es sinnvoll, derartige Datenobjekte erst während der Laufzeit als dynamische Datenobjekte einzurichten. Für den Umgang mit dynamischen Datenobjekten der OOP sind die Pascal-Standardprozeduren `New` und `Dispose` in ihrer Syntax und Semantik erweitert worden. Der folgende Programmausschnitt zeigt das Prinzip der Verwendung dynamischer Datenobjekte. Es wird eine Pointervariable deklariert und damit (als aktuellen Parameter) die Standardprozedur `New` aufgerufen. Diese legt das dynamische Datenobjekt (im Heap) an und gibt die Adresse des Datenobjekts als Wert der Pointervariablen zurück. Als weiterer Parameter der Prozedur `New` wird der Aufruf des Konstruktors für diesen Objekttyp mitgegeben, der automatisch von New nach Allokation des Speicherplatzes auf dem Heap für dieses neue Datenobjekt angestoßen wird und damit die Verbindung zur virtuellen Methodentabelle des Objekttyps herstellt. Selbstverständlich kann ein Konstruktoraufruf auch Aktualparameter haben.

```
TYPE obj_typ = OBJECT
                 { ... Komponenten des Objekttyps ...}
                 CONSTRUCTOR init (< Liste der Formalparameter > );
                 ...
                 PROCEDURE proc (< Liste der Formalparameter > } );
                 ...
                 DESTRUCTOR done; { siehe unten }
               END;

VAR obj_ptr : ^obj_typ;
...
BEGIN
  ...
  { Einrichten und initialisieren eines Objekts }
  New (obj_ptr, init (< Liste der Aktualparameter > ));
```

```
  ...
  { Aufruf der Methode proc für dieses Objekt }
  obj_ptr^.proc (< Liste der Aktualparameter > );
  ...
END;
```

Im anfänglichen Beispiel kann man eine Variable `root` mit Pointertyp definieren, deren Inhalt auf ein dynamisch erzeugtes Datenobjekt mit Objekttyp `block` zeigen soll:

```
VAR root : ^block;
```

Mit der Methode `block.init` als Konstruktor wird durch

```
New (root, init);
```

ein dynamisches Datenobjekt vom Objekttyp `block` auf dem Heap angelegt, `root` mit dessen Adresse versorgt und das Datenobjekt mit der Methode `init` initialisiert. Der Aufruf entspricht der Befehlsfolge

```
New (root);
root^.init;
```

Entsprechend ist die Pascal-Standardprozedur `Dispose` zur Freigabe des Speicherplatzes dynamisch erzeugter Variablen erweitert worden. In diesem Zusammenhang ist ein neuer Typ von Methoden wichtig, der **Destruktor**. Dabei handelt es sich um eine Methode innerhalb einer Objekttypdefinition (Schlüsselwort **`DESTRUCTOR`** anstelle von `PROCEDURE`). Ein Destruktor vereinigt die Freigabe des Speicherplatzes eines dynamisch erzeugten Datenobjekts mit dem entsprechenden Objekttyp auf dem Heap mit einer vom Benutzer definierten Methode, die unmittelbar vor der Speicherplatzfreigabe noch ausgeführt wird. Wird der Objekttyp block beispielsweise um einen Destruktor mit Namen `done` erweitert:

```
     block = OBJECT
               anz  : INTEGER;
               vorg : block_ptr;
               last : block_ptr;
               next : block_ptr;
               { Methoden von block }
               CONSTRUCTOR init;
               DESTRUCTOR done;
               PROCEDURE insert;
               PROCEDURE delete;
               PROCEDURE show
             END;

DESTRUCTOR block.done;
BEGIN
  ...
END;
```

so wird der Speicherplatz, auf den die Variable `root` nach dem Aufruf

```
New (root, init);
```

zeigt, durch

```
Dispose (root, done);
```

wieder freigegeben, nachdem die Anweisungen der Methode `done` vorher ausgeführt wurden.

Eine interessante Variante des Verfahrens ergibt sich unter Berücksichtigung erweiterter Typkompatibilitätsregeln. Die Objekttypdefinitionen werden zusätzlich um virtuelle Destruktoren jeweils mit Namen `done` erweitert:

```
DESTRUCTOR done; VIRTUAL;
```

und in jeden Objekttyp die Methoden `done` eingefügt:

```
DESTRUCTOR block.done;
BEGIN
 ...
END;

DESTRUCTOR datenblock.done;
BEGIN
 ...
END;

DESTRUCTOR indexblock.done;
BEGIN
 ...
END;
```

Außerdem seien folgende Datentypen und Variablen deklariert:

```
VAR bl_ptr : ^block;
    da_ptr : ^datenblock;
    in_ptr : ^indexblock;
```

Dann ist nach Ausführung der `New`-Prozedur mit den Initialisierungsmethoden:

```
New (da_ptr, init);
New (in_ptr, init);
```

jeweils ein Datenobjekt mit Objekttyp `datenblock` bzw. `indexblock` auf dem Heap eingerichtet und initialisiert und dessen Adresse in `da_ptr` bzw. `in_ptr` abgelegt worden. Die erweiterten Typkompatibilitätsregeln erlauben eine Zuweisung des Werts der Pointervariablen `da_ptr` bzw. `in_ptr` an `bl_ptr`. Zu beachten ist, daß bei dieser Wertzuweisung der Inhalt von `da_ptr` bzw. `in_ptr`, also jeweils eine Adresse und nicht der Wert des adressierten Datenobjekts, übertragen wird. Die Zuweisung ist sinnvoll, weil ja alle Komponenten, die im Objekttyp `block` angesprochen werden, auch in `datenblock` und `indexblock` vorkommen. Durch

```
bl_ptr := da_ptr;
Dispose (bl_ptr, done);
bl_ptr := in_ptr;
Dispose (bl_ptr,done);
```

wird zunächst der Destruktor `datenblock.done` aktiviert und der Speicherplatz für das Datenobjekt vom Objekttyp `datenblock` auf dem Heap freigegeben; dann wird der Destruktor `indexblock.done` angestoßen und der Speicherplatz des Datenobjekts vom Objekttyp `indexblock` auf dem Heap freigegeben. Es können unterschiedliche, Objekttyp-spezifische Methoden `done` ausgeführt werden, da diese jeweils als virtuelle Methoden definiert wurden. Aus der virtuellen Methodentabelle, die zu jedem Objekttyp gehört, wird zudem erkannt, wieviele Bytes durch `Dispose` jeweils auf dem Heap freizugeben sind.

Die virtuelle Methodentabelle eines Objekttyps enthält für jede Methode einen Adreßverweis auf den Code der virtuellen Methode. Wenn ein Objekttyp eine große Anzahl virtueller Methoden besitzt bzw. ererbt, wird dadurch viel Speicherplatz belegt. Auch wenn der Objekttyp nur wenige der ererbten virtuellen Methoden überschreibt, enthält seine virtuelle Methodentabelle Verweise auf alle Methoden, ob sie nun überschrieben wurden oder nicht. Durch die Verwendung **dynamischer Methoden** kann in dieser Situation der erforderliche Speicherplatz verkleinert werden. Die Syntax einer dynamischen Methode entspricht weitgehend einer virtuellen Methode; sie schreibt in der Deklaration neben dem Schlüsselwort `VIRTUAL` einen zusätzlichen numerischen **dynamischen Methodenindex** vor:

```
PROCEDURE methodenname (<parameter list>);
             VIRTUAL dynamischer_Methodenindex;
```

`dynamischer_Methodenindex` ist ein `INTEGER`-Wert, eine `INTEGER`-Konstante oder ein Ausdruck, der zu einem `INTEGER`-Wert ausgewertet wird (zwischen 1 und 65535). Überschreibt ein Nachkomme in der Vererbungshierarchie eine dynamische Methode, muß die Deklaration der überschreibenden Methode in Bezug auf Reihenfolge, Typ und Bezeichner der Parameter der Deklaration der überschriebenen Methode entsprechen. Auch sie muß das Schlüsselwort `VIRTUAL` angeben, dem der gleiche dynamische Methodenindexwert folgt, der in der Deklaration der überschriebenen Methode steht.

Bei der Verwendung dynamischer Methoden wird für einen Objekttyp neben seiner virtuellen Methodentabelle eine **dynamische Methodentabelle** generiert, die im wesentlichen Adreßverweise auf die Methoden enthält, die auch wirklich überschrieben wurden und auf die dynamische Methodentabelle des direkten Vorfahren. Das folgende Beispiel aus [BP7] erläutert das Prinzip. Der Objekttyp `TBase` vererbt Methoden an einen Objekttyp `TDerived`, der einige dieser Methoden überschreibt (`init`, `done`, `p10`, `p30`) und eine zusätzliche Methode (`p50`) deklariert:

```
TYPE TBase = OBJECT
               x : INTEGER;
               CONSTRUCTOR init;
               DESTRUCTOR done; VIRTUAL;
               PROCEDURE p10; VIRTUAL 10;
               PROCEDURE p20; VIRTUAL 20;
               PROCEDURE p30; VIRTUAL 30;
               PROCEDURE p40; VIRTUAL 40;
             END;
```

```
TDerived = OBJECT (TBase)
             y : INTEGER;
             CONSTRUCTOR init;
             DESTRUCTOR done; VIRTUAL;
             PROCEDURE p10; VIRTUAL 10;
             PROCEDURE p30; VIRTUAL 30;
             PROCEDURE p50; VIRTUAL 50;
           END;
```

Abbildung 9.2-5 zeigt im oberen Teil das Layout der virtuellen und dynamischen Methodentabellen der beiden Objekttypen. Im unteren Teil ist zum Vergleich das entsprechende Layout zu sehen, wenn anstelle dynamischer Methoden virtuelle Methoden verwendet, d.h. die dynamischen Methodenindizes weggelassen werden.

Der Aufruf einer dynamischen Methode ist zeitaufwendiger als der Aufruf einer virtuellen Methode: Die dynamischen Methodentabellen sind analog zur Vererbungshierarchie über Adreßverweise miteinander verbunden. Eventuell ist die aufgerufene Methode über mehrere Hierarchiestufen geerbt, und ihre Adresse ist nur in der dortigen dynamischen Methodentabelle verzeichnet und nicht in der virtuellen Methodentabelle, die zum Objekttyp des aufgerufenen Objekts gehört. Um die Adresse der Methode zu finden, müssen u.U. die dynamischen Methodentabellen über mehrere Hierarchiestufen durchsucht werden. Durch das in Abbildung 9.2-5 gezeigte Zwischenspeicherfeld in einer dynamischen Methodentabelle wird dieser Suchvorgang beschleunigt (Details siehe in [BP7]).

Offensichtlich lohnt sich der Einsatz dynamischer anstelle virtueller Methoden nur, wenn, wenn sehr viele virtuelle Methoden von Vorfahren geerbt werden, die vom Objekttyp selbst überschrieben werden.

Eine Besonderheit im Zusammenhang mit dem Aufruf einer Methode für ein Objekt stellt ein impliziter Parameter des Methodenaufrufs dar, der mit dem Schlüsselwort **`Self`** bezeichnet wird: Innerhalb des Codes einer Methode bezeichnet `Self` das Objekt (genauer: die Instanz), über das die Methode gerade aufgrufen wird. Mit dem Bezeichner `Self` kann man daher das beteiligte Objekt innerhalb des Methodencodes benennen. In Kapitel 11 wird von dieser Programmiertechnik Gebrauch gemacht.

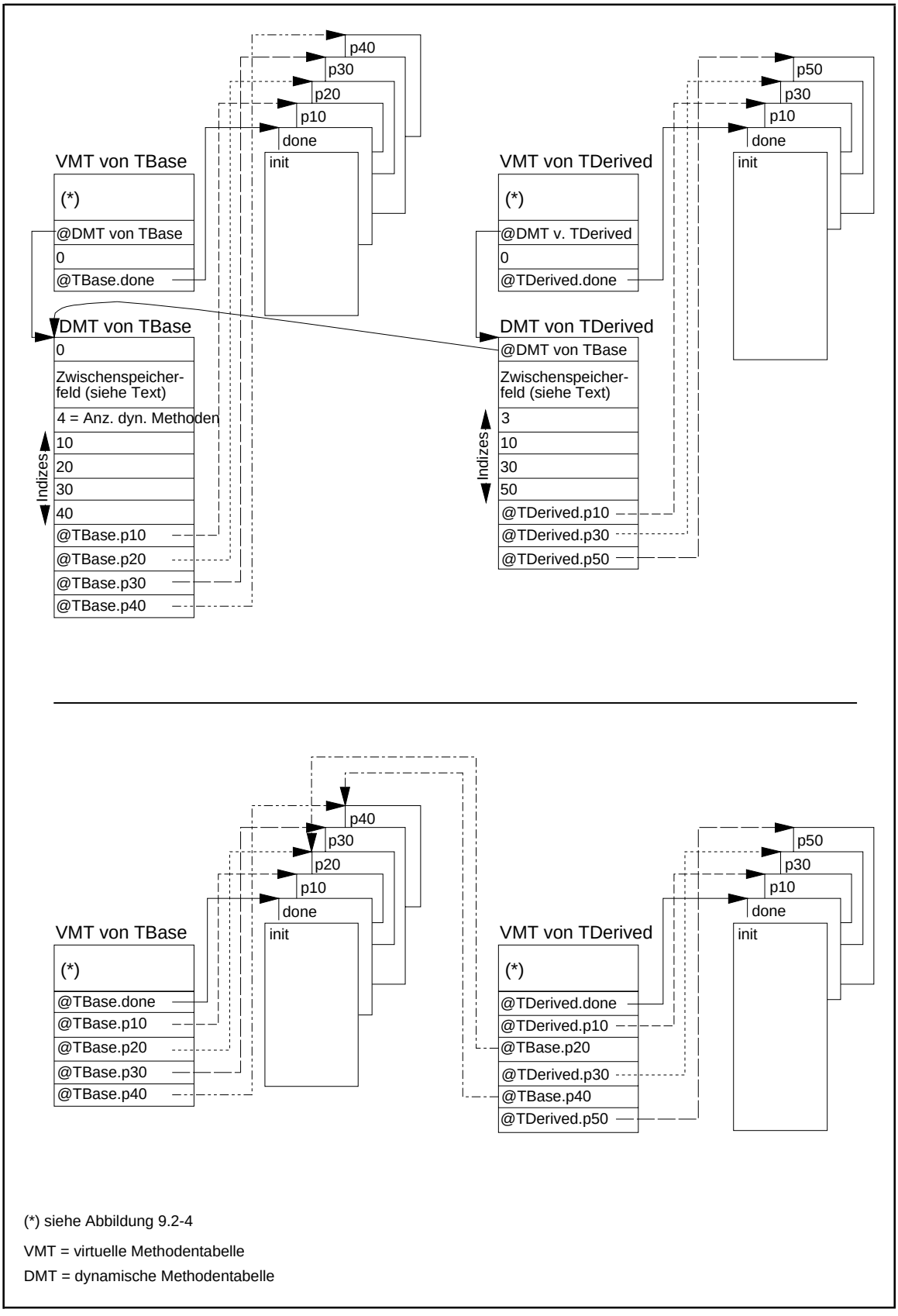


Abbildung 9.2-5: Dynamische Methodentabelle

9.3 Weiterführende Konzepte in Object Pascal in Hinblick auf die OOP

Als Weiterentwicklung des Pascal-Dialekts Borland Pascal enthält die Sprache Object Pascal des Entwicklungssystems Delphi ([WAR]) *als Sprachkonzepte* alle in Kapitel 9.1 erwähnten Konzepte der OOP (mit Ausnahme der multiplen Vererbung und der strengen Kapselung von Objekten). Die folgende Aufstellung gibt einen kleinen Überblick über wesentliche Aspekte von Object Pascal, die die Konzepte der OOP in PASCAL erweitern.

- Neben dem Objektmodell, wie es PASCAL durch das syntaktische Konstrukt `OBJECT` vorsieht, gibt es ein erweitertes Objektmodell, dessen Objektklassen (Objekttypen) mit dem Schlüsselwort **`CLASS`** anstelle des Schlüsselworts `OBJECT` deklariert werden. Die Möglichkeit der Deklaration eines Objekttyps mit Hilfe des Schlüsselworts `OBJECT` bleibt aus Kompatibilitätsgründen erhalten.

- Als "Urahn" aller Objektklassen vom `CLASS`-Typ gibt es eine vordefinierte Klasse mit Objekttyp `TObject`, für die einfache Basismethoden (z.B. Konstruktor und Destruktor) deklariert sind.

- Es gibt die Möglichkeit, Operationen auf ganzen Objektklassen zu erklären, d.h. es können **Metaklassen** mit entsprechenden Methoden, die dann `CLASS PROCEDURE`s bzw. `CLASS FUNCTION`s heißen, vereinbart werden.

- Objekte bestehen, wie in Kapitel 9.1 beschrieben, aus Daten (Komponenten), Eigenschaften (**properties**) und Methoden. Für die Komponenten sind verfeinerte Sichtbarkeitsregeln durch entsprechende Schlüsselwörter (**`PUBLISHED`**, **`PUBLIC`**, **`PROTECTED`**, **`PRIVATE`**) vereinbart.

- Neben den Attributen statisch, virtuell und dynamisch wie in PASCAL kann für eine virtuelle oder dynamische Methode das Attribut **`ABSTRACT`** angegeben werden. Eine so als **abstrakte Methode** gekennzeichnete Prozedur besteht nur aus dem Methodenkopf; die Implementierung erfolgt in einer Nachkommen-Klassen.

- Grundsätzlich werden Objekte im Heap angelegt. Eine Variable mit Objekttyp ist in Wirklichkeit immer eine Pointervariable auf ein dynamisches Datenobjekt (im Heap). Vor der ersten Verwendung eines Objekts muß also immer ein Konstruktor aktiviert werden, der automatisch die einzelnen Komponenten eines Objekts mit Defaultwerten initialisiert.

- Das Konzept von Botschaften einschließlich der Botschaftsbehandlungsmechanismen ist als Sprachkonzept in Object Pascal enthalten.

- Das Konzept der **Ausnahmebehandlung (exceptions)**, auch wenn es kein Charakteristikum der OOP ist, erlaubt die Definition von Ausnahmesituationen, die während der Laufzeit überprüft werden und auf die dann kontrollierte Reaktionen ermöglicht werden.

10 Anwendungsorientierte Datenstrukturen in der systemnahen Programmierung

Das Thema dieses Kapitels sind **Datenstrukturen mit den dazugehörigen Operationen**, die in vielen Anwendungen häufig vorkommen. Sie werden in Form von PASCAL-Deklarationen und -Prozeduren formuliert, da dadurch das Verständnis der Datenstrukturen erleichtert und der Anwendungsbezug deutlicher unterstrichen werden. Es gibt heute für viele Programmiersprachen mehr oder weniger umfangreiche Klassenbibliotheken, die vordefinierte Datenstrukturen bereitstellen. Beispiele sind die Objekttypen in Turbo Vision ([BP7]) und die Visuellen Komponenten in Delphi ([WAR]). In den folgenden Unterkapiteln soll eine Auswahl derartiger Datenstrukturen beschrieben werden, wobei der Schwerpunkt auf der Behandlung allgemeiner Basisstrukturen zur Manipulation von Klassen, bestehend aus einzelnen Datenobjekten, liegt und nicht etwa auf der Behandlung von Strukturen zur Gestaltung von Benutzeroberflächen und Dialoganwendungen.

Die Basis bildet eine Klasse von Objekten, deren innere Struktur hier nicht weiter betrachtet werden soll und die daher **Elemente** genannt werden. Die Objekttypdeklaration eines Elements lautet

```
TYPE Tentry = OBJECT
                ...
              END;
```

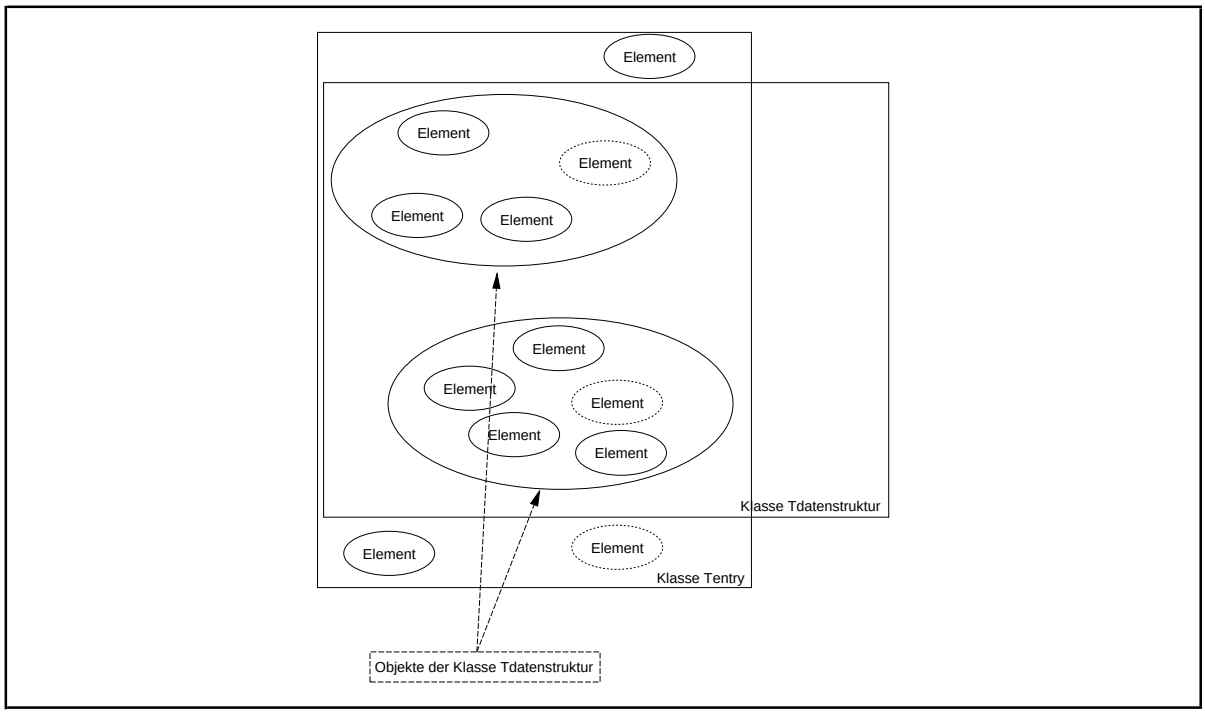


Abbildung 10-1: Elemente und Strukturen

Die Komponenten, die ein Objekt vom Objekttyp `Tentry` besitzt, und die Methoden, die auf ein derartiges Objekt anwendbar sind, sind zunächst irrelevant. Elemente können zu größeren Strukturen zusammengefaßt werden, auf denen bestimmte Operationen definiert sind, die für die Struktur charakteristisch sind. Eine derartige Struktur hat einen Objekttyp `Tdaten-struktur`, dessen Methoden den Zustand eines Objekts dieser Klasse, d.h. einer Zusammenfassung von Elementen, in der Regel beeinflußt, nicht aber den Zustand eines Elements in dieser Struktur (Abbildung 10-1).

Abbildung 10-2 führt tabellarisch einige Beispiele für Elemente und Strukturen auf und benennt darauf definierte Methoden (Operationen). Die ***Methoden*** können ***in drei Typen eingeteilt*** werden:

- Methoden, die den Zustand eines Objekts (des jeweiligen Objekttyps `Tdatenstruktur`) ändern, indem sie Elemente hinzufügen oder entfernen und damit den Zustand eines Objekts ändern

- Methoden, die den Zustand eines Objekts (des jeweiligen Objekttyps `Tdatenstruktur`) anzeigen und den Zustand unverändert lassen

- Methoden, die zwei oder mehr Objekte (des jeweiligen Objekttyps `Tdatenstruktur`) zu neuen Objekten verknüpfen bzw. aus einem Objekt neue Objekte generieren und dabei eventuell den Zustand des Ausgangsobjekts ändern.

Beispiele für den ersten Typ sind die Methoden ***insert*** und ***delete*** im Objekttyp `TListe`, für den zweiten Typ die Methoden ***is_member*** im Objekttyp `TListe` und ***min*** im Objekttyp `Tprio` und für den dritten Typ die Mengenoperationen $\cup$ und $\cap$ im Objekttyp `TMenge`.

Aus Anwendersicht ist die Implementation der Objekte in der Klasse Tdatenstruktur ohne Belang. Die Möglichkeiten der Manipulationen (Operationen auf) der gesamten Struktur sind von Interesse. Natürlich kann eine schlechte Implementation zu einem unhandlichen und ineffizienten Verhalten führen. Daher ist es interessant zu wissen, wie "aufwendig" die Durchführung der einzelnen Methoden ist. Ausgehend von einem (als "leer") initialisierten Objekt vom Typ `Tdatenstruktur` kann man fragen, wieviele Verarbeitungsschritte eine Folge von n hintereinander erfolgende Methodenaufrufe einer Implementation benötigt. Je nach Datenstruktur und Anwendung kann auch die Frage von Interesse sein, wieviele Verarbeitungsschritte ein einziger Methodenaufruf benötigt, wenn die Datenstruktur bereits n Elemente enthält. Die Ergebnisse diese Komplexitätsuntersuchungen hängen im wesentlichen vom Parameter n ab, und nur die Größenordnung dieser funktionalen Abhängigkeit soll dargestellt werden (und nicht etwa die Laufzeit der Methoden auf einem speziellen Rechner). Dazu wird wieder die ***O*-Notation** verwendet. Zur Erinnerung:

Eine Funktion $T(n)$ ist von der **Ordnung** $O(f(n))$, geschrieben $T(n) = O(f(n))$, wenn gilt:
Es gibt eine (nicht von n abhängige) Konstante $C > 0$ und eine natürliche Zahl n_0 mit
$|T(n)| \leq C|f(n)|$ für jedes $n \geq n_0$.

In den folgenden Unterkapiteln werden Implementationen vorgestellt, wobei eine Auswahl aus den Beispielen in Abbildung 10-2 getroffen wird.

Bezeichnung der Datenstruktur: ***Liste***, Objekttyp `TListe`
Bezeichnung eines Elements: Eintrag
charakteristische Operationen:

- ***init***: Initialisieren als leere Liste
- ***insert***: Einfügen eines Eintrags
- ***delete***: Entfernen eines Eintrags
- ***is_member***: Feststellen, ob ein spezifizierter Eintrag in der Liste vorkommt
- ∀**op** Ausführung der Operation **op** auf allen Elementen, die sich in der Liste befinden

Bemerkung: Die Datenstruktur Liste entspricht einer häufig auch als **Kollektion** bezeichneten Datenstruktur.

Bezeichnung der Datenstruktur: ***(sortierbare) Liste***, Objekttyp `TSortiertListe`
Bezeichnung eines Elements: Eintrag mit Sortierkriterium
charakteristische Operationen: wie Liste, zusätzlich

- ***sort***: Sortieren der Liste

Bezeichnung der Datenstruktur: ***FIFO-Warteschlange***, Objekttyp `TFIFO`
Bezeichnung eines Elements: Eintrag
charakteristische Operationen:

- ***init***: Initialisieren als leere FIFO-Warteschlange
- ***insert***: Einfügen eines Eintrags
- ***delete***: Entfernen des Eintrags, der sich am längsten unter allen Einträgen in der FIFO-Warteschlange befindet, und Rückgabe an den Aufrufer

Bezeichnung der Datenstruktur: ***Stack (LIFO-Warteschlange)***, Objekttyp `TStack`
Bezeichnung eines Elements: Eintrag
charakteristische Operationen:

- ***init***: Initialisieren als leeren Stacks
- ***insert***: Einfügen eines Eintrags
- ***delete***: Entfernen des zuletzt eingefügten Eintrags unter allen Einträgen, die sich gerade im Stack befinden, und Rückgabe an den Aufrufer

Bezeichnung der Datenstruktur: ***beschränkter Puffer***, Objekttyp `TPuffer`
Bezeichnung eines Elements: Eintrag
charakteristische Operationen:

- ***init***: Initialisieren als leere Datenstruktur mit einer festen Anzahl an Einträgen
- ***insert***: Einfügen eines Eintrags auf die nächste freie Position; sind bereits alle Positionen belegt, wird ein bisheriger Eintrag überschrieben
- ***delete***: Entfernen des Eintrags, der sich am längsten unter allen Einträgen im beschränkten Puffer befindet, und Rückgabe an den Aufrufer

Bezeichnung der Datenstruktur: ***Prioritätsschlange***, Objekttyp `TPrio`
Bezeichnung eines Elements: Eintrag mit Sortierkriterium
charakteristische Operationen:

- ***init***: Initialisieren als leere Prioritätsschlange
- ***insert***: Einfügen eines neuen Eintrags
- ***delete***: Entfernen eines spezifizierten Eintrags
- ***is_member***: Feststellen, ob ein spezifizierter Eintrag in der Prioritätsschlange vorkommt
- ***min***: Ermitteln des kleinsten Eintrags, der sich gerade in der Prioritätsschlange befindet

Bezeichnung der Datenstruktur: ***Menge***, Objekttyp `TMenge`
Bezeichnung eines Elements: Element
charakteristische Operationen:

- ***init***: Initialisieren als leere Menge
- Mengenoperationen: $\cup, \cap$
- $\in$ Feststellen, ob ein spezifiziertes Element in der Menge enthalten ist
- $\subseteq$ Feststellen, ob eine Menge in einer anderen Menge enthalten ist

Bezeichnung der Datenstruktur: ***(Datenbank-) Tabelle***, Objekttyp `TDBtable`
Bezeichnung eines Elements: Datensatz
charakteristische Operationen: wie sortierbare Liste, zusätzlich

- ***select***: Auswahl von Datensätzen nach spezifizierten Selektionskriterien
- ***join***: Verbinden von Tabellen gemäß spezifizierten Selektionskriterien
- ***union***: Vereinigung von Tabellen

Bezeichnung der Datenstruktur: ***(gerichteter oder ungerichteter oder gewichteter) Graph, Spezialisierung auf Bäume bzw. Binärbäume bzw. höhenbalancierte Bäume***, Objekttyp `TGraph`
Bezeichnung eines Elements: Knoten, Kante (Knotenpaar)
charakteristische Operationen:

- ***init***: Initialisieren als leerer Graph
- ***insert_node***: Einfügen eines neuen Knotens
- ***insert_edge***: Einfügen einer neuen Kante
- ***delete_node***: Entfernen eines spezifizierten Knotens und der mit ihm inzidierenden Kanten
- ***delete_edge***: Entfernen einer spezifizierten Kante
- ***b_search***: Durchlaufen des Graphen, beginnend bei einem spezifizierten Knoten, gemäß Breitensuche und Ermittlung der Kantenreihenfolge
- ***d_search***: Durchlaufen des Graphen, beginnend bei einem spezifizierten Knoten, gemäß Tiefensuche und Ermittlung der Kantenreihenfolge
- ***min***: Ermitteln des Wegs mit minimalem Gewicht zwischen zwei Knoten
- ***min_all***: Ermitteln aller kürzesten Wege zwischen je zwei Knoten

Bezeichnung der Datenstruktur: ***Graphik***, Objekttyp `TGraphic`
Bezeichnung eines Elements: Graphikelement (mit Attributen)
charakteristische Operationen:

- ***init***: Initialisieren als leere Graphik
- ***insert***: Einfügen eines neuen Graphikelements an eine spezifizierte Position innerhalb der Graphik
- ***delete***: Entfernen eines spezifizierten Graphikelements
- ***mark***: Markieren von Graphikelementen
- ***size***: Veränderung der Größe von Graphikelementen innerhalb der Graphik
- ***move***: Bewegen von Graphikelementen an eine andere Position innerhalb der Graphik
- ***rotate***: Drehen von Graphikelementen innerhalb der Graphik um einen spezifizierten Drehwinkel
- ***change***: Ändern von Attributwerten von spezifizierten Graphikelementen

Bezeichnung der Datenstruktur: ***Matrix***, Objekttyp `TMatrix`
Bezeichnung eines Elements: Eintrag
charakteristische Operationen:

- ***init***: Initialisieren mit einem Anfangswert
- ***trans***: Transponieren
- ***invert***: Invertieren
- ***determ***: Ermittlung der Determinante
- ***eigen***: Ermittlung des Eigenwerts
- ***arithm***: Diverse arithmetische Matrixoperationen

Abbildung 10-2: Beispiele für Elemente und Strukturen

Zur Konkretisierung wird die Gesamtaufgabe in einzelne Units eingeteilt. Die Anwendung ist in Form eines Programms definiert, in der die oben beschriebenen Elemente vorkommen. Die Deklaration des Objekttyps eines Elements steht in der

```
UNIT element;

INTERFACE

   { Objekttypdeklarationen (Bezeichner, Methoden) eines Elements: }
   TYPE Pentry = ^Tentry; { Pointer auf ein Element }
        Tentry = OBJECT
                   ...
                   CONSTRUCTOR init ( ... );
                     ...
                   PROCEDURE display; VIRTUAL;
                     ...
                   DESTRUCTOR  done; VIRTUAL;
                 END;


IMPLEMENTATION

CONSTRUCTOR Tentry.init ( ... );
  BEGIN { Tentry.init }
    ...
  END   { Tentry.init };

...
```

```
PROCEDURE Tentry.display;
  BEGIN { Tentry.display }
    ...
  END   { Tentry.display };

...

DESTRUCTOR Tentry.done;

  BEGIN { Tentry.done }
    ...
  END   { Tentry.done };


END.
```

Neben dem Objekttyp eines Elements wird ein Pointertyp auf ein Element deklariert, da dieser Pointertyp in den Anwendungen und Implementierungen der Datenstrukturen häufig benötigt wird. Als Methode des Objekttyps `Tentry` ist neben dem Konstruktor `init` (eine Formalparameterliste ist angedeutet) und dem Destruktor `done`[32] eine Methode `display` deklariert, die den Zustand eines Objekts, d.h. die Werte seiner Komponenten, anzeigt.

Die Deklaration (Bezeichner, Operationen usw.) und die Implementierung der Datenstruktur, zu der die Elemente zusammengefaßt werden, befindet sich ebenfalls in einer Unit, die im Prinzip folgendes Layout hat. Zu beachten ist, daß die Implementierung der Details nach außen verborgen wird:

```
UNIT struktur;
      { Deklaration der gesamten Struktur,
        zu der Elemente zusammengefaßt werden }

INTERFACE

  USES element;

   { Objekttypdeklarationen (Bezeichner, Methoden) der Struktur: }
   TYPE Pdatenstruktur = ^Tdatenstruktur;
        Tdatenstruktur = OBJECT
                           ...
                         END;

IMPLEMENTATION

  { Implementierung der Methoden }
  ...

END.
```

Das Anwendungsprogramm bindet dann diese Units ein:

[32] Der Destruktor ist als virtuelle Methode definiert, da dies die Implementation eines Destruktors für abgeleitete Objekttypen erleichtert.

```
PROGRAM anwender;

  USES element,
       struktur;

  ...

BEGIN { anwender }
  ...
END   { anwender }.
```

10.1 Lineare Datenstrukturen

Unter einer **linearen Datenstruktur** soll eine Datenstruktur verstanden werden, auf der die Operationen

- ***init***: Initialisieren der Datenstruktur als leer

- ***insert***: Einfügen eines weiteren Elements in die Datenstruktur

- ***delete***: Entfernen eines Elements aus der Datenstruktur, wobei definiert ist, ob ein genau bezeichnetes oder ein beliebiges Element entfernt wird; das zu entfernende Element wird je nach Definition der Operation an den Aufrufer übergeben

- ***op***: Eventuell ***weitere Operationen, die nicht voraussetzen, daß zwischen den Elementen der Datenstruktur irgendwelche Ordnungsbeziehungen bestehen***

definiert sind. Beispiele sind die in Abbildung 10-2 genannten Datenstrukturen Liste, FIFO-Warteschlange, Stack und Menge. Es sei darauf hingewiesen, daß der Begriff "lineare Datenstruktur" in der Literatur unterschiedlich definiert oder mit dem Begriff "Liste" synonym verwendet wird. Der Begriff assoziiert natürlich eine gängige Realisierungsform, nämlich als tabellarische oder (über Adreßverweise) verkettete Liste.

Einige Operationen verändern den gegenwärtigen Zustand eines Objekts der Datenstruktur. Um die korrekte Implementierung zu kontrollieren, wird eine zusätzliche Operation ***show*** definiert, die den gegenwärtigen Zustand eines Objekts der Datenstruktur anzeigt. Die Operation show beeinflußt den Zustand eines Objekts nicht.

10.1.1 Liste

Für die Datenstruktur **Liste** sind neben den Operationen ***init***, ***insert***, ***delete*** und ***show*** die Operationen ***is_member*** und ∀**op** (siehe Abbildung 10-2) definiert und erfordern die Deklaration entsprechender Methoden für den Objekttyp `TListe`.

Da keine Beziehungen zwischen den Einträge einer Liste bestehen, ist es angebracht, die Datenstruktur Liste als verkettete Folge von dynamisch angelegten Einträgen zu implementieren. Der Objekttyp eines Einzeleintrags sieht jedoch keine Verkettung von Einträgen vor, so daß im Objekttyp `TListe` eine Adreßverkettung aufgebaut werden muß. Des weiteren müßte vom Konzept her bei der Aufnahme eines Eintrags in eine

Liste (genauer: in ein Datenobjekt der Klasse `TListe`) eine Kopie des Eintrags erzeugt werden; denn ein Objekt der Klasse `TListe` *enthält* ja selbst alle Einträge, die zu diesem Objekt gehören. Abbildung 10.1.1-1 zeigt im oberen Teil eine mögliche Implementierung für ein Datenobjekt vom Typ `TListe` mit zur Zeit 5 Einträgen. Die Anzahl der Einträge wird (für Erweiterungen durch zusätzliche Methoden) ebenfalls vermerkt. Im unteren Teil von Abbildung 10.1.1-1 ist ein etwas abgewandeltes Layout zu sehen. Es erscheint nämlich von der Implementierung her nicht angebracht, bei der Aufnahme eines neuen Eintrags in eine Liste ein Kopie des Eintrags zu erzeugen, weil dadurch eventuell viel Speicherplatz benötigt wird. Daher wird im Objekttyp `TListe` nur die Anzahl der Einträge in einer Liste und eine verkettete Folge von Adreßverweisen auf die Einträge in der Liste implementiert; die Einträge selbst liegen nicht im Datenobjekt vom Typ `TListe`. Das Erzeugen und Vernichten eines Datenobjekt vom Typ `Tentry` liegt in der Verantwortung des Anwenders und wird vom Objekttyp `TListe` und seinen Methoden nicht kontrolliert.

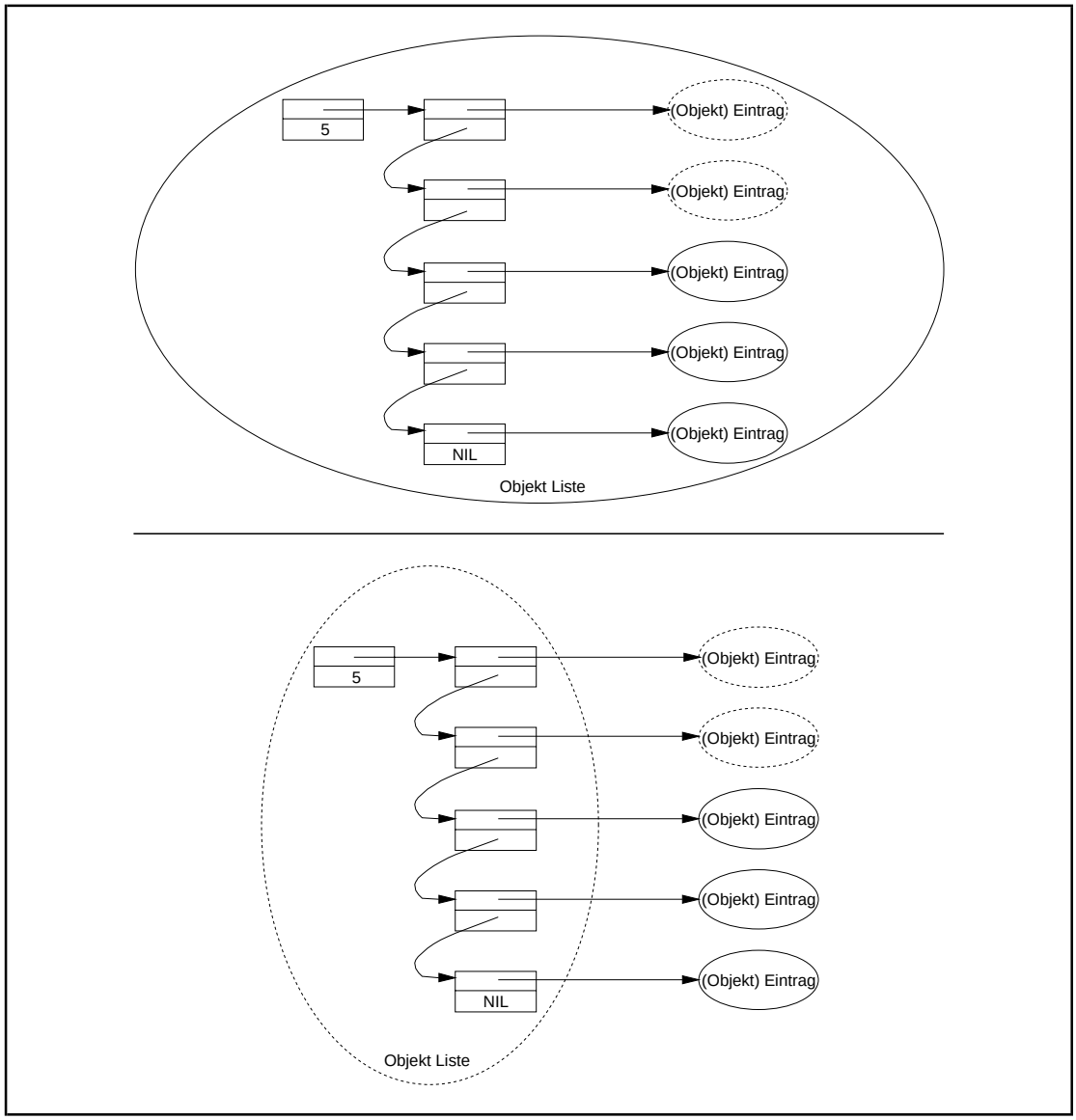


Abbildung 10.1.1-1: Liste

Die folgende Tabelle beschreibt die Benutzerschnittstelle des Objekttyps `TListe`.

Methode	Bedeutung
`CONSTRUCTOR TListe.init;`	Die Liste wird als leere Liste initialisiert.
`PROCEDURE TListe.insert (entry_ptr : Pentry);`	Ein neues Element wird in die Liste aufgenommen.
Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	
`PROCEDURE TListe.delete (entry_ptr : Pentry);`	Alle Vorkommen eines Elements werden aus der Liste entfernt.
Bedeutung des Parameters: `entry-ptr` Verweis auf das zu entfernende Element	
`FUNCTION TListe.is_member (entry_ptr : Pentry) : BOOLEAN;`	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element in der Liste vorkommt, ansonsten den Wert `FALSE`.
Bedeutung des Parameters: `entry-ptr` Zeiger auf das zu überprüfende Element	
`PROCEDURE TListe.for_all (op : Pointer);`	Auf alle zur Zeit in der Liste eingetragene Elemente wird die Operation (Elementmethode) `op` angewendet
Bedeutung des Parameters: `op` Zeiger auf die anzuwendende Elemenentmethode (siehe nachfolgenden Text)	
`PROCEDURE TListe.show;`	Alle zur Zeit in der Liste vorkommenden Elemente werden angezeigt.
`DESTRUCTOR TListe.done;`	Die Liste wird aus dem System entfernt.

Das Einfügen eines Eintrags in die Datenstruktur Liste (Methode `TListe.insert`) erfordert die Erzeugung eines neuen Adreßverweises auf den Eintrag und sein Einhängen in die Kette der Adreßverweise auf die bisherigen Einträge. Günstigerweise wird der neue Adreßverweis an den Anfang der Kette gehängt. Der Methode `TListe.insert` wird nicht das Objekt (Element) selbst, sondern ein Verweis auf das Objekt übergeben. Ein Element kann mehrmals in eine Liste eingefügt werden. Zur Entfernung eines Eintrags (Methode `TListe.delete`) wird zunächst der Verweis gesucht, der auf das Objekt (Element) zeigt, und dann entfernt. Alle Vorkommen eines Elements in der Liste werden entfernt. Die Methode `TListe.is_member` zur ***is_member***-Operation stellt fest, ob es einen Verweis in der Liste auf ein spezifiziertes Element gibt. Allen drei Methoden wird also das spezifizierte Element nicht direkt, sondern als Pointer auf das Element übergeben.

Die Implementierung der Operation $\forall$**op** erfolgt in der Methode `TListe.for_all`. Es wird vorausgesetzt, daß beim Aufruf dieser Methode in der Anwendung als Aktualparameter für den Formalparameter `op` eine Prozedur übergeben wird, die innerhalb des aufrufenden Blocks mit dem Attribut `FAR` deklariert ist und die selbst einen Parameter vom Typ Tentry hat.

Schließlich gibt die Methode `TListe.done` den Speicherplatz auf dem Heap für die Verweisstruktur, aus der ein Objekt vom Typ `TListe` besteht, wieder frei.

Insgesamt kann für die Datenstruktur Liste die Unit `struktur` zur Unit `Liste` abgewandelt werden:

```
UNIT Liste;

INTERFACE

  USES element;

  TYPE PListe = ^TListe;
       TListe = OBJECT
                  PRIVATE
                    first : Pointer;  { Anfang der Verkettung der
                                        Verweise auf Einträge der
                                        Liste                     }
                    anz   : INTEGER;  { Anzahl der Listeneinträge }
                  PUBLIC
                    CONSTRUCTOR init;
                    PROCEDURE insert (entry_ptr : Pentry);
                    PROCEDURE delete (entry_ptr : Pentry);
                    FUNCTION is_member (entry_ptr : Pentry)
                                       : BOOLEAN;
                    PROCEDURE for_all (op : Pointer);
                    PROCEDURE show; VIRTUAL;
                    DESTRUCTOR done; VIRTUAL;
                END;

IMPLEMENTATION

TYPE PListentry = ^TListentry;
     TListentry = RECORD
                    entry_ptr : Pentry;
                    next      : PListentry;
                  END;

     for_all_proc = PROCEDURE (entry : Tentry);


CONSTRUCTOR TListe.init;

  BEGIN { TListe.init };
    first := NIL;
    anz   := 0;
  END   { TListe.init };
```

```
PROCEDURE TListe.insert (entry_ptr : Pentry);

VAR p : PListentry;

  BEGIN { TListe.insert }
    New (p);
    p^.next      := first;
    p^.entry_ptr := entry_ptr;
    Inc(anz);
    first := p;
  END   { TListe.insert };


PROCEDURE TListe.delete (entry_ptr : Pentry);

VAR p           : PListentry;
    q           : PListentry;
    dispose_ptr : PListentry;

  BEGIN { TListe.delete }
    p := first;
    q := NIL;
    WHILE p <> NIL DO
      IF p^.entry_ptr = entry_ptr
      THEN BEGIN
             dispose_ptr := p;
             IF q = NIL THEN first   := p^.next
                        ELSE q^.next := p^.next;
             p := p^.next;
             Dispose (dispose_ptr);
             Dec (anz);
           END
           ELSE BEGIN
                  q := p;
                  p := p^.next;
                END;
  END   { TListe.delete };


FUNCTION TListe.is_member (entry_ptr : Pentry) : BOOLEAN;

VAR p : PListentry;

  BEGIN { TListe.is_member }
    p         := first;
    is_member := FALSE;
    WHILE p <> NIL DO
      IF p^.entry_ptr = entry_ptr
      THEN BEGIN
             is_member := TRUE;
             Break;
           END
      ELSE p := p^.next;
  END   { TListe.is_member };
```

```
PROCEDURE TListe.for_all (op : Pointer);

VAR p : PListentry;

  BEGIN { TListe.for_all }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        for_all_proc(op)(p^.entry_ptr^);
        p := p^.next;
      END;
  END   { TListe.for_all };


PROCEDURE TListe.show;

VAR p : PListentry;

  BEGIN { TListe.show }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        p^.entry_ptr^.display;
        p := p^.next;
      END;
  END   { TListe.show };


DESTRUCTOR TListe.done;

VAR p : PListentry;
    q : PListentry;

  BEGIN { TListe.done }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        q := p;
        p := p^.next;
        Dispose(q);
      END;
  END   { TListe.done };


END.
```

Die folgende Tabelle gibt eine Obergrenze für den Aufwand an, den ein jeweiliger Methodenaufruf benötigt, falls die Liste bereits *n* Elemente enthält.

Methode	Aufwand
`TListe.init;`	$O(1)$
`TListe.insert`	$O(1)$
`TListe.delete`	$O(n)$; dieser Aufwand ergibt sich (im ungünstigsten Fall) auch, wenn nur das erste Vorkommen des Elements entfernt wird
`TListe.is_member`	$O(n)$
`TListe.for_all`	$O(n \cdot A)$, wobei A den Aufwand für die einmalige Ausführung der Operation `op` bezeichnet.
`TListe.show;`	$O(n)$
`TListe.done;`	$O(n)$

10.1.2 FIFO-Warteschlange

Eine **FIFO-Warteschlange** verhält sich ähnlich einer Liste, nur wird jetzt ein neuer Eintrag durch die Operation ***insert*** so in ein Objekt eingefügt, daß ein Aufruf der Operation ***delete*** denjenigen Eintrag an den Aufrufer zurückliefert, der sich am längsten in der FIFO-Warteschlange aufhält. Um diese Bedingung zu erfüllen, wird ein neu aufzunehmendes Element stets an das Ende der aktuellen FIFO-Warteschlange angefügt, während die Operation ***delete*** vom Anfang der FIFO-Warteschlange entfernt.

Die folgende Tabelle beschreibt die Benutzerschnittstelle des Objekttyps `TFIFO`.

Methode	Bedeutung
`CONSTRUCTOR TFIFO.init;`	Die FIFO-Warteschlange wird als leer initialisiert.
`PROCEDURE TFIFO.insert (entry_ptr : Pentry);`	Ein neues Element wird in die FIFO-Warteschlange aufgenommen.
Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	
`PROCEDURE TFIFO.delete (VAR entry_ptr : Pentry);`	Das Element, das sich am längsten in der FIFO-Warteschlange aufhält, wird an den Aufrufer zurückgeliefert und aus der FIFO-Warteschlange entfernt.
Bedeutung des Parameters: `entry-ptr` Verweis auf das entfernte Element	
`PROCEDURE TFIFO.show;`	Alle zur Zeit in der FIFO-Warteschlange vorkommenden Elemente werden angezeigt.
`DESTRUCTOR TFIFO.done;`	Die FIFO-Warteschlange wird aus dem System entfernt.

Abbildung 10.1.2-1 zeigt die interne Struktur der Implementierung eines Objekts vom Objekttyp `TFIFO`. Sie besteht aus zwei Verweisen, nämlich auf den Anfang bzw. das Ende der Liste, einer Komponente, die die aktuelle Anzahl an Einträgen festhält, und jeweils vor- und rückwärtsverketteten Adreßverweisen auf die Elemente der FIFO-Warteschlange. Die Rückwärtsverkettung wird in `TFIFO.delete` verwendet, um direkt den nun als nächsten zu entfernenden Eintrag zu erhalten. Die Deklaration des Objekttyps `TFIFO` steht in der Unit `FIFO`. Prinzipiell läßt er sich vom Objekttyp `TListe` ableiten. Da sich die Datentypen eines Listeneintrags und die Methoden jedoch wesentlich unterscheiden, wird der Objekttyp `TFIFO` neu definiert.

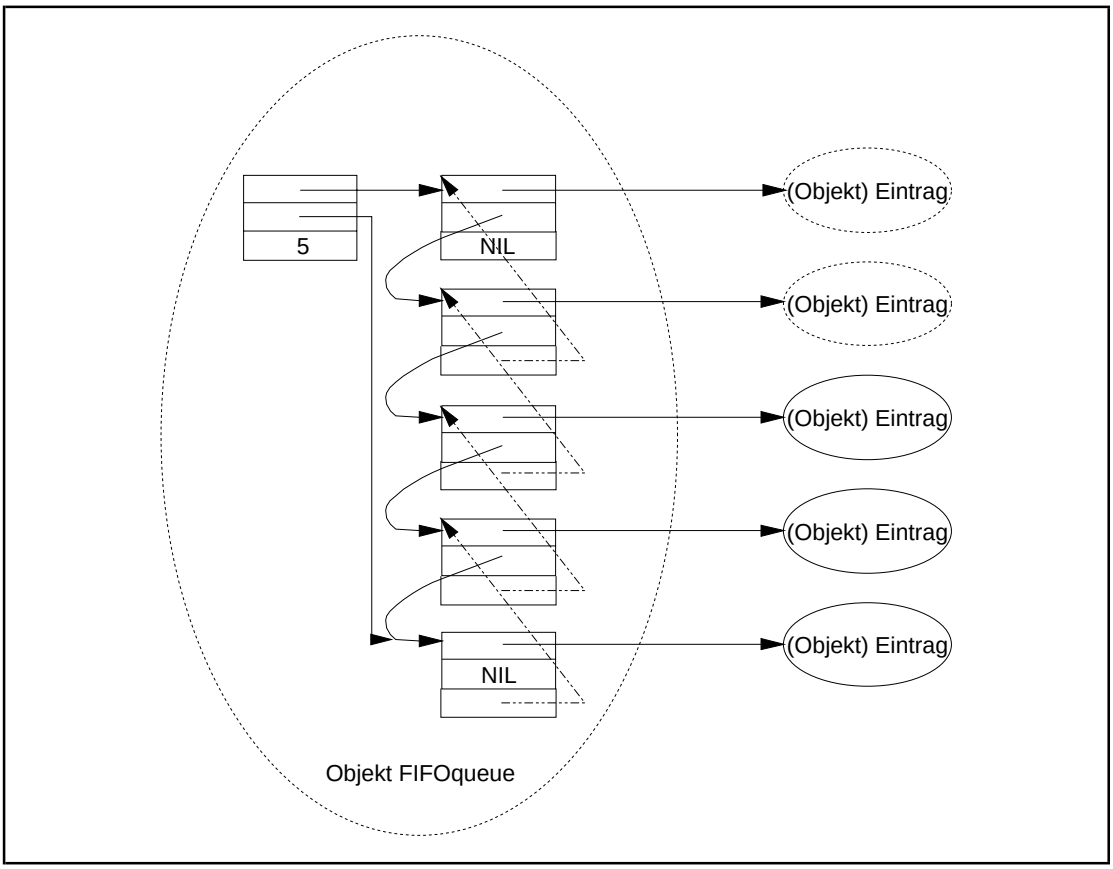


Abbildung 10.1.2-1: FIFO-Warteschlange

```
UNIT FIFO;

INTERFACE

  USES element;

  TYPE PFIFO = ^TFIFO;
       TFIFO = OBJECT
                 PRIVATE
                   first : Pointer;  { Anfang der Verkettung der
                                       Verweise auf Einträge der
                                       FIFO-Warteschlange        }
                   last  : Pointer;  { Ende der Verkettung       }
                   anz   : INTEGER;  { Anzahl der Einträge       }
                 PUBLIC
                   CONSTRUCTOR init;
                   PROCEDURE insert (entry_ptr : Pentry);
                   PROCEDURE delete (VAR entry_ptr : Pentry);
                   PROCEDURE show; VIRTUAL;
                   DESTRUCTOR done; VIRTUAL;
               END;

IMPLEMENTATION

TYPE PFIFOentry = ^TFIFOentry;
     TFIFOentry = RECORD
                    entry_ptr : Pentry;
                    next      : PFIFOentry;
                    last      : PFIFOentry;
                  END;


CONSTRUCTOR TFIFO.init;

  BEGIN { TFIFO.init };
    first := NIL;
    last  := NIL;
    anz   := 0;
  END   { TFIFO.init };


PROCEDURE TFIFO.insert (entry_ptr : Pentry);

VAR p : PFIFOentry;

  BEGIN { TFIFO.insert }
    New (p);
    p^.next      := first;
    p^.last      := NIL;
    p^.entry_ptr := entry_ptr;
    Inc(anz);
    first := p;
    IF last = NIL THEN last := p
                  ELSE p^.next^.last := p;
  END   { TFIFO.insert };
```

```
PROCEDURE TFIFO.delete (VAR entry_ptr : Pentry);

VAR p : PFIFOentry;

  BEGIN { TFIFO.delete }
    IF last <> NIL
    THEN BEGIN
           entry_ptr := PFIFOentry(last)^.entry_ptr;
           p         := last;
           last      := PFIFOentry(last)^.last;
           Dispose (p);
           Dec (anz);
           IF last = NIL THEN first := NIL
                         ELSE PFIFOentry(last)^.next := NIL;
         END
    ELSE entry_ptr := NIL;
  END   { TFIFO.delete };


PROCEDURE TFIFO.show;

VAR p : PFIFOentry;

  BEGIN { TFIFO.show }
    p := last;
    WHILE p <> NIL DO
      BEGIN
        p^.entry_ptr^.display;
        p := p^.last;
      END;
  END   { TFIFO.show };


DESTRUCTOR TFIFO.done;

VAR p : PFIFOentry;
    q : PFIFOentry;

  BEGIN { TFIFO.done }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        q := p;
        p := p^.next;
        Dispose(q);
      END;
  END   { TFIFO.done };


END.
```

Die folgende Tabelle gibt eine Obergrenze für den Aufwand an, den ein jeweiliger Methodenaufruf benötigt, falls die FIFO-Warteschlange bereits *n* Elemente enthält. Zu beachten ist, daß der Aufwand für den Aufruf von `TFIFO.delete` nun konstant ist (im Gegensatz zum entsprechenden Aufruf `TListe.delete`).

Methode	Aufwand
`TFIFO.init;`	$O(1)$
`TFIFO.insert`	$O(1)$
`TFIFO.delete`	$O(1)$
`TFIFO.show;`	$O(n)$
`TFIFO.done;`	$O(n)$

10.1.3 Stack

Die Datenstruktur **Stack** behandelt ihre Einträge nach dem **last-in-first-out-Prinzip**, d.h. derjenige Eintrag wird durch die Operation ***delete*** entfernt, der zuletzt eingetragen wurde[33].

Die Implementierung des Objekttyps `TStack` beinhaltet die folgenden Methoden, die z.T. aus dem Objekttyp `TListe` abgeleitet (geerbt) werden.

Methode	Bedeutung
`CONSTRUCTOR TStack.init;` (*)	Der Stack wird als leer initialisiert.
`PROCEDURE TStack.insert (entry_ptr : Pentry);` (*)	Ein neues Element wird in den Stack aufgenommen.
Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	
`PROCEDURE TStack.delete (VAR entry_ptr : Pentry);`	Das Element, das zuletzt in den Stack aufgenommen wurde, wird an den Aufrufer zurückgeliefert und aus dem Stack entfernt.
Bedeutung des Parameters: `entry-ptr` Verweis auf das entfernte Element	
`PROCEDURE TStack.show;` (*)	Alle zur Zeit im Stack vorkommenden Elemente werden angezeigt.
`DESTRUCTOR TStack.done;` (*)	Der Stack wird aus dem System entfernt.

(*) aus `TListe` abgeleitet.

[33] In Kapitel 7.2 wurden die Operationen PUSH und POP für einen Stack beschrieben. Sie entsprechen den hier benannten Operationen ***insert*** und ***delete***.

Die interne Struktur eines Objekts vom Objekttyp `TStack` besteht aus einem Verweis, nämlich auf den Anfang der Liste, einer Komponente, die die aktuelle Anzahl an Einträgen festhält, und jeweils vorwärtsverketteten Adreßverweisen auf die Elemente des Stacks wie im Objekttyp `TListe`. Der Objekttyp `TStack` wird daher aus dem Objekttyp `TListe` abgeleitet, wobei die Methode `TListe.delete` durch eine eigene Methode überschrieben und die Methoden `TListe.for_all` und `TListe.is_member` durch "leere" Methoden ersetzt werden, weil sie konzeptionell in der Datenstruktur Stack nicht vorkommen. Dazu bedarf es einer Änderung in der Unit `Liste`: Die als `PRIVATE` klassifizierten Komponenten im Objekttyp `TListe` müssen nun nach außen bekannt gegeben werden, da sie ja in eine andere Unit, nämlich die Unit `Stack`, vererbt werden.

```
UNIT Stack;

INTERFACE

  USES Liste, element;

  TYPE PStack = ^TStack;
       TStack = OBJECT (Tliste)
                   PROCEDURE delete (VAR entry_ptr : Pentry);
                   PROCEDURE for_all;    { "Dummy"-Methode }
                   FUNCTION is_member;   { "Dummy"-Methode }
                END;

IMPLEMENTATION

TYPE PListentry = ^TListentry;
     TListentry = RECORD
                    entry_ptr : Pentry;
                    next      : PListentry;
                  END;


PROCEDURE TStack.delete (VAR entry_ptr : Pentry);

VAR p : PListentry;

  BEGIN { TStack.delete }
    IF first <> NIL
    THEN BEGIN
           entry_ptr := PListentry(first)^.entry_ptr;
           p         := first;
           first     := PListentry(first)^.next;
           Dispose (p);
           Dec (anz);
         END
    ELSE entry_ptr := NIL;
  END   { TStack.delete };


PROCEDURE TStack.for_all;
  BEGIN { TStack.for_all }
  END   { TStack.for_all };


FUNCTION TStack.is_member;
  BEGIN { TStack.is_member }
  END   { TStack.is_member };


END.
```

Bezüglich des Aufwands, den ein jeweiliger Methodenaufruf benötigt, falls der Stack bereits *n* Elemente enthält, gelten dieselben Abschätzungen wie bei der FIFO-Warteschlange (Kapitel 10.1.2).

10.1.4 Beschränkter Puffer

Ein weiterer Spezialfall der Datenstruktur Liste ist die Datenstruktur **beschränkter Puffer**. Dieser wird bei der Initialisierung mit *m* leeren Plätzen eingerichtet und kann dann *m* Einträge aufnehmen. Wird versucht, einen weiteren Eintrag aufzunehmen, wird ein bereits im Puffer vorhandener Eintrag überschrieben. ***Es liegt also in der Verantwortung des Anwenders dafür zu sorgen, daß der Puffer nicht "überläuft".***

Eine Implementierung der Datenstruktur kann vom Objekttyp `TListe` abgeleitet werden. Im Konstruktor werden bereits *m* leere Plätze eingerichtet, wobei die Vorwärtsverkettung next im *m*-ten Platz auf den ersten Platz verweist. Man erhält dadurch eine zyklische Liste. Zwei Verweise verwalten die freien und belegten Plätze des Puffers: ein Verweis zeigt auf diejenige Listenposition, die den nächsten Eintrag bei Abarbeitung eines Aufrufs der Methode ***insert*** aufnimmt (der Verweis wird dann um eine Listenposition weitergesetzt), der andere Verweis zeigt auf die Listenposition, die den am längsten in der Datenstruktur vorhandenen Eintrag enthält (auch dieser wird bei jedem Aufruf der Methode ***delete*** um eine Position weitergesetzt).

Im folgenden wird eine Implementierung des Objekttyps `TPuffer` in der Unit `Puffer` beschrieben, die den Gebrauch eines dynamisch angelegten `ARRAY`s in Pascal zeigt.

Der Maximalwert für die Anzahl der verwalteten Puffereinträge wird als Konstante im `INTERFACE`-Teil der Unit definiert:

```
CONST max_eintraege = ...;
```

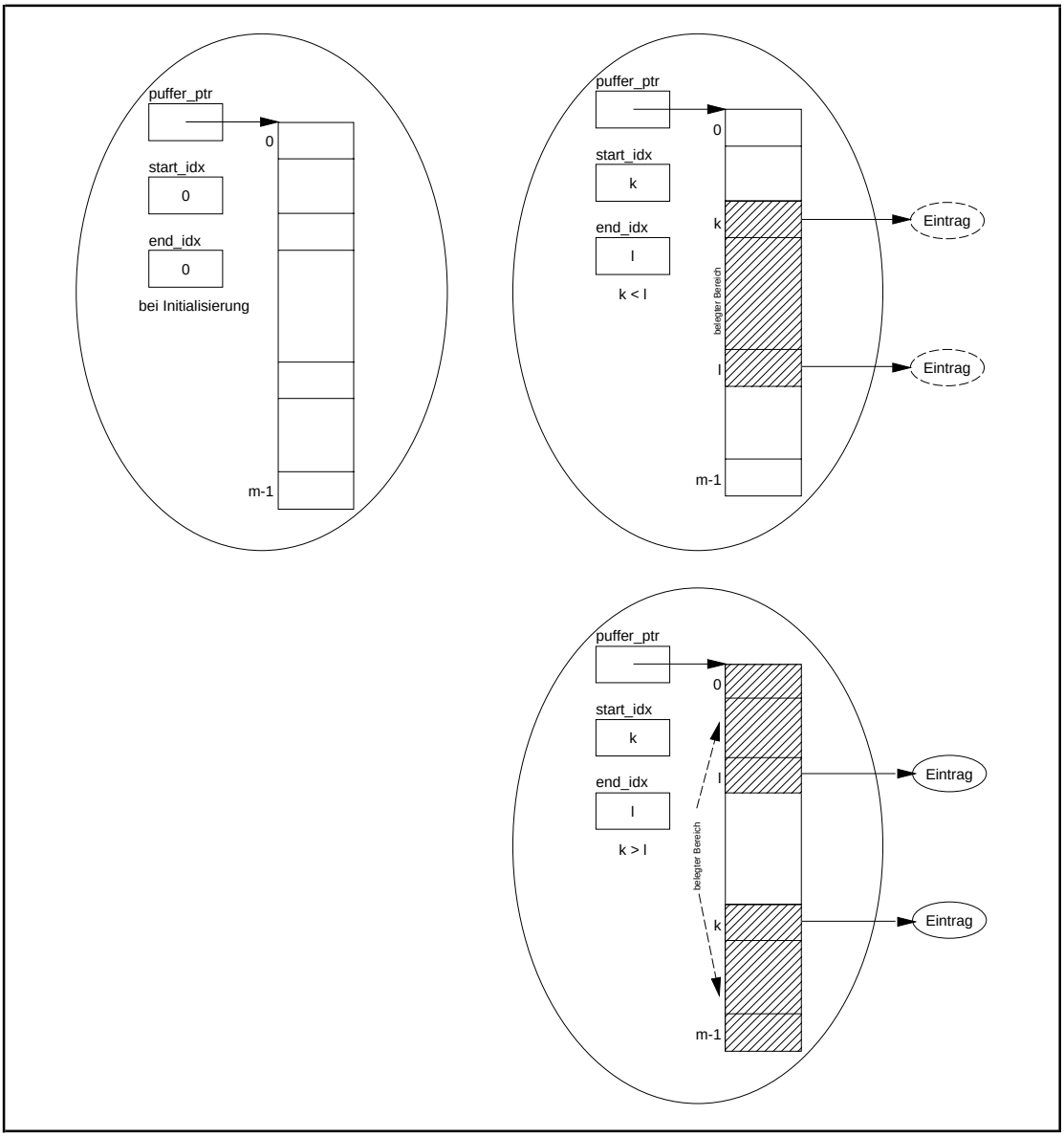


Abbildung 10.1.4-1: Beschränkter Puffer

Die Implementierung sieht wieder vor, daß nicht die Einträge selbst vom Objekttyp `TPuffer` verwaltet werden. Es wird vielmehr bei der Initialisierung ein ARRAY angelegt, das *m* Pointer aufnehmen kann (indiziert von 0 bis *m*-1). Die Komponente `puffer_ptr` eines Objekts vom Typ `TPuffer` adressiert dieses ARRAY. Es enthält Pointer auf diejenigen Einträge, die sich gerade im Puffer befinden, genauer: in der Pufferverwaltung. Der gegenwärtige Wert der Komponente `start_idx` indiziert die Position im ARRAY, die den Pointer auf den Eintrag enthält, der als nächstes aus der Pufferverwaltung entfernt werden soll. Entsprechend indiziert die Komponente `end_idx` die Position im ARRAY, die den Pointer auf den nächsten aufzunehmenden Eintrag enthalten wird. Abbildung 10.1.4-1 zeigt typische Belegungssituationen des Puffers.

Die folgende Tabelle präzisiert die Methodenbeschreibung des Objekttyps `TPuffer`.

Methode	Bedeutung
`CONSTRUCTOR TPuffer.Init (m : INTEGER; VAR OK : BOOLEAN);`	Der Puffer wird initialisiert, wobei in die Pufferverwaltung *m* leere Plätze aufgenommen werden (siehe Abbildung 10.1.4-1 im linken oberen Teil).
Bedeutung der Parameter: `m` Anzahl der Einträge, die im Puffer verwaltet werden `OK` = `TRUE`, falls $0 \leq n \leq$ max_eintraege gilt und genügend freier Platz zur Verwaltung von *m* Einträgen im Puffer zur Verfügung steht = `FALSE`, andernfalls	
`PROCEDURE TPuffer.insert (entry_ptr: Pentry);`	Ein neuer Eintrag wird in die Pufferverwaltung aufgenommen.
Bedeutung des Parameters: `entry_ptr` Verweis auf den einzutragenden Eintrag	
`PROCEDURE TPuffer.delete (VAR entry_ptr : Pentry);`	Der am längsten im Puffer stehende Eintrag wird aus der Pufferverwaltung entfernt und die Adresse des Eintrags an den Aufrufer zurückgegeben
Bedeutung des Parameters: `entry_ptr` Zeiger auf den entfernten Eintrag	
`DESTRUCTOR TPuffer.done;`	Der zyklische Puffer wird aus dem System entfernt.

Im Konstruktor TPuffer.init wird mit Hilfe der **`GetMem`**-Prozedur genau soviel Speicherplatz auf dem Heap angefordert, wie zur Aufnahme von *m* Pointern erforderlich ist. Der Speicherplatz wird im Destruktor `TPuffer.done` mit Hilfe der **`FreeMem`**-Prozedur wieder freigegeben. Um einen Eintrag im ARRAY über einen Feldindex anzusprechen, wird der bereitgestellte Speicherplatz mit einem ARRAY-Datentyp überlagert (typisiert). Weitere Details sind dem im folgenden wiedergegebenen Quelltext der Unit `Puffer` zu entnehmen.

```
UNIT Puffer;

INTERFACE

  USES element;

  CONST max_eintraege = ...;          { maximale Puffergröße      }

  TYPE PPuffer = ^TPuffer;
       TPuffer = OBJECT
                   PRIVATE
                     puffer_ptr : Pointer;
                                  { Zeiger auf das Feld der
                                    Einträge im Puffer        }
                     start_idx  : INTEGER;
                                  { Position des gerade ersten
                                    Eintrags im Puffer        }
                     end_idx    : INTEGER;
```

```
                                    { Position des ersten freien
                                      Eintrags im Puffer        }
      dimension  : INTEGER;
                                    { Puffergröße;
                                      Indizierung des Puffers:
                                      0 .. (dimension - 1)      }
    PUBLIC
      CONSTRUCTOR init (m      : INTEGER;
                        VAR OK : BOOLEAN);
      PROCEDURE insert (entry_ptr : Pentry);
      PROCEDURE delete (VAR entry_ptr : Pentry);
      DESTRUCTOR done;
  END;


IMPLEMENTATION

TYPE TPuffer_array = ARRAY [0..(max_eintraege-1)] OF Pentry;


FUNCTION MyHeapFunc (Size : WORD) : INTEGER; FAR;

BEGIN { MyHeapFunc }
  MyHeapFunc := 1;
END   { MyHeapFunc };


CONSTRUCTOR TPuffer.init (m : INTEGER; VAR OK : BOOLEAN);

VAR OldHeapFunc : Pointer;
    pufferLen   : WORD;

BEGIN { TPuffer.init }
  IF (m < 1) OR (m > max_eintraege)
  THEN BEGIN
         OK := FALSE;
         puffer_ptr := NIL;
       END
  ELSE BEGIN
         OldHeapFunc := HeapError;
         HeapError   := @MyHeapFunc;

         pufferLen := SizeOf(Pointer) * m;
         GetMem (puffer_ptr, pufferLen);
         IF puffer_ptr = NIL
         THEN OK := FALSE
         ELSE BEGIN
                OK        := TRUE;
                start_idx := 0;
                end_idx   := 0;
                dimension := m;
              END;

         HeapError := OldHeapFunc;
       END;
END   { TPuffer.init };
```

```
PROCEDURE TPuffer.insert (entry_ptr : Pentry);

BEGIN { TPuffer.insert }
  TPuffer_array(puffer_ptr^)[end_idx] := entry_ptr;
  end_idx  := (end_idx + 1) MOD dimension;
END   { TPuffer.insert };


PROCEDURE TPuffer.delete (VAR entry_ptr : Pentry);

BEGIN { TPuffer.delete }
  entry_ptr  := TPuffer_array(puffer_ptr^)[start_idx];
  start_idx  := (start_idx + 1) MOD dimension;
END   { TPuffer.delete };


DESTRUCTOR TPuffer.done;

BEGIN { TPuffer.done }
  IF puffer_ptr <> NIL
  THEN FreeMem (puffer_ptr, SizeOf(Pointer)*dimension);
END   { TPuffer.done };


END.
```

Da der Puffer maximal *m* Einträge enthält, läßt sich hier der Aufwand, den ein jeweiliger Methodenaufruf benötigt, durch eine Konstante abschätzen. Zu beachten ist jedoch, daß ohne weitere Kontrollen durch den Benutzer eventuell im Puffer stehende Einträge überschrieben werden.

10.1.5 Hashtabelle

Eine Kombination aus einem beschränkten Puffer (Kapitel 10.1.4) und einer Liste (Kapitel 10.1.1) stellt eine **Hashtabelle** dar. Diese ist eine weitere Implementierung der Datenstruktur Liste, deren ***mittleres Laufzeitverhalten*** sich bei der ***delete***-, ***is_member***- und ∀**op**-Operation als wesentlich besser gegenüber der Implementation mit dem Objekttyp `TListe` erweist.

Im folgenden wird eine Implementierung der Datenstruktur Liste durch den Objekttyp `THashtab` in der Unit `Hash` beschrieben. Zur Vereinfachung werden nur die Methoden zu den Operationen ***insert*** und ***delete*** und der Konstruktor und der Destruktor beschrieben. Dabei soll angenommen werden, daß $m > 0$ eine feste natürliche Zahl ist, die die Größe der Hashtabelle festlegt (die Hashtabelle enthält genau *m* Einträge). Außerdem gebe es eine Abbildung *h*, die jedem Objekt vom Typ `Tentry` einen Wert aus $[0,m-1]$ zuweist; diese Abbildung heißt **Hashfunktion**. Die Deklaration der Hashfunktion befindet sich im `IMPLEMENTATION`-Teil der Unit `Hash`, so daß sie dem Anwender verborgen bleibt.

Die Benutzerschnittstelle der Unit `Hash` lautet:

Methode	Bedeutung

`CONSTRUCTOR`
`  THashtab.Init`
`   (m      : INTEGER;`
`    VAR OK : BOOLEAN);`
	Die Hashtabelle wird mit *m* leeren Plätzen initialisiert.

Bedeutung der Parameter: `m` Anzahl der Einträge der Hashtabelle `OK` = TRUE, falls genügend freier Platz zur Verwaltung von *m* Einträgen in der Hashtabelle zur Verfügung steht = FALSE, andernfalls	

`PROCEDURE`
`  THashtab.insert`
`   (entry_ptr: Pentry);`
	Ein neuer Eintrag wird in die Hashtabelle aufgenommen.

Bedeutung des Parameters: `entry_ptr` Verweis auf den einzutragenden Eintrag	

`PROCEDURE`
`  THashtab.delete`
`   (entry_ptr: Pentry);`
	Das adressierte Element wird aus der Hashtabelle entfernt.

Bedeutung des Parameters: `entry_ptr` Zeiger auf das zu entfernende Element	
`DESTRUCTOR` `  THashtab.done;`	Die Hashtabelle wird aus dem System entfernt.

Ein Objekt vom Typ `THashtab` wird als festes Array mit *m* Einträgen (numeriert von 0 bis $m-1$) implementiert, die über einen Pointer `Hashtab_ptr` adressiert ist. Ein Eintrag in diesem Array ist ein Pointer auf ein Objekt vom Typ `TListe`. Der *i*-te Eintrag ($0 \le i \le m-1$) im Array adressiert diejenige Liste, in die alle Objekte vom Typ `Tentry` aufgenommen werden, für die die Hashfunktion den Wert *i* liefert.

Das Verfahren wird in Abbildung 10.1.5-1 erläutert. Die zu verwaltenden Elemente vom Typ `Tentry` werden hier durch natürliche Zahlen repräsentiert. Die Hashtabelle enthält $m = 7$ Einträge. Als Hashfunktion wird die durch

$h(x) = x \text{ MOD } m$

definierte Funktion genommen. Der linke Teil der Abbildung zeigt ein Objekt vom Typ `THashtab`, nachdem nacheinander die durch 3, 13, 7, 1, 18, 25, 29 und 9 repräsentierten Elemente vom Typ `Tentry` aufgenommen wurden. Der rechte Teil zeigt die Hashtabelle nach Entfernen der durch 29, 3 und 18 repräsentierten Elemente.

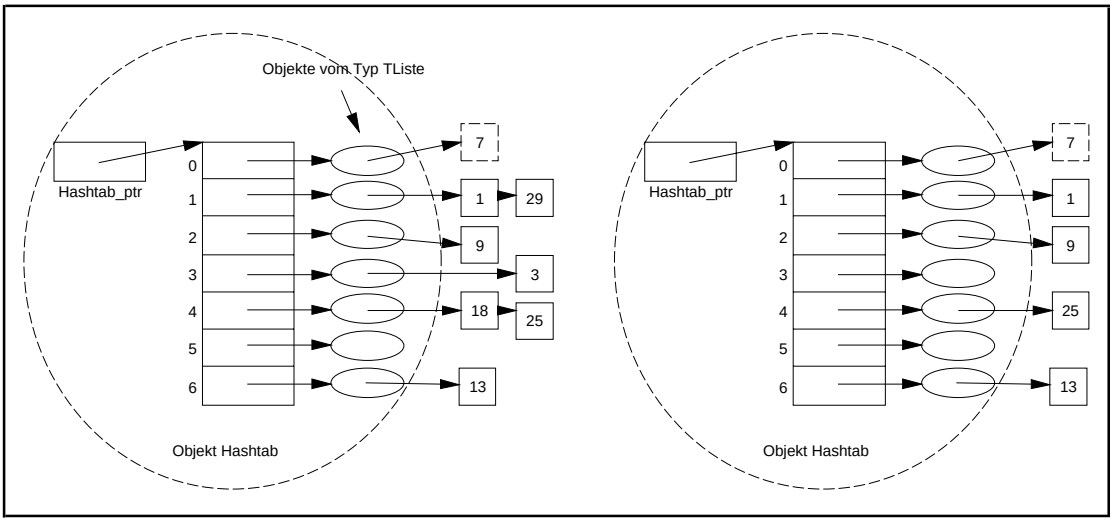


Abbildung 10.1.5-1: Hashtabelle

Im folgenden ist eine Implementierung des Objekttyps `THashtab` wiedergegeben.

```
UNIT Hash;

INTERFACE

USES element;

CONST max_eintraege = ...;

TYPE PHashtab = ^THashtab;
     THashtab = OBJECT
                  PRIVATE
                    Hashtab_ptr : Pointer;
                                  { Zeiger auf die eigentliche
                                    Hashtabelle                 }
                    dimension   : INTEGER;
                                  { Größe der Hashtabelle       }
                  PUBLIC
                    CONSTRUCTOR init (m      : INTEGER;
                                      VAR OK : BOOLEAN);
                    PROCEDURE insert (entry_ptr : Pentry);
                    PROCEDURE delete (entry_ptr : Pentry);
                    DESTRUCTOR done;
                  END;


IMPLEMENTATION

USES Liste;

TYPE THashtab_array = ARRAY [0..max_eintraege] OF PListe;
```

```
{ Hashfunktion }
FUNCTION Hashfunktion (entry_ptr : Pentry) : INTEGER;
  { Die Funktion ermittelt den Wert der Hashfunktion
    für das durch entry_ptr adressierte Objekt       }
BEGIN { Hashfunktion }
  ...
END   { Hashfunktion };


FUNCTION MyHeapFunc (Size : WORD) : INTEGER; FAR;

BEGIN { MyHeapFunc }
  MyHeapFunc := 1;
END   { MyHeapFunc };


CONSTRUCTOR THashtab.init (m : INTEGER; VAR OK : BOOLEAN);

VAR OldHeapFunc : Pointer;
    idx         : INTEGER;

BEGIN { THashtab.init }
  IF (m < 1) OR (m > max_eintraege)
  THEN BEGIN
         OK          := FALSE;
         Hashtab_ptr := NIL;
       END
  ELSE BEGIN
         OldHeapFunc := HeapError;
         HeapError   := @MyHeapFunc;

         GetMem (Hashtab_ptr, SizeOf(Pointer) * m);
         IF Hashtab_ptr = NIL
         THEN OK := FALSE
         ELSE BEGIN
                OK        := TRUE;
                dimension := m;
                FOR idx := 0 TO m-1 DO
                  THashtab_array(Hashtab_ptr^)[idx]^.init;
              END;

         HeapError := OldHeapFunc;
       END;
END   { THashtab.init };


PROCEDURE THashtab.insert (entry_ptr : Pentry);

BEGIN { THashtab.insert }
  THashtab_array(Hashtab_ptr^)[Hashfunktion(entry_ptr)]^.insert
                                                      (entry_ptr);
END   { THashtab.insert };


PROCEDURE THashtab.delete (entry_ptr : Pentry);

BEGIN { THashtab.delete }
  THashtab_array(Hashtab_ptr^)[Hashfunktion(entry_ptr)]^.delete
                                                      (entry_ptr);
END   { THashtab.delete };
```

```
DESTRUCTOR THashtab.done;

VAR idx : INTEGER;

BEGIN { THashtab.done }
  IF Hashtab_ptr <> NIL
  THEN BEGIN
         FOR idx := 0 TO dimension - 1 DO
           THashtab_array(Hashtab_ptr^)[idx]^.done;
         FreeMem (Hashtab_ptr, SizeOf(Pointer)*dimension);
       END;
END   { THashtab.done };


END.
```

Die Hashfunktion bestimmt, in welche (Teil-) Liste der Hashtabelle ein neu aufzunehmendes Element eingetragen wird. Bei ungünstiger Wahl der Hashfunktion werden viele Elemente in nur wenige Listen aufgenommen. In diesem Fall ist das Laufzeitverhalten einer Hashtabelle nicht besser als dasjenige der Datenstruktur `TListe`. Insbesondere werden im ungünstigsten Fall bei Vorhandensein von bereits n Elementen $O(n)$ viele Teilschritte für die einmalige Ausführung eines Aufrufs `THashtab.delete` ausgeführt.

Die Hashfunktion ist besonders gut, wenn die eingetragenen Elemente möglichst gleichmäßig über alle Listen verteilt werden. Es soll daher das ***mittlere Laufzeitverhalten*** von n ***insert***- und ***delete***-Operationen auf einer anfangs leeren Hashtabelle betrachtet werden. Dabei werden an die Hashfunktion folgende Bedingungen gestellt:

1. Die Berechnung eines Werts der Hashfunktion benötigt konstanten Aufwand.

2. Die Hashfunktion verteilt alle einzufügenden Elemente gleichmäßig über das Intervall $[0,m-1]$, d.h. für alle i und j aus dem Intervall $[0,m-1]$ gilt: $||h^{-1}(i)| - |h^{-1}(j)|| \leq 1$

3. Sämtliche mögliche einzufügende Elemente sind mit gleicher Wahrscheinlichkeit Argument der nächsten Operation.

Dann läßt sich zeigen ([BLU]): Der Erwartungswert für die von einer Folge von n Einfüge- und Zugriffsoperationen benötigte Zeit ist $\leq (1+\beta/2)\cdot n$ mit $\beta = n/m$.

Die Bedingungen 1 und 2 lassen sich durch Hashfunktionen der Form $h(x) = x \text{ MOD } m$ realisieren, wobei hierbei m sorgfältig gewählt werden muß (siehe [O/W]). Bedingung 3 ist in der Praxis i.a. nicht leicht zu erreichen, da sie ein bestimmtes Verhalten des Benutzers der Hashfunktion postuliert.

Wählt man m zu Beginn hinreichend groß, so läßt sich die Größe $\beta = n/m$ in obiger Abschätzung durch eine Konstante beschränken, so daß der obige Erwartungswert von

der Ordnung $O(n)$ ist. In der Implementierung `TListe` benötigen n Aufrufe von `TListe.delete` (nachdem n Elemente eingefügt wurden) im ungünstigsten Fall $O(n^2)$ viele Verarbeitungsschritte.

Abschließend sei bemerkt, daß in der Literatur eine Reihe weiterer Hashverfahren beschrieben wird (vgl. z.B. [O/W] und [BLU]).

10.1.6 Menge und Teilmengensystem einer Grundmenge

Pascal hat einen im Sprachkonzept vorgesehenen Mengentyp (Kapitel 4.2). Eine Deklaration der Form

```
TYPE TMenge = SET OF typ;
```

erwartet, daß `typ` einen Ordinaltyp bezeichnet. Die üblichen Mengenoperationen lassen sich mit den definierten Operatoren realisieren. Sind beispielsweise die Variablen

```
VAR amenge : TMenge;
    bmenge : TMenge;
    cmenge : TMenge;
```

deklariert, so erfolgt die Initialisierung der Menge `amenge` als leere Menge durch

```
amenge := [];
```

Die Mengenoperationen $\cup$, $\cap$ und \ lauten

```
cmenge := amenge + bmenge; { Vereinigung }
cmenge := amenge * bmenge; { Schnitt     }
cmenge := amenge - bmenge; { Differenz   }
```

Ist `x` eine Variable vom Typ der Elemente in `amenge`, etwa

```
VAR x : typ;
```

so realisiert

```
x IN amenge
```

die Enthaltenseinrelation $\in$; die Teilmengenrelation $\subseteq$ ist durch

```
bmenge <= cmenge
```

erklärt.

In einer Pascal-Menge können nur Elemente vom Ordinaltyp liegen, da eine Menge intern auf ein Byte abgebildet wird. Das bedeutet, daß Mengen, die aus Objekten eines Objekttyps bestehen, nicht definierbar sind. Viele Anwendungen erfordern aber Definitionsmöglichkeiten, in denen **Mengen aus Objekten** eines (allgemeinen) Objekttyps

bestehen. Es ist daher ein Objekttyp `TMenge` erforderlich, der die in 10-2 beschriebenen Mengen mit entsprechenden Mengenoperationen zuläßt. Eine Möglichkeit besteht darin, Mengen in Form von Listen wie in Kapitel 10.1.1 zu implementieren, d.h. als lineare Datenstruktur.

Einige Anwendungen erfordern die Implementation ***disjunkter Teilmengen einer Grundmenge mit speziellen Mengenoperationen***: Bei der Initialisierung wird eine Teilmenge als leere Menge oder als Menge erzeugt, die genau ein Element enthält. Eine Grundoperation vereinigt zwei disjunkte Teilmengen zu einer weiteren Menge. Typische Anwendungen führen auf eine als **Union-Find-Struktur** bezeichnete Datenstruktur. Dazu gehören Algorithmen, die Mengenpartitionen gemäß einer Äquivalenzrelation manipulieren, oder Algorithmen auf Graphen, z.B. bei der Suche von minimalen aufspannenden Bäumen in Kommunikationsnetzen (siehe [O/W]).

Eine Union-Find-Struktur besteht also aus einem Objekt, das disjunkte Teilmengen einer Grundmenge enthält; jede dieser Teilmengen enthält als Elemente Objekte vom Typ `Tentry` (Abbildung 10.1.5-1). Eine Union-Find-Struktur ist also ein **Mengensystem**. Die spezielle Form der Union-Find-Strukturen soll in diesem Kapitel betrachtet werden; sie führt auf eine nichtlineare Implementierung einer im Grunde linearen Datenstruktur.

Die Implementierung erfolgt durch die Implementierung der in ihm enthaltenen Teilmengen als dynamisch erzeugt Objekte. Die Organisation der Teilmengen im Mengensystem kann z.B. als Liste erfolgen; sie soll hier nicht weiter betrachtet werden. Vielmehr geht es um eine für die speziellen Operationen der Union-Find-Struktur geeignete Implementierung der Teilmengen, für die ein Objekttyp `TUnion_Find` deklariert wird.

Aus der Tatsache, daß mit disjunkten Teilmengen einer Grundmenge hantiert wird, erübrigt sich für `TUnion_Find` die Operation $T \cap S$, da diese immer die leere Menge ergibt. Die ***init***-Operation initialisiert eine Teilmenge als leere Menge oder als einelementige Teilmenge der Grundmenge; es muß ihr also als Parameter das eventuelle "Initialisierungselement" mitgegeben werden, d.h. sie hat zur Erzeugung der Teilmenge $T = \{a_i\}$ das Aufrufformat init (a_i). Dabei soll vorausgesetzt werden, daß stets alle Initialisierungselemente neue, bisher noch nicht verwendete Elemente der Grundmenge sind. Die ***init***-Operation vergibt implizit auch eine ***Identifikation*** für die Teilmenge, und zwar wird zunächst als Identifikation der Teilmenge das Element genommen, das anfangs in ihr enthalten ist. Das Element a_i in einer Teilmenge der Form $T = \{a_i\}$ dient also auch als Identifikation von T; gleichzeitig liegt innerhalb T eine Repräsentation des Elements a_i (wieder als Adreßverweis auf a_i). Im Laufe der Lebensdauer einer Teilmenge, wenn sie nämlich mit anderen Teilmengen vereinigt wird und dann mehrere Elemente enthält, kann sich ihre Identifikation ändern (siehe unten).

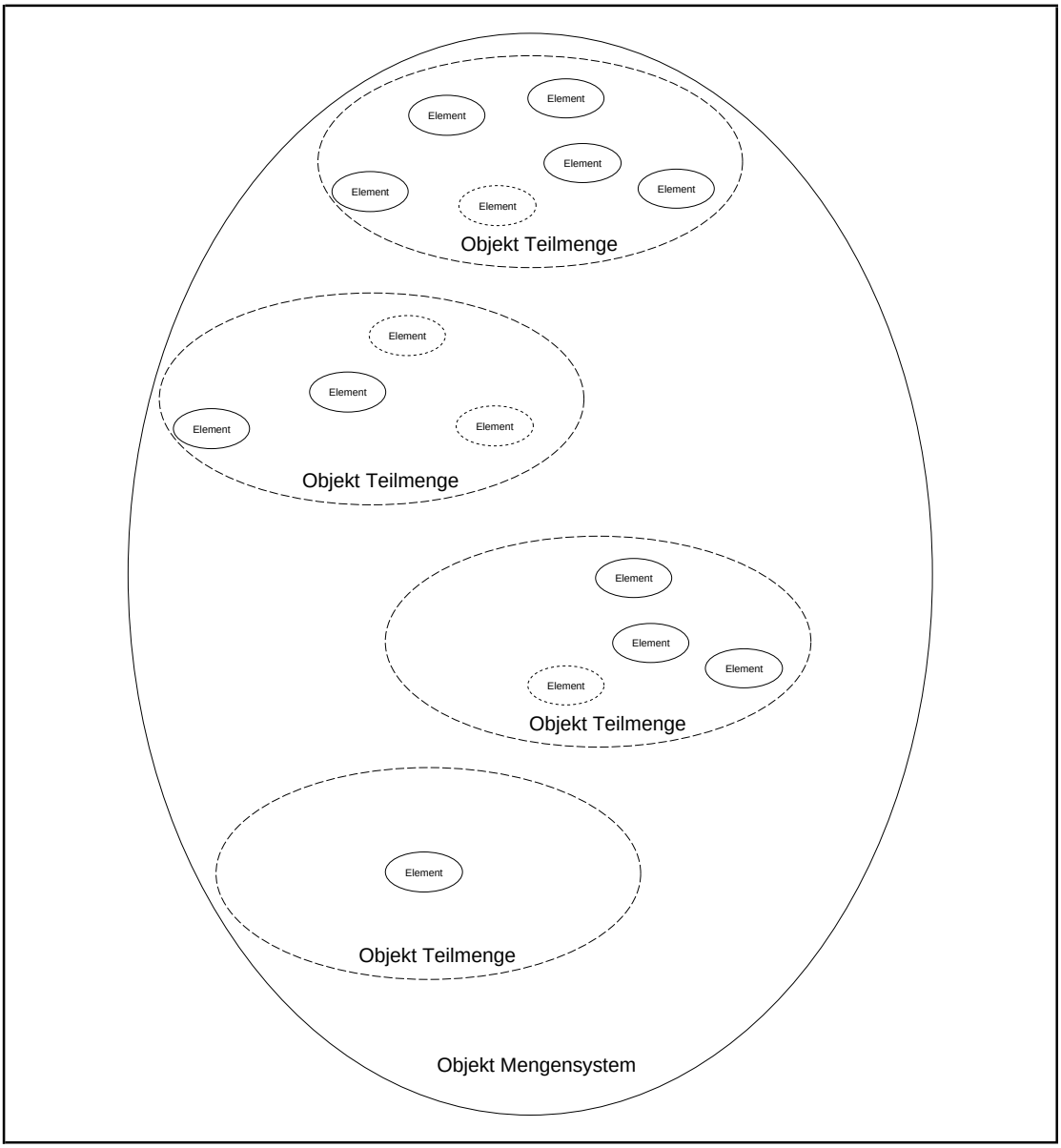


Abbildung 10.1.5-1: Union-Find-Struktur

In den Anwendungen ist es wichtig, für ein Element *x* die gegenwärtige Identifikation derjenigen Teilmenge zu ermitteln, die *x* enthält (diese ist eindeutig, da hier ja nur disjunkte Teilmengen der Grundmenge vorkommen). Dazu dient die Operation ***find*** (*x*). Zwei Elemente *x* und *y* liegen bei ***find*** (*x*) = ***find*** (*y*) in derselben Teilmenge. Für eine effiziente Implementierung der ***find***-Operation wird in die ***init***-Operation ein Rückgabewert eingebaut: ***init*** gibt einen Pointer auf die Repräsentation des Initialisierungselements in *T* zurück, *der vom Anwender nicht geändert werden darf*. Mit Hilfe dieses Pointers durchsucht ein Aufruf ***find*** (*x*) die Teilmenge nach deren Identifikation; der Pointer verweist auf die Repräsentation von *x*, da er bei der Initialisierung der Teilmenge $\{x\}$, die *x* zum ersten Mal enthielt, so erzeugt wurde.

Als weitere Mengenoperation ist die Vereinigung $T \cup S$ zweier Teilmengen T und S Mengen möglich sein. Durch die Annahmen über die ***init***-Operation werden Vereinigungsoperationen nur mit disjunkten Mengen durchgeführt. Die Operation ***union*** (T, S) bewirkt entweder $T := T \cup S$ oder $S := T \cup S$, je nachdem, ob T bzw. S mindestens soviele Elemente wie die andere Menge hat. Gleichzeitig bekommt die Vereinigungsmenge die Identifikation von T bzw. von S, und die andere Menge wird aus dem Mengensystem entfernt.

Auf eine Operation ***show***, die die Elemente einer Teilmenge anzeigt, soll hier verzichtet werden, da Teilmengen im folgenden als nichtlinerare Datenstrukturen implementiert werden und im Kapitel 10.2 entsprechende Implementierungen von ***show*** gezeigt werden.

Eine Teilmenge $T = \{a_1, \ldots, a_n\}$ der Grundmenge M kann konzeptionell als gerichteter Baum mit durch Elemente markierte Knoten angesehen werden, wobei die Wurzel des Baums mit einem der Elemente a_i und die übrigen Knoten mit a_j mit $i \neq j$ markiert sind. Von jedem Knoten, der mit a_j markiert ist, führt eine Pfad über eine oder mehrere Kante zu dem mit a_i markierten Knoten. Das Element a_i stellt auch die Identifikation von T dar. Abbildung 10.1.5-2 zeigt im oberen Teil Beispiele von Teilmengen S_1 = {1, 7, 8, 9}, S_2 = {2, 5, 10} und S_3 = {3, 4, 6} der Grundmenge M = {1, 2, ..., 10} und zwei prinzipielle Möglichkeiten, $S_1 \cup S_2$ darzustellen. In diesem Beispiel sind die Elemente als Objekte natürliche Zahlen. Die oben beschriebene Operation ***union*** (S_i, S_j) zur Vereinigung $S_i \cup S_j$ zweier Teilmengen S_i und S_j bildet die linke Alternative in Abbildung 10.1.5-2 nach: Es wird diejenige Teilmenge S_j an S_i als Unterbaum angehängt, die weniger Elemente enthält.

In Abbildung 10.1.5-2 ist ***find*** (9) auf der Teilmenge S_1 das Element 1 und ***find*** (10) auf der (linken Alternative) von $S_1 \cup S_2$ das Element 1, ebenso wie ***find*** (2) oder ***find*** (8).

Die Annahmen führen auf folgende Unit `Unionfd` zur Implementierung einer Union-Find-Struktur auf einer Grundmenge aus Objekten vom Objekttyp `Tentry`. Die Teilmengen haben den Objekttyp `TUnion_Find`. Wieder werden in einem Objekt vom Typ `TUnion_Find` die in ihr enthaltenen Objekte nicht selbst, sondern Verweise auf die jeweiligen Objekte festgehalten: Der untere Teil von Abbildung 10.1.5-2 zeigt die Implementierung für eine einelementige Teilmenge und für $S_1 \cup S_2$. Die Erzeugung der Elemente liegt wieder in der Kontrolle der Anwendung. Alle weiteren Details können dem folgenden Code entnommen werden.

Die Implementierung des Destruktors `TUnion_Find.done` ist aus Gründen der Übersichtlichkeit nicht ausgeführt; hierfür empfiehlt es sich nämlich, die Einträge im Baum, die eine Teilmenge ausmachen, zusätzlich vorwärts zu verketten, damit sie von der Wurzel aus erreicht werden können.

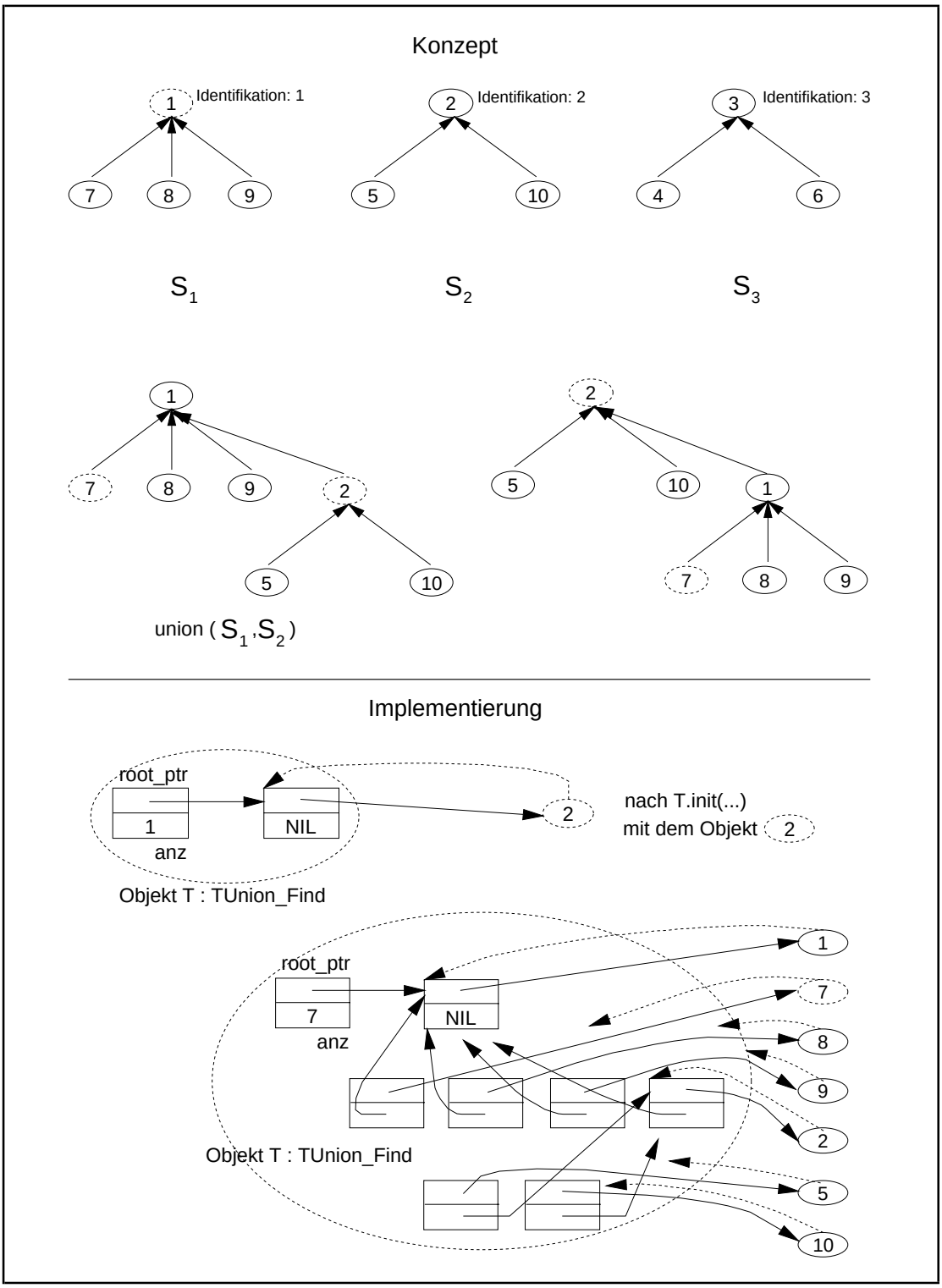


Abbildung 10.1.5-2: Teilmengen einer Grundmenge (Beispiel)

Die folgende Tabelle beschreibt die Benutzerschnittstelle der Unit `Unionfd`.

Methode	Bedeutung
`CONSTRUCTOR TUnion_Find.init (entry_ptr : Pentry; VAR strukt_ptr : Pointer);`	Es wird eine Teilmenge mit genau einem Element eingerichtet.

Bedeutung der Parameter:

`entry_ptr` Verweis auf das Element der erzeugten Teilmenge; bei NIL, wird die leere Menge erzeugt.

`strukt_ptr` Verweis auf die Repräsentation der erzeugten Teilmenge (gestrichelte Linie in Abbildung 10.1.5-2); *der Wert darf vom Anwender nicht geändert werden*.

`DESTRUCTOR TUnion_Find.done;`	Die Teilmenge wird aus dem System entfernt.
`FUNCTION find (strukt_ptr : Pointer): Pentry;`	Die Funktion ermittelt einen Verweis auf das Element, das die Teilmenge identifiziert, die das durch strukt_ptr repräsentierte Element enthält.

Bedeutung des Parameters:

`strukt_ptr` Verweis auf die Repräsentation eines Elements

`PROCEDURE Union (VAR T : PUnion_Find; VAR S : PUnion_Find);`	Die Prozedur bildet die Vereinigung von T und S in der Teilmenge, die mindestens so viele Elemente wie die andere Teilmenge enthält, die dann entfernt wird

Bedeutung der Parameter:

`T bzw. S` Verweise auf die zu vereinigenden Teilmengen

```
UNIT Unionfd;

INTERFACE

  USES element;

  TYPE PMengensystem = ^TMengensystem;
       TMengensystem = OBJECT
                         ...
                       END;

       PUnion_Find = ^TUnion_Find;
       TUnion_Find = OBJECT
                     PRIVATE
                       root_ptr : Pointer;
                                  { Verweis auf das Element,
                                    das die Menge identifiziert  }
                       anz  : INTEGER;
                                  { Anzahl der Elemente der Menge}
                     PUBLIC
                       CONSTRUCTOR init (entry_ptr      : Pentry;
```

```
                                          VAR strukt_ptr : Pointer);
                      DESTRUCTOR done; VIRTUAL;
                    END;

  FUNCTION find (strukt_ptr : Pointer) : Pentry;

  PROCEDURE Union (VAR T : PUnion_Find;
                   VAR S : PUnion_Find);
  { Bildet die Vereinigung von T und S in der Teilmenge, die
    mindestens so viele Elemente wie die andere Teilmenge enthält,
    die dann entfernt wird                                          }


IMPLEMENTATION

TYPE PUnion_Findentry = ^TUnion_Findentry;
     TUnion_Findentry = RECORD
                          entry_ptr : Pentry;
                          pred      : PUnion_Findentry;
                        END;


CONSTRUCTOR TUnion_Find.init (entry_ptr      : Pentry;
                              VAR strukt_ptr : Pointer);

  BEGIN { TUnion_Find.init };
    IF entry_ptr = NIL
    THEN BEGIN
           root_ptr   := NIL;
           anz        := 0;
           strukt_ptr := NIL;
         END
    ELSE BEGIN
           New (PUnion_Findentry(root_ptr));
           PUnion_Findentry(root_ptr)^.entry_ptr := entry_ptr;
           PUnion_Findentry(root_ptr)^.pred      := NIL;
           anz        := 1;
           strukt_ptr := root_ptr;
         END;
  END   { TUnion_Find.init };


DESTRUCTOR TUnion_Find.done;

  BEGIN { TUnion_Find.done }
    ...
  END   { TUnion_Find.done };


FUNCTION find (strukt_ptr : Pointer) : Pentry;

VAR p : PUnion_Findentry;

  BEGIN { find }
    p := strukt_ptr;
    WHILE p^.pred <> NIL DO
      p := p^.pred;

    find := p^.entry_ptr;
  END   { find };


PROCEDURE Union (VAR T : PUnion_Find;
                 VAR S : PUnion_Find);

VAR U : PUnion_Find;
    V : PUnion_Find;
```

```
  BEGIN { Union }
    IF T^.anz >= S^.anz THEN BEGIN
                               U := T;
                               V := S;
                             END
                        ELSE BEGIN
                               U := S;
                               V := T;
                             END;
    U^.anz := U^.anz + V^.anz;
    PUnion_Findentry(V^.root_ptr)^.pred := U^.root_ptr;

    Dispose (V, done);

  END   { Union };


END.
```

Die folgenden Aufwandsabschätzung einer Folge von ***union***- und ***find***-Operationen legt eine Modifikation der Operation ***find*** nahe.

Die einmalige Durchführung der ***union***-Operation benötigt einen konstanten Aufwand. Beginnt man mit n Union-Find-Strukturen, die mit jeweils einem Element initialisiert wurden, und führt (n-1)-mal die ***union***-Operation aus, so haben alle zwischenzeitlich entstandenen Bäume höchstens eine Höhe h mit $h \leq \log_2 n$ (siehe [O/W]), so daß eine ***find***-Operation dann höchstens $O(\log n)$ Schritte benötigt. Eine Verbesserung erhielte man, indem man bei der ***union***-Operation alle Elemente unter den obersten Knoten der größeren Menge hängt, allerdings auf Kosten des konstanten Aufwands. Eine andere Möglichkeit besteht darin, einen Baum bei der ***find***-Operation zu reorganisieren, da er ja sowieso durchlaufen werden muß. Der erhöhte Aufwand, der dabei bei der ***find***-Operation entsteht, wirkt sich günstig auf die Höhe des Baums aus, so daß spätere ***find***-Operationen weniger aufwendig ablaufen. Daher wird in die Methode zur ***find***-Operation eine **Pfadkompression** eingebaut. Drei Ansätze sind in Abbildung 10.1.5-3 zu sehen. Die Objekte in einer Union-Find-Struktur werden wieder durch Knoten dargestellt, die mit dem Objekt bezeichnet sind. Das Objekt in der Wurzel ist die Identifikation einer Teilmenge. Es werde ***find*** (x) aufgerufen, so daß der "Einstieg" in den Baum beim Objekt x erfolgt und ein Pfad bis zur Wurzel (mit dem Objekt) v durchlaufen wird. Abbildung 10.1.4-3 zeigt das Beispiel einer Ausgangssituation, bei der auf dem Pfad von x nach v die Objekt y, z, w und u liegen. Weitere mögliche Teilbäume unter den Objekten sind durch Dreiecke angedeutet.

Die Ansätze zur Pfadkompression in Abbildung 10.1.5-3 sind:

- **Kollapsregel**: Es werden alle Knoten auf dem Pfad von x zur Wurzel direkt unter die Wurzel gehängt. Da man die Wurzel erst finden muß, wird der Baum dabei zweimal durchlaufen.

- **Pfad-Splitting**: Während des Durchlaufens des Pfads von x nach v wird jeder Knoten (und der darunter hängende Teilbaum) unter seinen übernächsten Nachfolger gehängt.

- **Pfad-Halbierung**: Während des Durchlaufens des Pfads von *x* nach *v* wird der *i*-te Knoten bei ungeradem *i* (und der darunter hängende Teilbaum) unter seinen übernächsten Nachfolger gehängt; die Position innerhalb des Baum des *i*-te Knotens bei geradem *i* (und der darunter hängende Teilbaum) bleibt unverändert.

Die Bedeutung der Kollapsregel läßt sich in folgender Aussage zusammenfassen (siehe [O/W]).

> Werden *n* ***init***-Operationen zur Erzeugung einelementiger Teilmengen und anschließend höchstens *n*-1 ***union***-Operationen, gemischt mit einer Reihe von ***find***-Operationen durchgeführt, in der die Kollapsregel eingebaut ist, und ist die Gesamtanzahl der ***union***- und ***find***-Operationen $m \geq n$, so erfordern alle Operationen zusammen einen Gesamtaufwand von $O(m \cdot \alpha(m,n))$ Rechenschritten. Hierbei ist $\alpha(m,n)$ die (extrem langsam wachsende) inverse Funktion zur Ackermann-Funktion[34]. Es ist z.B. $\alpha(m,n) \leq 3$ für $n < 2^{16} = 65.536$ und $\alpha(m,n) \leq 4$ für alle praktisch auftretenden Werte von *m* und *n*, so daß $\alpha(m,n)$ als praktisch konstant angesehen werden kann.

Die ***find***-Operation mit Kollapsregel lautet:

```
FUNCTION find (strukt_ptr : Pointer) : Pentry;

VAR p : PUnion_Findentry;
    q : PUnion_Findentry;
    u : PUnion_Findentry;

  BEGIN { find }
    p := strukt_ptr;
    WHILE p^.pred <> NIL DO
      p := p^.pred;
    { Kollapsregel }
        q := strukt_ptr;
        WHILE q <> p DO
          BEGIN
            u       := q^.pred;
            q^.pred := p;
            q       := u;
          END;

    find := p^.entry_ptr;
  END   { find };
```

---

[34] Die Ackermannfunktion $A(i,j)$ ist für $i > 0$ und $j > 0$ definiert durch
$A(1,j) = 2^j$ für $j > 0$

$A(i,1) = A(i-1,2)$ für $i > 1$

$A(i,j) = A(i-1,A(i,j-1))$ für $i > 1$ und $j > 1$.
Die inverse Funktion zu $A(i,j)$ lautet für $m \geq n > 0$
$\alpha(m,n) = \min\{i \mid i > 0 \wedge A(i,\lfloor m/n \rfloor) > \log_2 n\}$.

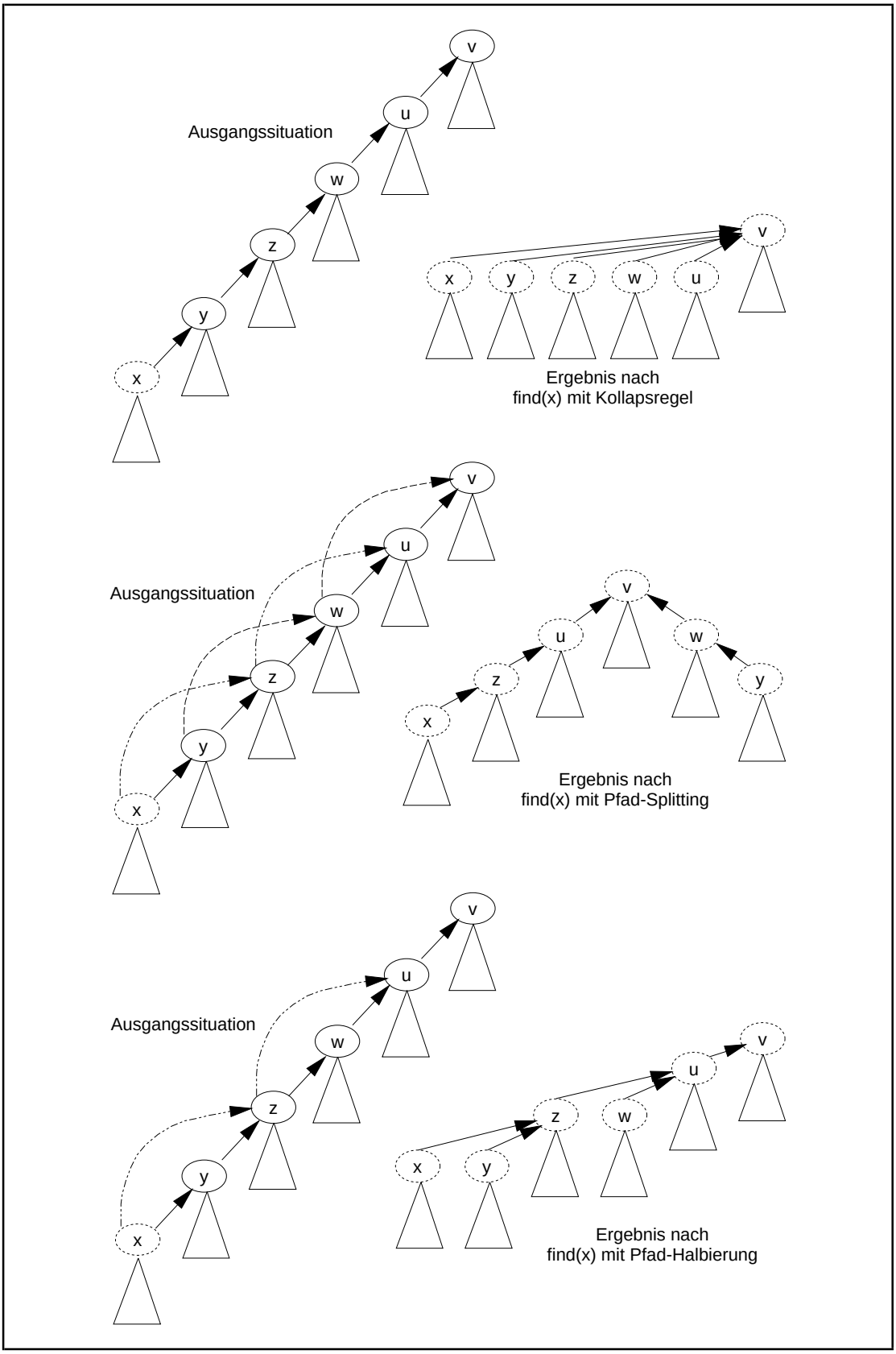


Abbildung 10.1.5-3: Pfadkompression

Eine alternative Möglichkeit zur Entscheidung, welche Teilmenge als Ergebnismenge der ***union***-Operation genommen wird, ist nicht die Anzahl der Elemente in den Teilmengen, sondern die Höhen der Bäume, die die Teilmengen darstellen. Dadurch gilt ebenfalls die obige Komplexitätsabschätzung.

10.2 Nichtlineare Datenstrukturen

Unter einer **nichtlinearen Datenstruktur** werden alle übrigen Datenstrukturen zusammengefaßt. Dazu gehören insbesondere diejenigen, auf denen Operationen definiert sind, die davon Gebrauch machen, daß auf den Elementen der Datenstruktur eine Ordnungsrelation ≤ (Sortierkriterium) erklärt ist. Die genaue Definition der Ordnungsrelation wird in der Implementation des Objekttyps `Tentry` verborgen. Zum Vergleich zweier Objekte dieses Typs wird der `INTERFACE`-Teil der Unit `element` um den Prozedurkopf einer Funktion `compare` erweitert, deren Implementierung im `IMPLEMENTATION`-Teil der Unit `element` liegt:

```
TYPE Tordnung = (kleiner, gleich, groesser);

FUNCTION compare (obj1_ptr : Pentry;
                  obj2_ptr : Pentry) : Tordnung;
{ liefert den Wert  kleiner,  falls das Objekt, auf das obj1_ptr
                              zeigt kleiner als das Objekt ist,
                              auf das obj_ptr zeigt;
                    gleich,   bei Übereinstimmung der Zustände
                              beider Objekte;
                    groesser, falls das Objekt, auf das obj1_ptr
                              zeigt größer als das Objekt ist,
                              auf das obj_ptr zeigt             }
```

Auch hier verändern einige Operationen den gegenwärtigen Zustand eines Objekts der Datenstruktur, so daß wieder eine Operation ***show*** definiert wird, die den gegenwärtigen Zustand eines Objekts der Datenstruktur anzeigt.

10.2.1 Prioritätsschlange

Auf einer Prioritätsschlange sind die Operationen ***init***, ***insert***, ***delete***, ***is_member*** und ***min*** definiert (vgl. Abbildung 10-2).

Typische Anwendungen einer Prioritätsschlange sind das Führen einer Symboltabelle in einem Compiler (das Ordnungskriterium ist die lexikographische Reihenfolge der Namen), die Verwaltung von Datensätzen in einer Datei, die durch einen Primärschlüssel (Ordnungskriterium) identifizierbar sind oder die Verwaltung prioritätengesteuerter Warteschlangen in einem Betriebssystem (das Ordnungskriterium ist die Priorität).

Die Operationen auf einer Prioritätsschlange sollen in ihrem Zeitaufwand nicht von den *Zuständen* der Objekte; es kann jedoch eine Abhängigkeit zwischen dem Zeitaufwand zur Durchführung einer Operation und der Anzahl der in der Datenstruktur vorhandenen Elemente bestehen.

Bei der Realisierung einer Prioritätsschlange kann man die Ordnungsrelation auf den einzelnen Elementen nutzen. Beispielsweise bietet sich an, die Prioritätsschlange als geordnete (lückenlose) Tabelle mit aufsteigendem Ordnungskriterium in den Elementen zu implementieren, vorausgesetzt, die maximale Anzahl an Elementen ist von vornherein bekannt. Die Realisierung der Operation ***min*** greift dann gerade auf den ersten Tabelleneintrag zu. Die Methoden, die die Operationen ***insert*** und ***delete*** implementieren, müssen sicherstellen, daß auch nach ihrer Ausführung alle Tabelleneinträge lückenlos in der Tabelle stehen und daß die Ordnung der Elemente in der Tabelle erhalten bleibt. Die Methode für die Operation ***insert*** muß also das neu einzutragende Element an die richtige Position innerhalb der bereits in der Prioritätsschlange vorhandenen Elemente einfügen. Daher werden eventuell eine Reihe von Tabelleneinträgen, nämlich diejenigen, die einen größeren Ordnungswert aufweisen als das hinzuzufügende Element, auf die jeweils nächste Position verschoben. Eine Verschiebung von Tabelleneinträgen, jetzt um eine Position nach vorn, ist in der Regel auch bei der zur Operation ***delete*** gehörenden Methode erforderlich. Die Methode zur Realisierung der Operation ***is_member*** kann von der Ordnung der Tabelleneinträge Gebrauch machen, indem eine Binärsuche (siehe Kapitel 7.4) auf der Tabelle durchgeführt wird.

Wegen des zeitlichen Aufwands bei der Durchführung mehrerer ***insert***- und ***delete***-Operationen in dieser Realisierung, der sich durch die Verschiebung der Tabelleneinträge ergibt, sind jedoch andere Implementationen einer Prioritätsschlange vorzuziehen. Im folgenden soll eine Prioritätsschlange als binärer Suchbaum realisiert werden. Hierbei findet ein Ausgleich zwischen dem Aufwand aller Operationen statt, die für eine Prioritätsschlange definiert sind. Andere Methoden werden in der angegebenen Literatur beschrieben.

Ein **binärer Suchbaum** besteht aus einem Binärbaum mit durch Werte des Ordnungskriteriums markierten Knoten und gerichteten Kanten. Der Struktur des Baums berücksichtigt die Ordnungsrelation ≤ der Elemente. Es gilt für die Knotenmarkierung m jedes Knotens K: Alle Knotenmarkierungen m_l im ***linken Teilbaum*** unterhalb von K (das ist der Teilbaum, dessen Wurzel der linke Nachfolger von K ist) erfüllen die Bedingung $m_l \leq m$, und alle Knotenmarkierungen m_r im ***rechten Teilbaum*** unterhalb von K (das ist der Teilbaum, dessen Wurzel der rechte Nachfolger von K ist) erfüllen die Bedingung $m_r > m$.

Die Anzahl der gerichteten Kanten, die von der Wurzel beginnend durchlaufen werden müssen, um einen Knoten zu erreichen, heißt der **Rang** des Knotens. Die **Höhe des Baums** wird durch den maximalen Rang + 1 definiert. Ein Baum, der nur aus der Wurzel besteht, hat demnach die Höhe 1.

Abbildung 10.2.1-1 zeigt einen binären Suchbaum. Die Knotenmarkierungen sind hier Elemente vom Typ `INTEGER`. Die Pfeile stellen die Kanten dar. Ihre Positionierung an den Knoten beschreibt die Nachfolgerrelation der Knoten (linker Nachfolger, rechter Nachfolger).

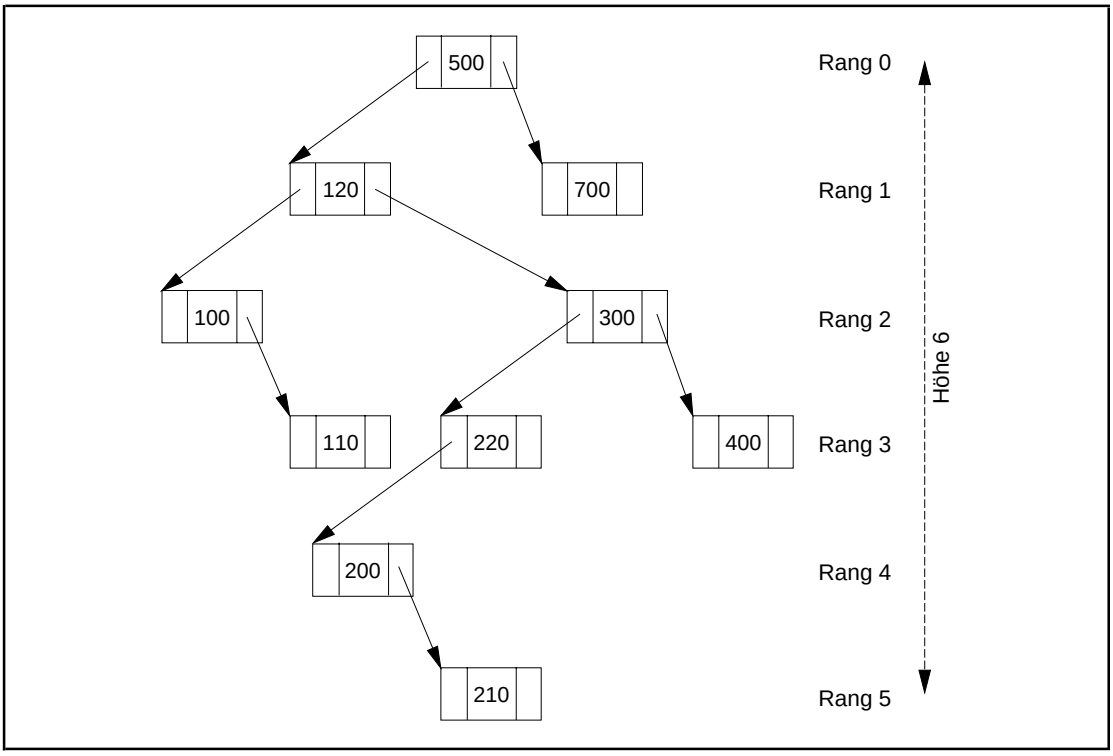


Abbildung 10.2.1-1: Binärer Suchbaum

Um eine Prioritätsschlange als binären Suchbaum zu realisieren, wird der Objekttyp `TPrio` deklariert. Ein Objekt dieses Typs bildet eine dynamisch erzeugte Verweisstruktur in Form eines binären Suchbaums, wobei die Knotenmarkierungen nicht die in ihr enthaltenen Elemente selbst, sondern (wie beim Objekttyp `TListe`) Verweise auf die entsprechenden Objekte sind. Mit der Bezeichnung *Element eines Knotens K* ist im folgenden das Element gemeint, auf das die Markierung im Knoten *K* verweist. Die Ordnungsrelation der Elemente wird dann offensichtlich auf die Knoten des binären Suchbaums übertragen, der zur Implementierung einer Prioritätsschlange aufgebaut wird.

Abbildung 10.2.1-2 zeigt ein Objekt vom Typ `TPrio` zur Implementierung der Elemente im binären Suchbaum von Abbildung 10.2.1-1. Das Ordnungskriterium der Elemente (Objekte) ist hier wieder vom Typ `INTEGER` und in den jeweiligen Objekten vermerkt.

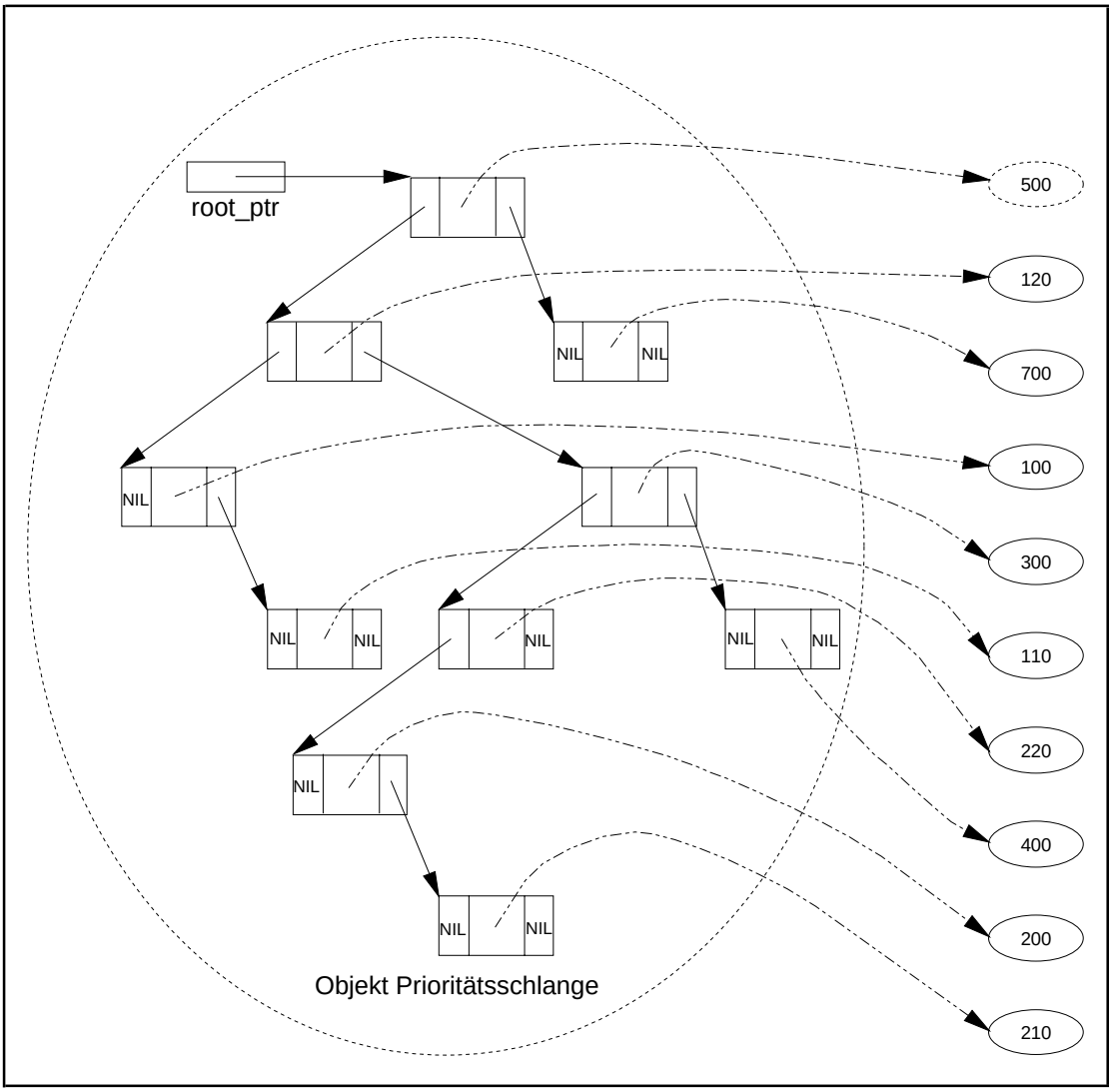


Abbildung 10.2.1-2: Prioritätsschlange (Beispiel)

Die folgende Unit `Prio` implementiert die Datenstruktur Prioritätsschlange. Die Benutzerschnittstelle lautet:

Methode	Bedeutung
`CONSTRUCTOR TPrio.init;`	Die Prioritätsschlange wird leer initialisiert.
`PROCEDURE TPrio.insert (entry_ptr : Pentry);`	Ein neues Element wird in die Prioritätsschlange aufgenommen.
Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	
`PROCEDURE TPrio.delete (entry_ptr : Pentry);`	Das adressierte Element wirdn aus der Prioritätsschlange entfernt.
Bedeutung des Parameters: `entry-ptr` Verweis auf das zu entfernende Element	
`FUNCTION TPrio.is_member (entry_ptr : Pentry) : BOOLEAN;`	Die Funktion liefert den Wert TRUE, falls das durch entry_ptr adressierte Element in der Prioritätsschlange vorkommt, ansonsten den Wert FALSE.
Bedeutung des Parameters: `entry-ptr` Zeiger auf das zu überprüfende Element	
`FUNCTION TPrio.min : Pentry;`	Die Funktion liefert einen Verweis auf das kleinste Element, das zur Zeit in der Prioritätsschlange vorkommt.
`DESTRUCTOR TPrio.done;`	Die Prioritätsschlange wird aus dem System entfernt.

Die Methode `TPrio.insert` zur Realisierung der ***insert***-Operation durchläuft den binären Suchbaum bei der Wurzel beginnend. Dabei wird bei jedem Knoten *K* geprüft, ob der Wert des Ordnungskriteriums im neu aufzunehmenden Element kleiner oder gleich dem Wert des Ordnungskriteriums des Elements des Knotens *K* ist oder nicht. Im ersten Fall wird zum linken Nachfolger von *K* verzweigt, im zweiten Fall zum rechten, bis ein Knoten gefunden ist, der an der betreffenden Nachfolgerstelle keinen Nachfolger besitzt. Hier wird ein neuer Knoten angehängt und mit einem Verweis auf das neu aufzunehmende Element markiert. Die Struktur des binären Suchbaums, der durch mehrere hintereinander ausgeführte `TPrio.insert`-Aufrufe entsteht, hängt wesentlich von der Reihenfolge der auftretenden Elemente und ihrer Ordnungskriterien ab.

In der Methode `TPrio.delete` zur Realisierung der ***delete***-Operation wird zunächst der Knoten gesucht, dessen Markierung auf das zu entfernende Element verweist. Falls ein derartiger Knoten existiert, wird geprüft, welche der drei möglichen Situationen, die in Abbildung 10.2.1-3 dargestellt sind, an diesem Knoten herrscht. Entsprechend wird anschließend ein Knoten aus dem binären Suchbaum gelöscht.

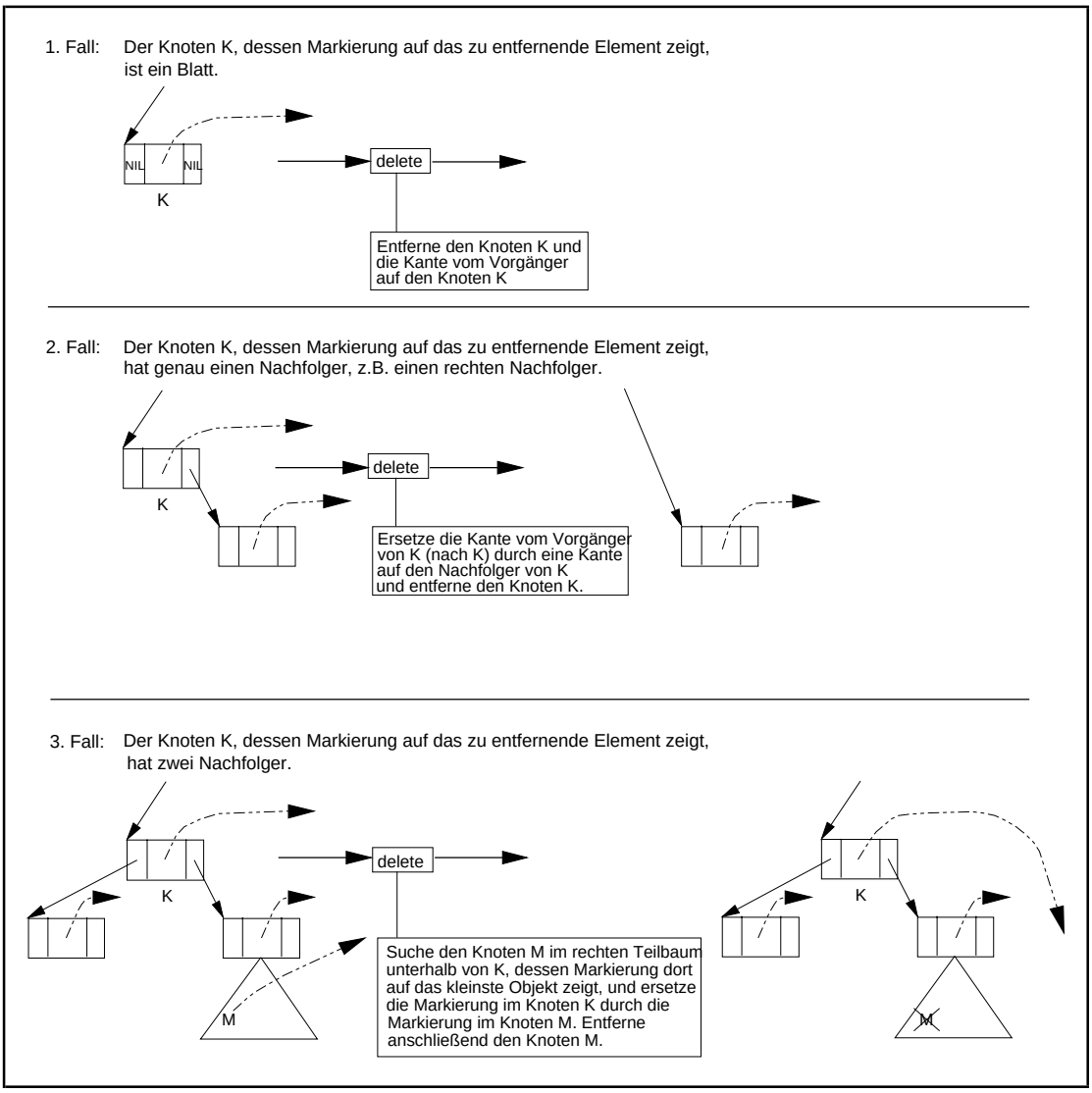


Abbildung 10.2.1-3: Methode TPrio.delete

Abbildung 10.2.1-4 zeigt die Prioritätsschlange aus Abbildung 10.2.1-3 nach einem Aufruf von `TPrio.delete` mit dem Element mit Ordnungskriteriums-Wert 120. Hier liegt der 3. Fall vor: Nachdem der Knoten *K* des zu entfernenden Elements gefunden worden ist, wird der Knoten *M*, dessen Markierung auf das kleinste Objekt im rechten Teilbaum unterhalb *K* zeigt, gesucht; die im Knoten *M* stehende Markierung ersetzt die Markierung in *K*. Anschließend wird *M* entfernt. Offensichtlich wird durch `TPrio.delete` nicht notwendigerweise der Knoten entfernt, dessen Markierung auf das bezeichnete Element zeigt.

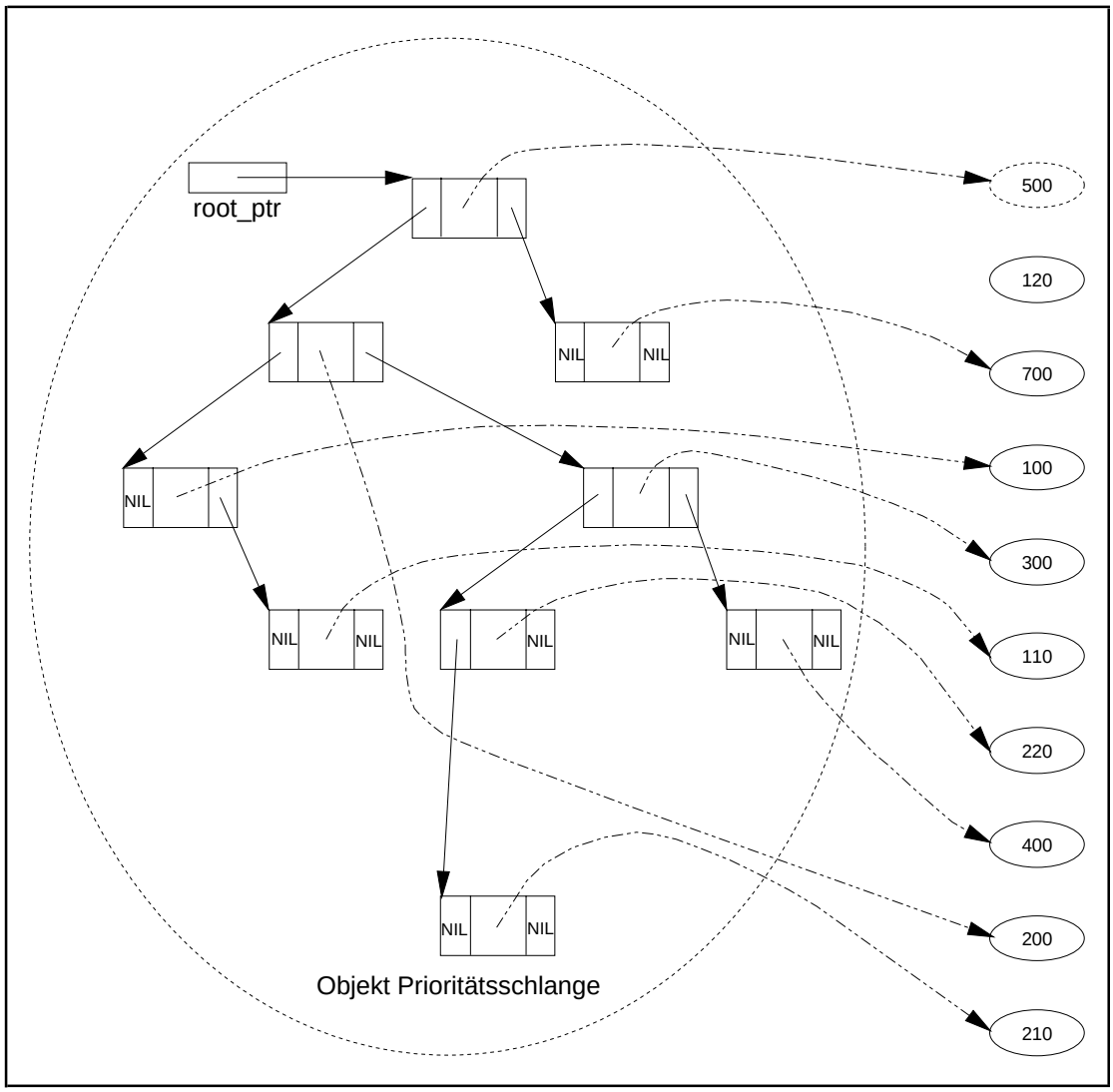


Abbildung 10.2.1-4: Ergebnis eines Aufrufs der Methode `TPrio.delete` (Beispiel)

Die zur ***is_member***-Operation gehörende Methode TPrio.is_member durchläuft den binären Suchbaum an der Wurzel beginnend ähnlich wie die Methode `TPrio.delete`. Dabei wird bei jedem Knoten geprüft, ob die dortige Markierung auf das im Aufruf der Methode `TPrio.is_member` spezifizierte Element zeigt oder nicht. Im ersten Fall wird das Vorhandensein des Elements festgestellt, im zweiten Fall wird zum linken Nachfolger verzweigt, wenn der Wert des Ordnungskriteriums des zu findenden Elements kleiner als das Ordnungskriterium im Element des gerade betrachteten Knotens ist, bzw. zum rechten Nachbarn, wenn er größer ist. Trifft man auf einen Knoten, der an der aufzusuchenden Nachfolgerstelle keinen Nachfolger besitzt, so ist das zu findende Element in der Prioritätsschlange nicht vorhanden.

Für die Realisierung der ***min***-Operation wird die Beobachtung genutzt, daß der Knoten, dessen Markierung auf das Element mit minimalem Wert des Ordnungskriteriums verweist, derjenige Knoten ist, der keinen linken Nachfolger mehr hat, wenn man den Baum bei der Wurzel beginnend jeweils zum linken Nachfolger verzweigend durchläuft.

Eine Methode zur Realisierung der Operation ***show*** ist zunächst nicht implementiert; sie wird allgemeiner in Kapitel 10.2.2 behandelt.

```
UNIT prio;

INTERFACE

   USES element;

   TYPE PPrio = ^TPrio;
        TPrio = OBJECT
                  PRIVATE
                    root_ptr : Pointer;
                  PUBLIC
                    CONSTRUCTOR init;
                    PROCEDURE insert (entry_ptr : Pentry);
                    PROCEDURE delete (entry_ptr : Pentry);
                    FUNCTION is_member (entry_ptr : Pentry)
                                       : BOOLEAN;
                    FUNCTION min : Pentry;
                    DESTRUCTOR done; VIRTUAL;
                END;


IMPLEMENTATION

TYPE PPrioentry = ^TPrioentry;
     TPrioentry = RECORD
                    left   : PPrioentry;
                    marker : Pentry;
                    right  : PPrioentry;
                  END;


CONSTRUCTOR TPrio.init;

  BEGIN { TPrio.init }
    root_ptr := NIL;
  END   { TPrio.init };


PROCEDURE TPrio.insert (entry_ptr : Pentry);

  PROCEDURE insert_ext (entry_ptr     : Pentry;
                        VAR tree_ptr  : PPrioentry);
  { fügt das Element, auf das entry_ptr zeigt, in den
    binären Suchbaum ein, der bei tree_ptr^ beginnt   }

    BEGIN { insert_ext }
      IF tree_ptr = NIL
      THEN BEGIN { einen neuen Knoten erzeugen und einfügen }
             New (tree_ptr);
             tree_ptr^.left   := NIL;
             tree_ptr^.marker := entry_ptr;
             tree_ptr^.right  := NIL;
           END
      ELSE       { Teilbaum suchen, in den der neue Knoten
                 eingefügt werden soll                    }
             IF compare (tree_ptr^.marker, entry_ptr) = kleiner
             THEN insert_ext (entry_ptr, tree_ptr^.right)
             ELSE insert_ext (entry_ptr, tree_ptr^.left);
    END   { insert_ext };
```

```
  BEGIN { TPrio.insert }
    insert_ext (entry_ptr, PPrioentry(root_ptr));
  END   { TPrio.insert };


PROCEDURE TPrio.delete (entry_ptr : Pentry);

  PROCEDURE delete_ext (entry_ptr : Pentry;
                        tree_ptr  : PPrioentry);
  { entfernt das Element, auf das entry_ptr zeigt, aus dem
    binären Suchbaum, der bei tree_ptr^ beginnt            }

  VAR mem_ptr  : PPrioentry;
      pred_ptr : PPrioentry;
      next_ptr : PPrioentry;

    BEGIN { delete_ext }
      { Knoten suchen, dessen Markierung auf das zu
        entfernende Element verweist, und dessen Vorgänger:}
      IF tree_ptr = NIL
      THEN { leerer Baum }
           mem_ptr := NIL
      ELSE BEGIN
             mem_ptr  := tree_ptr;
             pred_ptr := NIL;
             WHILE (mem_ptr <> NIL)
                   AND (mem_ptr^.marker <> entry_ptr) DO
               BEGIN
                 pred_ptr := mem_ptr;
                 IF compare (entry_ptr, mem_ptr^.marker) <> groesser
                 THEN { im linken Teilbaum weitersuchen }
                      mem_ptr := mem_ptr^.left
                 ELSE { im rechten Teilbaum weitersuchen }
                      mem_ptr := mem_ptr^.right;
               END;
           END;

      IF mem_ptr <> NIL
      THEN BEGIN { mem_ptr zeigt auf den Knoten des zu entfernenden
                   Elements, pred_ptr auf dessen Vorgänger bzw. hat
                   den Wert NIL, wenn das Element der Wurzel zu
                   entfernen ist                                   }
           IF (mem_ptr^.left = NIL) OR (mem_ptr^.right = NIL)
           THEN BEGIN { 1. Fall oder 2. Fall }
                  IF mem_ptr^.right <> NIL
                  THEN next_ptr := mem_ptr^.right
                  ELSE next_ptr := mem_ptr^.left;
                  IF pred_ptr = NIL
                  THEN { Das Element der Wurzel ist zu entfernen }
                       root_ptr := next_ptr
                  ELSE BEGIN
                         IF mem_ptr = pred_ptr^.left
                         THEN pred_ptr^.left  := next_ptr
                         ELSE pred_ptr^.right := next_ptr;
                       END;
                  { Speicherplatz freigeben }
                  Dispose (mem_ptr);
                END   { 1. Fall oder 2. Fall }
           ELSE BEGIN { 3. Fall }
                  { Ermittlung des Knotens im rechten Teilbaum
                    unterhalb mem_ptr mit dem kleinsten Wert des
                    Ordnungskriteriums (Verweis in next_ptr)     }
                  next_ptr := mem_ptr^.right;
                  WHILE next_ptr^.left <> NIL DO
                    next_ptr := next_ptr^.left;
                  { Übernahme der dortigen Knotenmarkierung }
                  mem_ptr^.marker := next_ptr^.marker;
```

```
{ Entfernung des überflüssigen Knotens aus dem
                      rechten Teilbaum unterhalb mem_ptr          }
                    delete_ext (next_ptr^.marker, mem_ptr^.right);
                  END   { 3. Fall };
            END;
    END   { delete_ext };

  BEGIN { TPrio.delete }
    delete_ext (entry_ptr, root_ptr);
  END   { TPrio.delete };


FUNCTION TPrio.is_member (entry_ptr : Pentry) : BOOLEAN;

  FUNCTION is_member_ext (entry_ptr : Pentry;
                          tree_ptr  : PPrioentry) : BOOLEAN;
  { sucht das Element entry in dem binären Suchbaum,
    der bei tree_ptr^ beginnt                          }

    BEGIN { is_member_ext }
      IF tree_ptr <> NIL
      THEN BEGIN
             IF tree_ptr^.marker = entry_ptr
             THEN is_member_ext := TRUE
             ELSE IF compare (tree_ptr^.marker, entry_ptr) = kleiner
                  THEN is_member_ext
                         := is_member_ext (entry_ptr, tree_ptr^.right)
                  ELSE is_member_ext
                         := is_member_ext (entry_ptr, tree_ptr^.left);
           END
      ELSE is_member := FALSE;
    END   { is_member_ext };

  BEGIN { is_member }
    is_member := is_member_ext (entry_ptr, root_ptr);
  END   { is_member };


FUNCTION TPrio.min : Pentry;

VAR p : PPrioentry;

  BEGIN { TPrio.min }
    IF root_ptr = NIL
    THEN { leerer Baum } min := NIL
    ELSE BEGIN { im linken Teilbaum weitersuchen, bis es keinen
                 linken Nachfolger mehr gibt                    }
           p := root_ptr;
           WHILE p^.left <> NIL DO
             p := p^.left;
           min := p^.marker
         END;
  END   { TPrio.min };

DESTRUCTOR TPrio.done;

  BEGIN { TPrio.done }
    WHILE root_ptr <> NIL
      DO delete (PPrioentry(root_ptr)^.marker);
  END   { TPrio.done };

END { prio }.
```

Der Vorteil der Implementation einer Prioritätsschlange durch einen binären Suchbaum liegt in der Einfachheit der Methoden. Bei gleichverteilten Werten des Ordnungskriteriums der Elemente ist zu erwarten, daß die Struktur des binären Suchbaums ausgeglichen ist, d.h. daß möglichst viele Knoten auch zwei Nachfolger besitzen. Das kann natürlich nicht garantiert werden. Werden z.B. Elemente mit aufsteigenden bzw. absteigenden Werten des Ordnungskriteriums nacheinander in den binären Suchbaum aufgenommen, so entsteht eine lineare Kette von Knoten. Nachfolgende Operationen haben dann eine Laufzeit der Ordnung $O(n)$, falls n Elemente in der Struktur sind. Weiterhin wird bei der hier behandelten Implementierung angenommen, daß die Elemente selbst im Arbeitsspeicher liegen, daß also der Zugriff über den Verweis in einem Knoten "schnell" erfolgt. Liegen jedoch die Elemente beispielsweise in einem Peripheriespeicher, so sind Plattenzugriffe erforderlich, und die Adreßvergleiche zur Feststellung der Gleichheit zweier Elemente erscheint problematisch. Daher eignet sich ein binärer Suchbaum nur gut zur Implementierung "interner" Prioritätsschlangen.

10.2.2 Durchlaufen eines Baums

In Kapitel 10.2.1 wurde ein binärer Suchbaum zur Implementation einer Datenstruktur behandelt. In diesem Kapitel werden verschiedene Verfahren vorgestellt, die die Knoten eines Baums durchlaufen, d.h. die Knoten nacheinander besuchen, und dabei die Knotenmarkierungen bzw. Werte anzeigen, auf die die Knotenmarkierungen verweisen (Operation ***show***). Die Ordnungsstruktur der Elemente ist in allen Fällen von sekundärer Bedeutung; es kommt allein auf die Verweisstruktur zwischen den Knoten des Baums an. Die im folgenden behandelten Verfahren sind auf allgemeinere Strukturen wie gerichtete und ungerichtete Graphen übertragbar. Beispielsweise kann das Prinzip eines Algorithmus, der nacheinander alle Knoten eines gerichteten Graphs besucht, wie folgt lauten (vgl. [O/W]):

```
PROCEDURE besuche ( G = (V, E) );
{ besucht nacheinander alle Knoten des Graphen G:
  Dazu werden die einzelnen Kanten nacheinander als
  "benutzt" markiert                                  }

VAR B : SET OF knotentyp;  { Menge der besuchten Knoten }

  BEGIN { besuche }
    B := { b };    { b ist ein erster besuchter Knoten }
    markiere alle Kanten in E als unbenutzt;
    WHILE (es gibt noch unbenutzte Kanten (v,v') ∈ E mit v ∈ B) DO
      BEGIN
        markiere (v,v') als benutzt;
        B := B ∪ {v'};
      END;
  END   { besuche };
```

Es ist noch nicht festgelegt, in welcher Reihenfolge die unbenutzten Kanten ausgewählt werden und welcher Knoten als erster besucht wird. Das beschriebene Prinzip läßt sich auf viele praktische Aufgaben (auch aus der systemnahen Welt) anwenden wie die

Bestimmung von Strukturinformationen (Zusammenhangskomponenten) in Computernetzen nach Ausfall einzelner Rechner, kürzeste Wege in Graphen, Flüsse in Netzwerken usw. Im folgenden soll eine Spezialisierung auf Binärbäume erfolgen.

Die zu durchlaufende Datenstruktur sei ähnlich definiert wie der binäre Suchbaum in Kapitel 10.2.1, jetzt in der Unit `tree`. Der Objekttyp eines Baums lautet (im `INTERFACE`-Teil):

```
TYPE PTree = ^TTree;
     TTree = OBJECT
               PRIVATE
                 root_ptr : Pointer;
               PUBLIC
                 CONSTRUCTOR init;
                 PROCEDURE insert (entry_ptr : Pentry);
                 PROCEDURE delete (entry_ptr : Pentry);
                 FUNCTION is_member (entry_ptr : Pentry)
                                    : BOOLEAN;
                 PROCEDURE show_praefix; VIRTUAL;
                 PROCEDURE show_symmetric; VIRTUAL;
                 PROCEDURE show_postfix; VIRTUAL;
                 DESTRUCTOR done; VIRTUAL;
             END;
```

Ein Knoten wird durch den Typ

```
TYPE PTree_entry = ^TTree_entry;
     TTree_entry = RECORD
                     left   : PTree_entry;
                     marker : Pentry;
                     right  : PTree_entry;
                   END;
```

beschrieben. Er hat einen Verweis auf zwei Nachfolger (einen linken und einen rechten Nachfolger), auf einen Nachfolger (einen linken oder einen rechten Nachfolger) oder keinen Nachfolger. In der Komponente steht ein Verweis auf ein Element, das zu diesem Knoten gehört (vgl. Abbildung 10.2.1-2). Die Wurzel des Baums ist über eine Pointervariable `root_ptr` erreichbar. Weitere Bedingungen über die Knotenmarkierungen (wie die Ordnungsrelation der Elemente, die zu den Knoten im linken bzw. rechten Teilbaum unterhalb eines Knotens gehören, wie bei einem binären Suchbaum) werden nicht berücksichtigt. Methoden zur Initialisierung der Struktur, zum Einfügen weiterer Knoten in die Struktur bzw. zum Entfernen von Knoten aus der Struktur werden im folgenden nicht weiter betrachtet. Der Schwerpunkt der Darstellung liegt auf Methoden zur Realisierung des Durchlaufens eines Baums und der Operation ***show***.

Eine Methode zur ***show***-Operation muß den gesamten Baum durchlaufen, wobei der Weg nur über die gerichteten Kanten erfolgen kann. Bei jedem Knoten wird das Element angezeigt, auf das die jeweilige Knotenmarkierung verweist. Dabei kann die "rekursive" Struktur eines Baums genutzt werden: Ein nicht-leerer Baum besteht ja entweder nur aus der Wurzel *W* oder aus der Wurzel *W* und einem linken oder rechten Teilbaum, dessen Wurzel der linke bzw. rechte Nachfolger von *W* ist. Dabei kann einer dieser Teilbäume auch leer sein. Entsprechend wird man bei der Methode die Wurzel aufsuchen und die Methode dann (rekursiv) auf den linken bzw. rechten Teilbaum unterhalb der Wurzel

anwenden. Je nachdem, in welcher Reihenfolge man in die Teilbäume unterhalb eines Knotens *W* geht und das Element von *W* anzeigt, ergibt sich eine andere Reihenfolge aller angezeigten Elemente. In der Unit `tree` sind drei mögliche Realisierungen der Operation ***show*** angegeben. Man bezeichnet die Arbeitsweisen als **Durchlaufen des Baums in Präfixordnung**, **symmetrischer Ordnung** bzw. **Postfixordnung**. Entsprechend sind die Methoden `TTree.show_praefix`, `TTree.show_symmetric` bzw. `TTree.show_postfix` bezeichnet und im `IMPLEMENTATION`-Teil der Unit `tree` definiert (Abbildung 10.2.2-1). Alle drei Methoden führen auf dem Binärbaum eine **Tiefensuche** (depth-first-search) durch.

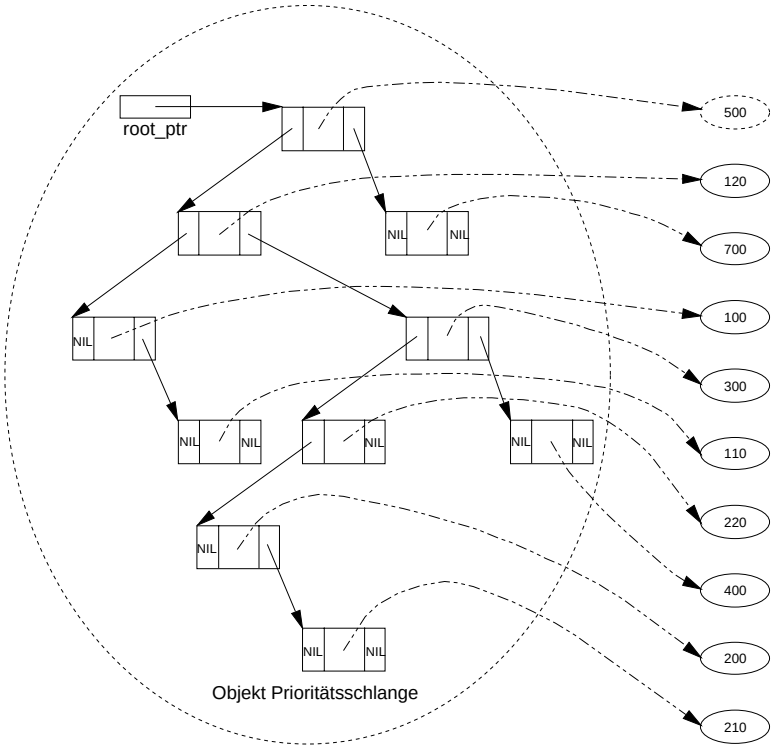
Eine andere Reihenfolge, um die Knotenmarkierungen anzuzeigen, spiegelt die Hierarchie in einem Baum wider: Zuerst wird das Element der Wurzel (Rang 0) angezeigt, anschließend die Elemente aller Nachfolger der Wurzel, d.h. aller Knoten mit Rang 1. Allgemein wird zu den Knoten des Rangs *i* übergegangen, nachdem alle Knoten des Rangs *i*-1 besucht wurden ($1 \leq i <$ Höhe des Baums). Diese Art des Durchlaufen eines Baums heißt **Breitensuche** (breadth first search).

Anzeigen der Knotenmarkierungen in Präfixordnung

```
PROCEDURE TTree.show_praefix;

  PROCEDURE show (ptr : PTree_entry);
  { Anzeigen aller Knoten in Präfixordnung, die im Baum stehen,
    auf dessen Wurzel ptr zeigt                                  }

    BEGIN { show }
      IF ptr <> NIL
      THEN BEGIN
             ptr^.marker^.display;
             show (ptr^.left);
             show (ptr^.right);
           END;
    END   { show };

  BEGIN { show_praefix }
    show (root_ptr);
  END   { show_praefix };
```

Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums in Abbildung 10.2.1-2:

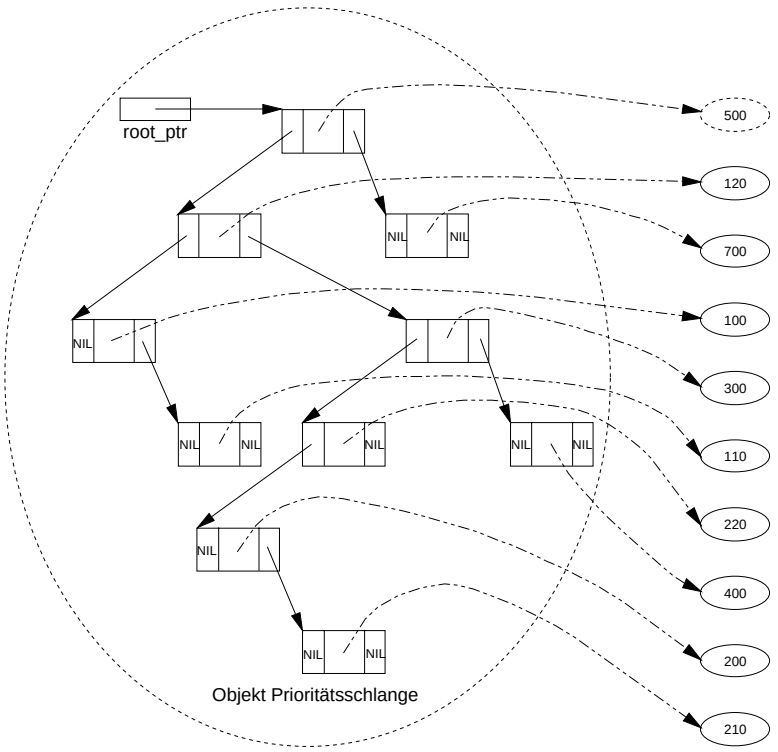
500, 120, 100, 110, 300, 220, 200, 210, 400, 700



Anzeigen der Knotenmarkierungen in symmetrischer Ordnung

```
PROCEDURE TTree.show_symmetric;

  PROCEDURE show (ptr : PTree_entry);
  { Anzeigen aller Knoten in symmetrischer Ordnung,
    die im Baum stehen, auf dessen Wurzel ptr zeigt           }

    BEGIN { show }
      IF ptr <> NIL
      THEN BEGIN
             show (ptr^.left);
             ptr^.marker^.display;
             show (ptr^.right);
           END;
    END   { show };

  BEGIN { show_symmetric }
    show (root_ptr);
  END   { show_symmetric };
```

Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums in Abbildung 10.2.1-2:



100, 110, 120, 200, 210, 220, 300, 400, 500, 700

Anzeigen der Knotenmarkierungen in Postfixordnung

```
PROCEDURE TTree.show_postfix;

  PROCEDURE show (ptr : PTree_entry);
  { Anzeigen aller Knoten in Postfixordnung, die im Baum stehen,
    auf dessen Wurzel ptr zeigt                                    }

    BEGIN { show }
      IF ptr <> NIL
      THEN BEGIN
             show (ptr^.left);
             show (ptr^.right);
             ptr^.marker^.display;
           END;
    END   { show };

  BEGIN { show_postfix }
    show (root_ptr);
  END   { show_postfix };
```

Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums in Abbildung 10.2.1-2:

110, 100, 210, 200, 220, 400, 300, 120, 700, 500

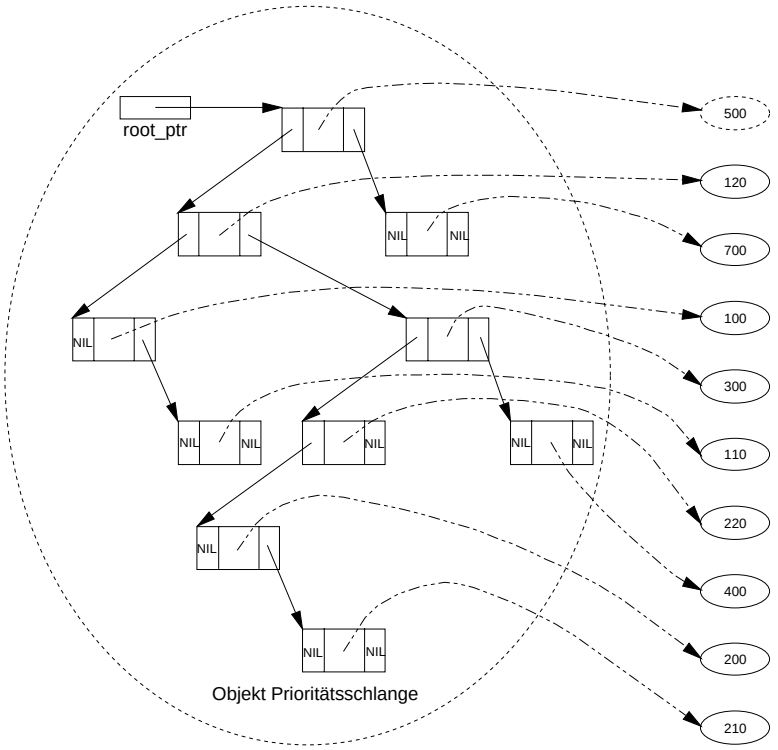


Abbildung 10.2.2-1: Tiefensuche auf einem Binärbaum

Das Durchlaufen eines Baums gemäß Breitensuche kann die Verweisstruktur zwischen den Knoten nicht direkt nutzen, da es keine Verweise zu den Nachbarknoten auf gleichem Niveau gibt. Zur Erläuterung der Breitensuche werde angenommen, daß der zu durchlaufende Binärbaum vollständig ist, d.h. alle Niveaus des Baums sind vollständig

ausgefüllt. Während des Ablaufs der Breitensuche werden sukzessive Verweise auf Knoten des Baums in eine ***Agenda***, ein Objekt vom Objekttyp FIFO-Warteschlange (`TYPE TFIFO`), eingefügt und entfernt:

Es wird die Wurzel bearbeitet und Verweise auf den oder die Nachfolger der Wurzel in die Agenda eingetragen. Solange die Agenda nicht leer ist, werden die folgenden drei Schritte durchgeführt:

(1) Ein Verweis auf einen Knoten *K* wird der Agenda entnommen;

(2) der Knoten *K* wird bearbeitet, d.h. das zu *K* gehörende Element wird angezeigt.

(3) falls *K* einen oder zwei Nachfolger besitzt, werden Verweise auf den bzw. die Nachfolger von *K* in die Agenda eingetragen;

Denkt man sich die Knoten des Baums bei der Wurzel beginnend nach aufsteigenden Niveaus und auf jedem Niveau von links nach rechts durchnumeriert (Startnummer ist die Nummer 1), so enthält die Agenda (zwischen last und first und falls sie nicht bereits leer ist) zu jedem Zeitpunkt Verweise auf Knoten mit lückenlos aufsteigenden Nummern $i, i+1, ..., i+k$. Befindet sich in Schritt (2) der Knoten auf dem Niveau j und hat die Nummer i, dann werden in Schritt (3) die Knoten mit den Nummern $2i$ und $2i+1$ auf dem Niveau $j+1$ in die Agenda eingetragen (Abbildung 10.2.2-2).

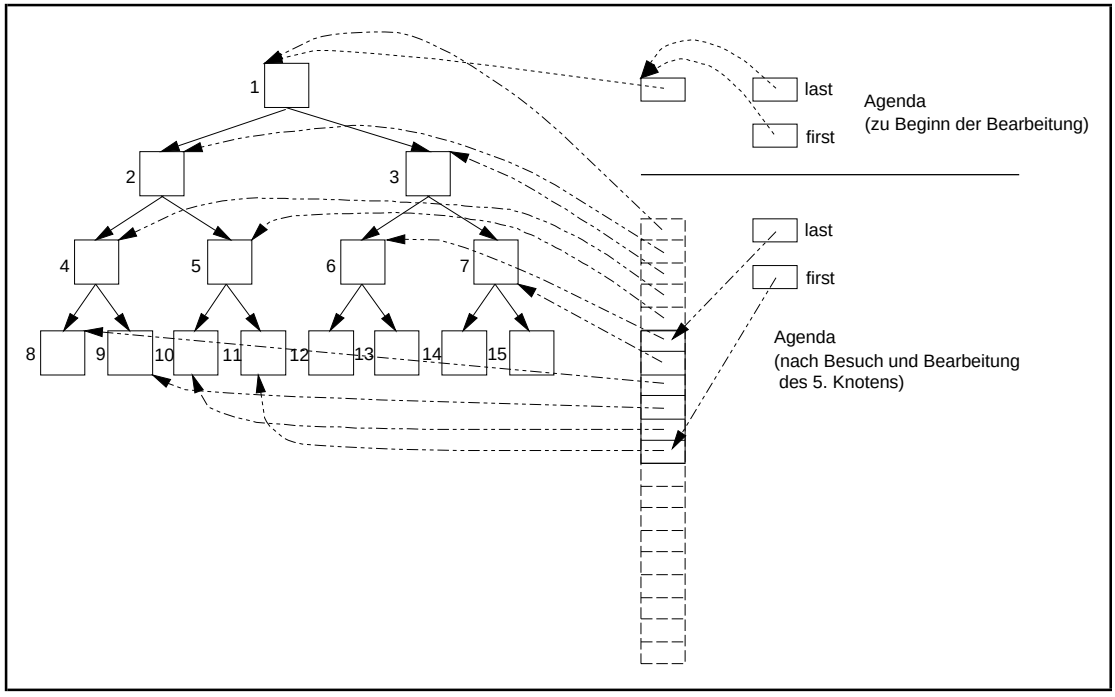


Abbildung 10.2.2-2: Breitensuche auf einem Binärbaum

Zur Realisierung der ***show***-Operation, die nach dem Prinzip der Breitensuche vorgeht, wird der zugehörige Methodenkopf

```
PROCEDURE show_br_search; VIRTUAL;
```

in die Objekttypdeklaration `TTree` aufgenommen. Innerhalb der Implementierung von `TTree.show_br_search` wird die Agenda als FIFO-Warteschlange geführt. Dazu kann die Unit `FIFO` aus Kapitel 10.1.2 verwendet werden. Allerdings ist ein Eintrag im Objekt FIFO-Warteschlange jetzt nicht ein Verweis auf ein Objekt vom Typ `Tentry`, sondern auf einen Knoten im Binärbaum. Die Datentypdeklaration

```
TYPE PTree_entry = ^TTree_entry;
     TTree_entry = RECORD
                     left   : PTree_entry;
                     marker : Pentry;
                     right  : PTree_entry;
                   END;
```

wird daher aus der Unit `tree` entfernt und in eine eigene Unit `treedef` gelegt, die von den Units `FIFO` und `tree` eingebunden wird:

```
UNIT treedef;

INTERFACE

   USES element;

   TYPE PTree_entry = ^TTree_entry;
        TTree_entry = RECORD
                        left   : PTree_entry;
                        marker : Pentry;
                        right  : PTree_entry;
                      END;

IMPLEMENTATION

END { tree }.
```

Außerdem wird in der Unit `FIFO` jeder Bezeicher `Pentry` durch `PTree_entry` ersetzt. Und schließlich wird die Unit `FIFO` in Unit `agenda` umbenannt, um zu verdeutlichen, daß es sich um eine spezielle Anwendung der Datenstruktur FIFO-Warteschlange handelt. In die Unit `tree` wird die Unit `agenda` in den `IMPLEMENTATION`-Teil angebunden. Die Unit-Abhängigkeiten zeigt Abbildung 10.2.2-3.

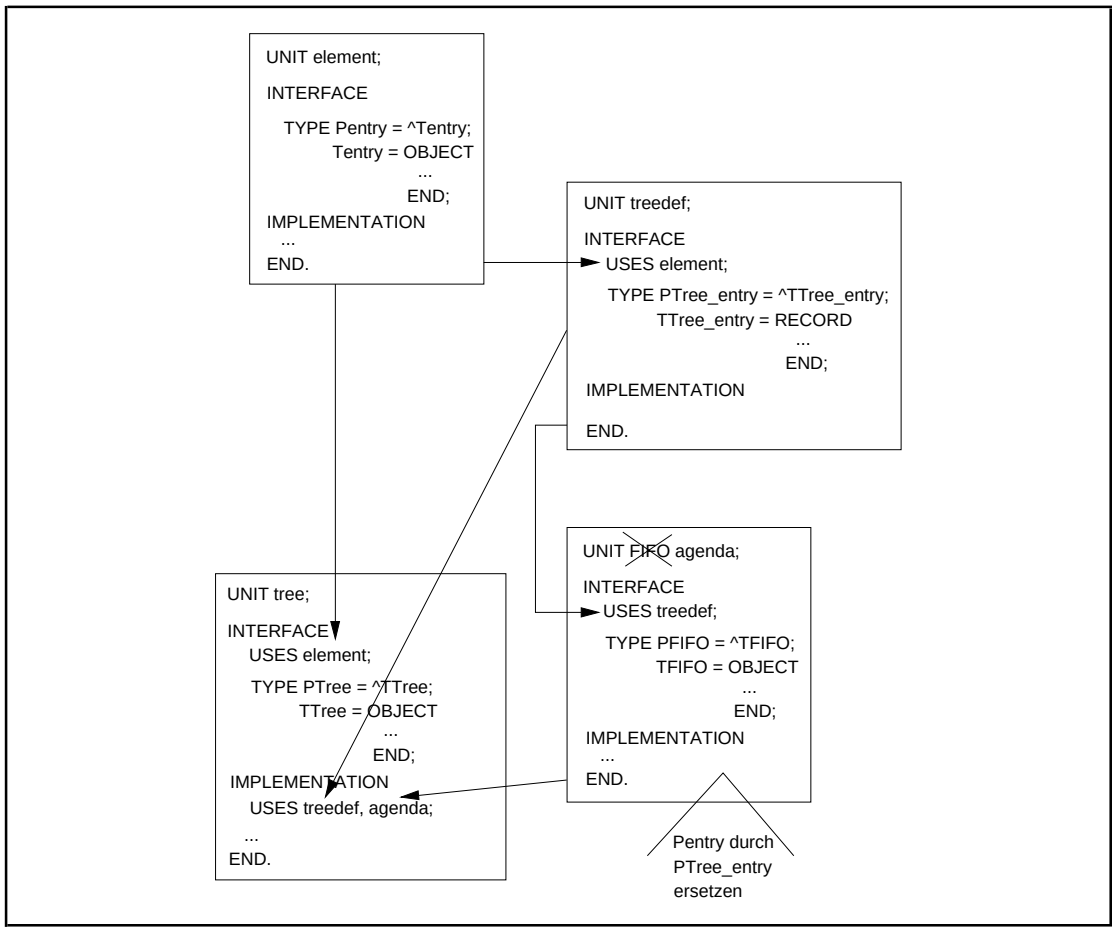


Abbildung 10.2.2-3: Unit-Abhängigkeiten bei der Breitensuche auf einem Binärbaum

Die Unit `tree` lautet nun (es ist nur der `INTERFACE`-Teil und der hier wesentliche Ausschnitt des `IMPLEMENTATION`-Teils gezeigt):

```
UNIT tree;

INTERFACE

   USES element;

   TYPE PTree = ^TTree;
        TTree = OBJECT
                  PRIVATE
                    root_ptr : Pointer;
                  PUBLIC
                      ...
                    PROCEDURE show_br_search; VIRTUAL;
                      ...
                END;


IMPLEMENTATION

USES treedef, agenda;

...
```

```
PROCEDURE TTree.show_br_search;

VAR agenda_ptr : PFIFO;
    node_ptr   : PTree_entry;

  BEGIN { TTree.show_br_search }
    IF root_ptr <> NIL
    THEN BEGIN
           { Agenda einrichten }
           New (agenda_ptr, init);
           { Wurzel des Baums bearbeiten und Verweise auf
             weitere Knoten in die Agenda stellen; dann die
             Agenda abarbeiten                              }
           node_ptr := root_ptr;
           WHILE node_ptr <> NIL DO
             BEGIN
               { Element des Knotens anzeigen }
               node_ptr^.marker^.display;
               { Nachfolgerknoten in die Agenda stellen }
               IF node_ptr^.left <> NIL
               THEN agenda_ptr^.insert (node_ptr^.left);
               IF node_ptr^.right <> NIL
               THEN agenda_ptr^.insert (node_ptr^.right);
               { einen weiteren Knoten aus der Agenda holen }
               agenda_ptr^.delete (node_ptr);
             END;
           { Agenda wieder entfernen }
           Dispose (agenda_ptr, done);
         END;
  END   { TTree.show_br_search };

...


END { tree }.
```

10.2.3 Höhenbalancierter Baum

Der Zeitaufwand zur Durchführung einer der behandelten Operationen auf einem Binärbaum hängt im wesentlichen von seiner Höhe ab. Im günstigsten Fall ist diese bei n Knoten proportional zu $\log n$, im ungünstigsten Fall proportional zu n (siehe Kapitel 13.3). Beispielsweise erfordert bei einem binären Suchbaum das Einfügen von n Elementen, ausgehend von einem leeren binären Suchbaum, im ungünstigsten Fall $O(n^2)$ viele Schritte. Es läßt sich zeigen ([AHU]), daß die mittlere Anzahl von Schritten, um n Elemente mit zufälligen Werten des Ordnungskriteriums[35] in einen anfangs leeren binären Suchbaum einzufügen, von der Ordnung $O(n \cdot \log n)$ ist. Weiter gilt, daß eine Folge von n ***insert***-, ***delete***-, ***is_member***- und ***min***-Operationen mit einem mittleren Zeitaufwand der Ordnung $O(n \cdot \log n)$ ausgeführt werden kann. Es erscheint daher erstrebenswert, eine Datenstruktur als Binärbaum so zu entwerfen, daß durch die mehrfache Ausführung der für sie definierten Operationen jeweils ein **höhenbalancierter Baum** entsteht, d.h. für den sich die Pfadlängen von der Wurzel zu den einzelnen Blättern möglichst wenig unterscheiden. Der Idealfall ist der **vollkommen höhenbalancierte Baum**, bei dem für jeden Knoten die Anzahl der Knoten in seinem

[35] Eine Folge von Werten a_1, ..., a_n ist eine zufällige Folge, wenn jedes a_i mit gleicher Wahrscheinlichkeit $1/n$ das j-kleinste ($1 \le j \le n$) ist.

linken Teilbaum um höchstens 1 von der Anzahl der Knoten im rechten Teilbaum differiert. Als Konsequenz folgt, daß sich bei einem vollkommen höhenbalancierten Baum für jeden Knoten die Höhen seiner von ihm ausgehenden Teilbäume um höchstens 1 unterscheiden.

Diese Folgerung liefert eine notwendige, aber nicht hinreichende Bedingung für einen vollkommen höhenbalancierten Baum. Ein **AVL-Baum** (benannt nach G.M. Adelson-Velskii und E.M. Landis) beispielsweise ist dadurch definiert, daß sich für jeden Knoten die Höhen der beiden von ihm ausgehenden Teilbäume um höchstens 1 unterscheiden; er ist aber keineswegs vollkommen höhenbalanciert, wie das Beispiel in Abbildung 13.3-2 zeigt.

Eine genaue Analyse der notwendigen Zugriffe, um bei einem binären Suchbaum mit n Knoten zu einem Knoten zu gelangen, zeigt die folgende Zusammenfassung.

	maximale Anzahl $M(n)$	durchschnittliche Anzahl $D(n)$
binärer Suchbaum	n	$2(n+1)/n\,H_n - 3$ mit $H_n = \sum_{k=1}^{n} 1/k$. Für große n gilt $D(n) \sim 1{,}39 \log_2 n$.
vollkommen höhenbalancierter binärer Suchbaum	$\lfloor \log_2(n+1) \rfloor$	$1/n\,(1 + (n+1)M(n) - 2^{M(n)})$. Für große n gilt $D(n) \sim \log_2 n - 1$.
AVL-Baum	$< 1{,}44 \log_2(n+2)$	wie beim vollkommen höhenbalancierten binären Suchbaum

In praktischen Anwendungsfällen besteht ein Element der Datenstruktur häufig aus einem Datensatz, der über den Wert eines Primärschlüsselfelds eindeutig identifiziert wird und eine Reihe weiterer meist umfangreicher Komponenten beinhaltet. Die Datenstruktur stellt in diesem Fall eine (physikalische) Datei dar. Bei einer großen Anzahl von Datensätzen, auf die die Operationen einer Prioritätsschlange angewendet werden sollen und die auf einem externen Speichermedium liegen, ist es im allgemeinen nicht angebracht, diese Datei in Form eines binären Suchbaums zu organisieren. Jeder Zugriff auf einen Datensatz, der dann in einem Knoten des binären Suchbaums liegt, erfordert einen Datentransfer zwischen Arbeitsspeicher und Peripherie. Der hierbei zu veranschlagende Zeitaufwand ist selbst im Idealfall des vollkommen höhenbalancierten binären Suchbaums in der Regel nicht akzeptabel. Aus den obigen Formeln wird ersichtlich, daß im Idealfall des vollkommen höhenbalancierten binären Suchbaums bei einer Datei mit beispielsweise 20 000 Datensätzen im Durchschnitt ca. 13,3 und bei 200 000 Datensätzen im Durchschnitt 16,6 (vergleichsweise sehr zeitaufwendige)

Plattenzugriffe erforderlich sind, um zu einem Datensatz zu gelangen. In der Praxis ist dieser Wert unakzeptabel. Es sind daher andere Realisierungsformen für Dateien angebracht.

Von den in der Literatur (vgl. z.B. [AHO]) beschriebenen Vorschlägen ist das Konzept des B^*-Baums zur Realisierung von Dateien von hoher praktischer Relevanz; es wird zur Speicherung von Datensätzen in ISAM-Dateien des SNI BS2000 und VSAM-Dateien im IBM-Betriebssystem MVS eingesetzt. Für Programmiersprachen wie PASCAL gibt es generierbare Units, die eine ISAM-Dateiverwaltung in Form eines B^*-Baums zur Verfügung stellen[36].

Die satzorientierte Zugriffsmethode ISAM verarbeitet Datensätze, die über Primärschlüsselwerte identifiziert werden. Physikalisch werden die Datensätze einer so bearbeiteten Datei (**ISAM-Datei**) geblockt, so daß i.a. ein Datenblock mehrere Datensätze enthält. Es gibt in einer ISAM-Datei zwei Typen von Blöcken (Abbildung 10.2.3-1):

- **Datenblöcke** enthalten die logischen Datensätze des Anwenders und Verwaltungsinformationen zur logischen Verkettung der Datenblöcke. Innerhalb eines Datenblocks sind die enthaltenen Sätze nach aufsteigenden Primärschlüsselwerten sortiert. Die Datenblöcke sind vorwärts bzw. rückwärts miteinander verkettet (Verweise über Blocknummern), und zwar zeigt die Vorwärtsverkettung auf den Datenblock, der in der Reihenfolge der Primärschlüsselwerte die nächsten Datensätze enthält; die Rückwärtsverkettung zeigt auf den Datenblock, der in dieser Reihenfolge die vorhergehenden Datensätze enthält. Bei einem Einstieg in die Datei in den Datenblock, der den Datensatz mit dem kleinsten Primärschlüsselwert enthält, ist daher eine sequentielle Satzverarbeitung möglich.

- **Indexblöcke** enthalten neben Verwaltungsinformationen mögliche Primärschlüsselwerte von Datensätzen und Verweise auf andere Indexblöcke bzw. Datenblöcke.

[36] Turbo Pascal Database Toolbox, Borland GmbH, 4. Auflage, 1990.

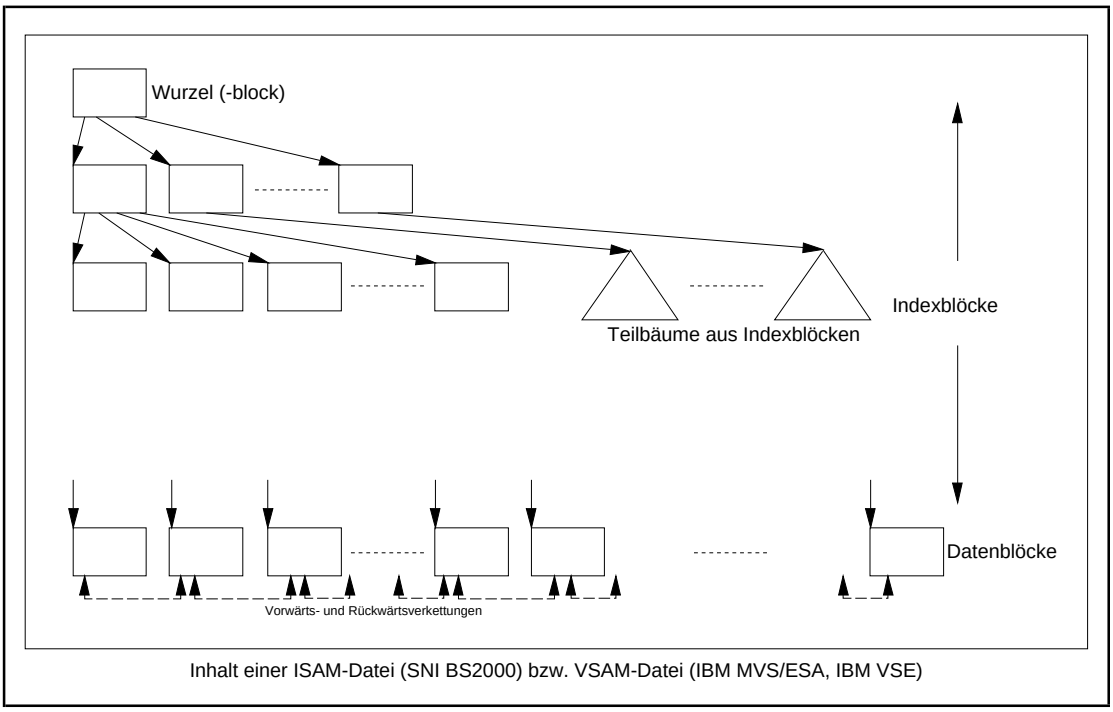


Abbildung 10.2.3-1: ISAM-Datei

Eine ISAM-Datei ist in Form eines B^*-Baums organisiert:

Ein B^*-Baum ist ein Baum, in der ein Knoten eventuell mehrere Nachfolger (auch mehr als 2) hat.

Im Fall einer ISAM-Datei stellen die Indexblöcke die inneren Knoten und die Datenblöcke die Blätter des B^*-Baums dar. Die Begriffe Knoten und Block werden im folgenden synonym verwendet.

Ein **B^*-Baum** wird durch zwei **Parameter *u* und *v*** und durch die folgenden Eigenschaften charakterisiert; die Parameter *u* und *v* beschreiben dabei einen Mindestfüllungsgrad der Indexblöcke bzw. der Datenblöcke:

1. Alle Blätter haben denselben Rang.

2. Die Wurzel ist ein Blatt oder hat mindestens 2 Nachfolger.

3. Jeder innere Knoten außer der Wurzel hat mindestens $u+1$ und höchstens $2u+1$ Nachfolger; falls die Wurzel kein Blatt ist, hat sie mindestens 2 und höchstens $2u+1$ Nachfolger.

4. Jeder innere Knoten mit $s+1$ Nachfolgern ist mit s Primärschlüsselwerten markiert; die Primärschlüsselwerte sind aufsteigend sortiert.

5. Jedes Blatt enthält, wenn es nicht die Wurzel ist, mindestens v und höchstens $2v$ Datensätze, die nach aufsteigenden Primärschlüsselwerten sortiert sind; ist die Wurzel ein Blatt, so enthält sie höchstens $2v$ nach aufsteigenden Primärschlüsselwerten sortierte Datensätze.

Ein **innerer Knoten (Indexblock)** läßt sich folgendermaßen skizzieren:

$[p_0 \quad k_1 \quad p_1 \quad k_2 \quad \ldots \quad k_s \quad p_s]$

Hierbei sind die Werte p_i Verweise (Pointer) auf die Nachfolger des Knotens, k_i Primärschlüsselwerte mit $k_1 \le k_2 \le \ldots \le k_s$. Jedes p_i zeigt auf einen untergeordneten Teilbaum T_{p_i}, der selbst wieder Primärschlüsselwerte enthält. Zu den obigen Bedingungen 1. - 5. kommt zusätzlich noch folgende Bedingung:

6. Alle in $T_{p_{i-1}}$ vorkommenden Primärschlüsselwerte sind kleiner oder gleich k_i, und k_i ist selbst kleiner als alle in T_{p_i} vorkommenden Primärschlüsselwerte, $1 \le i \le s$, $u \le s \le 2u$.

Ein **Blatt (Datenblock)** hat die Darstellung (hier sind zur vereinfachten Darstellung nur die Primärschlüsselwerte der Datensätze angegeben)

$[k_1 \quad k_2 \quad \ldots \quad k_t]$

mit $k_1 \le k_2 \le \ldots \le k_t$, $v \le t \le 2v$.

Zum schnelleren Navigieren durch den B*-Baum sind je nach Implementierung die Knoten im Baum vorwärts und rückwärts verkettet, d.h. ein Knoten enthält neben den (Vorwärts-) Verweisen auf seine Nachfolgerknoten einen zusätzlichen (Rückwärts-) Verweis auf seinen Vorgängerknoten. Außerdem sind (wieder implementierungsabhängig) alle Knoten desselben Rangs als doppelt-verkettete Liste miteinander verknüpft.

Um auf einen Datensatz in einem B*-Baum zugreifen zu können, muß ein Pfad von der Wurzel des Baums bis zu dem Blatt (Datenblock) durchlaufen werden, der den Datensatz enthält. In der Regel bewirkt der Besuch eines jeden Knotens einen physikalischen Datentransfer.

Das **Aufsuchen** des Datenblocks, der einen **Datensatz** mit vorgegebenem Primärschlüsselwert k enthält bzw. bei Nichtvorhandensein in der Datei enthalten würde, wenn der Datensatz in der Datei vorkäme, kann folgendermaßen ablaufen: Ist die Wurzel ein Blatt, also selbst ein Datenblock, so ist damit der Datenblock gefunden. Ist die Wurzel kein Blatt, sondern ein Indexblock, so werden die Werte k_j dieses Indexblocks mit k verglichen, bis derjenige Eintrag gefunden ist, für den $k_j < k \le k_{j+1}$ gilt; für $k \le k_1$ ist

$j = 0$, für $k > k_s$ wird nur mit k_s verglichen und $j = s$ gesetzt. Dann wird im Indexblock, auf den p_j verweist, nach der gleichen Methode weitergesucht, bis man ein Blatt, also einen Datenblock erreicht hat. Dieses ist der Datenblock, der den Datensatz mit dem Primärschlüsselwert *k* enthält bzw. enthalten würde.

Alle **Routinen zur Manipulation** (Einfügen, Entfernen, Aufsuchen eines Datensatzes, Durchlaufen eines B^*-Baums usw.) müssen sicherstellen, daß durch eine Manipulation im Baum die definierenden Eigenschaften nicht verletzt werden. Problematisch sind die Operationen zur Aufnahme weiterer Datensätze und zur Entfernung von Datensätzen aus der Datenstruktur, da diese Operationen sicherstellen müssen, daß die vollkommene Höhenbalance jeweils erhalten bleibt. Dabei tritt eventuell eine Situation ein, in der ein weiterer Datensatz in einen Datenblock einzufügen ist, der aber bereits vollständig belegt ist (**Überlauf**). Entsprechend kann eine Situation entstehen, in der ein Datensatz zu entfernen ist, so daß der betreffende Datenblock anschließend weniger als *v* Datensätze enthält (**Unterlauf**). Über- und Unterlaufsituationen gibt es auch im Zusammenhang mit Indexblöcken.

Die üblichen in der Literatur besprochenen Verfahren bewirken, daß der B^*-Baum in Richtung von den Blättern zur Wurzel wächst.

Im folgende Beispiel (Abbildung 10.2.3-2) werden Datensätze in einen anfangs leeren B^*-Baum eingefügt und aus ihm entfernt. Dargestellt sind nur die Primärschlüsselwerte (hier key-Werte genannt) und die Baumstruktur ohne die zusätzlichen implementierungsabhängi-
gen Verweise zur Erleichterung der Navigation durch den Baum. Die Parameter des B^*-Baums lauten $u = v = 2$. Das Prinzip der Über- und Unterlaufbehandlung wird im wesentlichen Ablauf erläutert (weitere Details werden z.B. in [HOF] beschrieben):

Um einen neuen Datensatz mit Primärschlüsselwert *k* einzufügen, wird zunächst der Datenblock aufgesucht (siehe oben), der in der Sortierreihenfolge den Datensatz aufnehmen müßte. Wenn dieser Datenblock noch nicht vollständig gefüllt ist, also noch nicht $2v$ Sätze enthält, wird der neue Datensatz an die in der Sortierreihenfolge richtigen Position eingefügt. Ist der Datenblock bereits vollständig belegt, so hat er die Form

$[k_1 \quad k_2 \quad \ldots \quad k_v \quad k_{v+1} \quad \ldots \quad k_{2v}]$.

Da der neue Datensatz in diesen Datenblock gehört, gilt $k_1 \le k \le k_{2v}$. Damit der neuer Datensatz eingefügt werden kann, wird der Datenblock in zwei Datenblöcke

$[k_1 \quad k_2 \quad \ldots \quad k_v]$ und $[k_{v+1} \quad \ldots \quad k_{2v}]$

aufgeteilt und der neue Datensatz bei $k \le k_v$ in den ersten bzw. bei $k > k_v$ in den zweiten dieser aufgeteilten Datenblöcke an der in der Sortierung richtigen Position eingefügt. Außerdem wird der *größte Primärschlüsselwert im ersten* der aufgeteilten Datenblöcke *zusammen mit einem Verweis auf den zweiten der aufgeteilten Datenblöcke* in den

übergeordneten Indexblock an die richtige Position einsortiert, die sich durch die Reihenfolge der Primärschlüsselwerte im Indexblock ergibt. In Abbildung 10.2-2 tritt diese Überlaufsituation z.B. beim Einfügen der Datensätze mit Primärschlüsselwerten 19 bzw. 12 ein.

Eine eventuelle Überlaufbehandlung in einem Indexblock wird analog behandelt, wobei sich diese u.U. rekursiv bis zur Aufnahme einer neuen Wurzel fortsetzt (in Abbildung 10.2.3-2 z.B. bei der Aufnahme der Datensätze mit Primärschlüssel 5 bzw. 12): Ein vollständig belegter Indexblock hat die Form

$[p_0 \quad k_1 \quad p_1 \quad k_2 \quad \ldots \quad k_{2u} \quad p_{2u}]$;

der einzusortierende Eintrag sei $[k, p]$. Man kann sich zunächst $[k, p]$ in den vollen Indexblock an die größenmäßig richtige Stelle einsortiert denken, d.h. bei $k \leq k_1$ entsteht dabei

$[p_0 \quad k \quad p \quad k_1 \quad p_1 \quad k_2 \quad \ldots \quad k_{2u} \quad p_{2u}]$,

bei $k_j < k \leq k_{j+1}$ für ein j mit $1 \leq j < 2u$ entsteht

$[p_0 \quad k_1 \quad p_1 \quad \ldots \quad k_j \quad p_j \quad k \quad p \quad k_{j+1} \quad p_{j+1} \quad \ldots \quad k_{2u} \quad p_{2u}]$,

und bei $k > k_{2u}$ entsteht

$[p_0 \quad k_1 \quad p_1 \quad k_2 \quad \ldots \quad k_{2u} \quad p_{2u} \quad k \quad p]$.

Dieser (gedachte) Block enthält $2u$+1 Primärschlüsselwerte und $2u$+2 Verweise und kann daher als

$[q_0 \quad l_1 \quad q_1 \quad l_2 \quad \ldots l_u \quad q_u \quad l_{u+1} \quad q_{u+1} \quad l_{u+2} \quad \ldots \quad l_{2u+1} \quad q_{2u+1}]$

geschrieben werden ($\{q_0, \ldots, q_{2u+1}\} = \{p_0, \ldots, p_{2u}, p\}$ und $\{l_1, \ldots, l_{2u+1}\} = \{k_1, \ldots, k_{2u}, k\}$). Er wird in zwei Indexblöcke

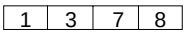
$[q_0 \quad l_1 \quad q_1 \quad l_2 \quad \ldots l_u \quad q_u]$ und

$[q_{u+1} \quad l_{u+2} \quad \ldots \quad l_{2u+1} \quad q_{2u+1}]$

aufgeteilt und der *Wert l_{u+1} zusammen mit einem Verweis auf den zweiten der aufgeteilten Indexblöcke* nach dem gleichen Prinzip in den Indexblock eingeordnet, der dem übergelaufenen Indexblock im B*-Baum übergeordnet war (evtl. muß dabei eine neue Wurzel angelegt werden).

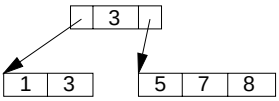
Eine Unterlaufbehandlung eines Datenblocks ist erforderlich, wenn er bei Entfernen eines Datensatzes weniger als v Datensätze enthalten würde. Er wird dann mit einem Nachbar-Datenblock zu einem Datenblock verschmolzen. Wenn dabei ein Überlauf entsteht, wird diese Situation wie oben behandelt. Aus dem übergeordneten Indexblock wird ein entsprechender Verweis entfernt, wobei auch hierbei ein Unterlauf eintreten kann, dem durch Zusammenlegen benachbarter Indexblöcke begegnet wird. Eine eventuelle Überlaufbehandlung beim Verschmelzen der Indexblöcke wird wie oben behandelt. In Abbildung 10.2.3-2 finden derartige komplexe Vorgänge bei der Entfernung des Datensatzes mit dem Primärschlüsselwert 2 statt.

Ausgangssituation ist ein leerer B*-Baum
Es werden nacheinander Datensätze mit den key-Werten
7, 1, 3, 8 aufgenommen:



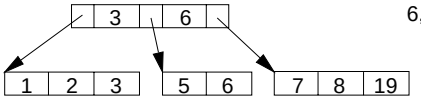
Aufnahme eines Datensatzes mit key-Wert

5



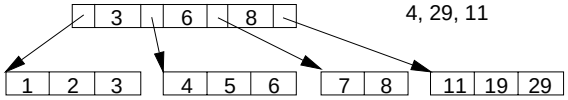
Aufnahme von Datensätzen mit key-Werten

6, 2, 19



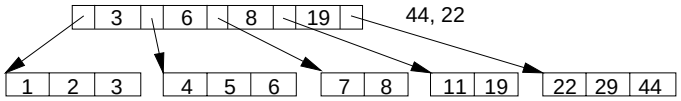
Aufnahme von Datensätzen mit key-Werten

4, 29, 11



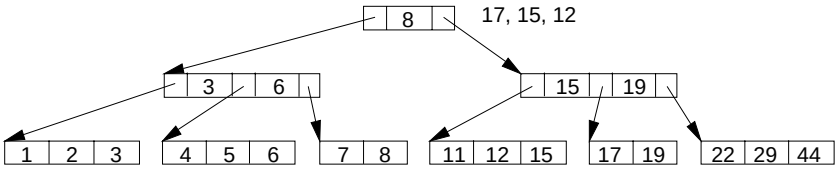
Aufnahme von Datensätzen mit key-Werten

44, 22



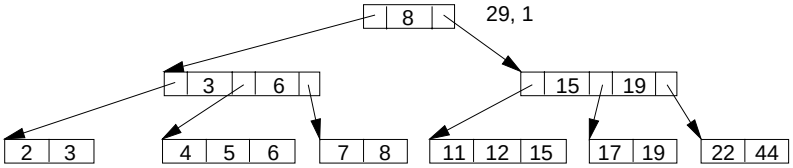
Aufnahme von Datensätzen mit key-Werten

17, 15, 12



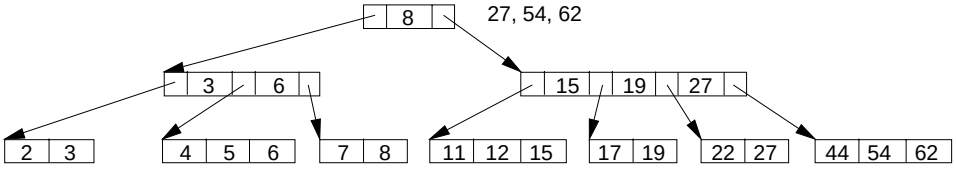
Entfernen der Datensätze mit key-Werten

29, 1



Aufnahme von Datensätzen mit key-Werten

27, 54, 62



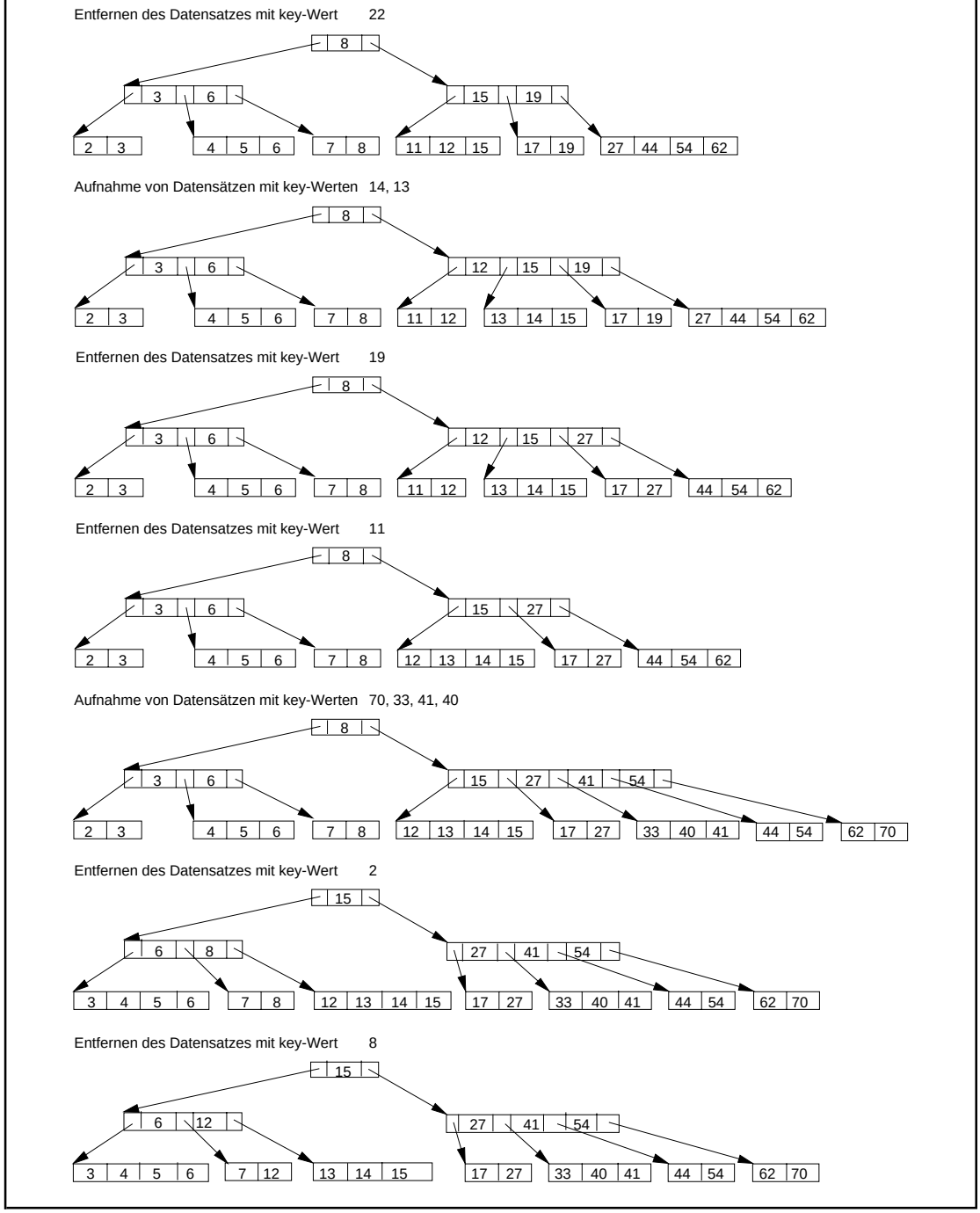


Abbildung 10.2.3-2: Beispiel eines B*-Baums

Wie oben beschrieben, ist die Suchdauer nach einem vorgegebenen Datensatz bzw. das Entfernen oder Einfügen eines Datensatzes in einem B*-Baum proportional zur Länge eines Pfades von der Wurzel zu einem Blatt. Alle diese Pfade sind aufgrund der vollständigen Höhenbalance des Baums gleichlang, und zwar gleich der Höhe des B*-Baums, die somit einen Maßstab für die Zugriffsgeschwindigkeit (Performance) dieser Datenstruktur liefert. Die Länge eines Pfades bzw. die Zugriffsgeschweindigkeit auf einen Datenblock hängt nicht vom Primärschlüsselwert ab, nach dem gesucht wird,

sondern von der Anzahl an Datensätzen, die sich zur Zeit im B*-Baum befinden. Abbildung 10.2.3-3 zeigt das **Mengengerüst eines B*-Baums**. In dieser Darstellung führen aus einem inneren Knoten nur zwei Kanten heraus, die jedoch alle aus diesem Knoten herauskommenden Kanten repräsentieren. Der Zahlenwert an der linken Kante gibt an, wieviele Nachfolger der Knoten mindestens hat; der Zahlenwert an der rechten Kante beschreibt die Maximalzahl an herausführenden Kanten aus diesem Knoten. Bei einem Datenblock beziehen sich die Zahlenangaben auf die Mindest- bzw. Maximalanzahl an Datensätzen des Datenblocks.

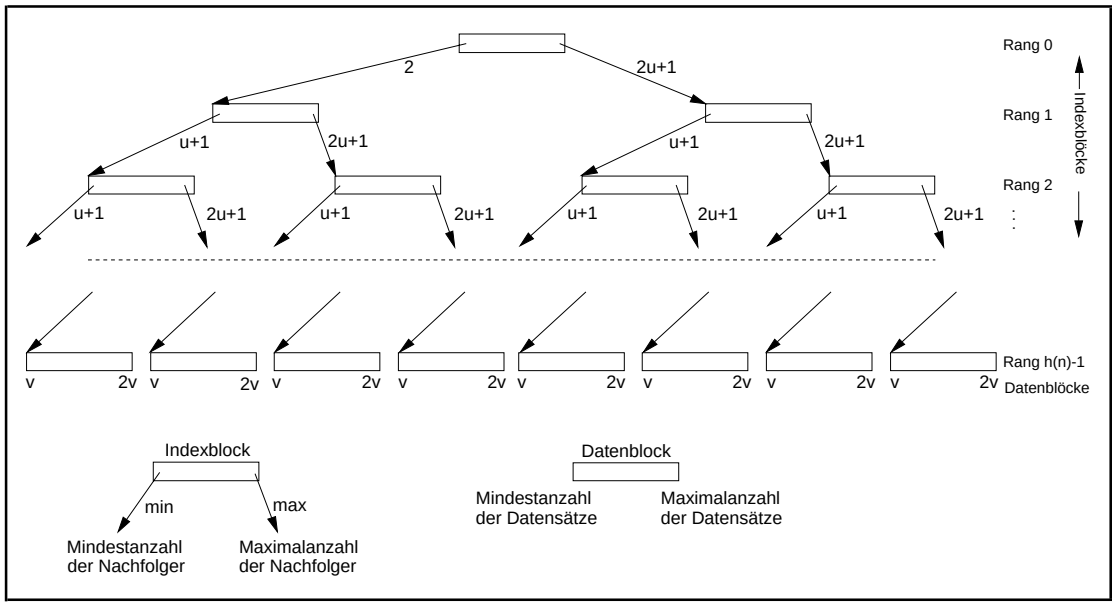


Abbildung 10.2.3-3: Mengengerüst eines B*-Baums

Der B*-Baum mit Parametern u und v enthalte $n \geq 1$ Datensätze. Mit $DB(n)$ werde die Anzahl der Datenblöcke bezeichnet, um diese Datensätze im B*-Baum zu speichern. Die Anzahl der dabei verwendeten Indexblöcke sei $IB(n)$. Die Höhe des B*-Baums sei $h(n)$. Dieser Wert beschreibt die Anzahl von Blocktransfers, die nötig sind, um an einen beliebigen Datensatz der Datei zu gelangen; der Aufwand zum Suchen *innerhalb* eines Datenblocks nach einem bezeichneten Datensatz bzw. eines Primärschlüsselwerts *innerhalb* eines Indexblocks bleibe hier unberücksichtigt.

Besteht der Baum nur aus der Wurzel, so gilt bei $1 \leq n \leq 2v$:
$h(n) = 1, \quad DB(n) = 1, \quad IB(n) = 0.$

Sobald der B*-Baum mindestens $2v+1$ Datensätze enthält, können diese nicht mehr in der Wurzel gespeichert werden, d.h. die Wurzel wird ein Indexblock, und der B*-Baum bekommt mindestens die Höhe 2. Die entsprechenden Werte bei Höhe 2 lauten:

$h(n) = 2, \quad 2v + 1 \leq n \leq 2v(2u + 1), \quad 2 \leq DB(n) \leq 2u + 1, \quad IB(n) = 1.$

Aus Abbildung 10.2.3-3 sieht man, daß bei $h(n) = h \geq 2$ gilt:

$h(n) = h \geq 2, \quad 2v(u+1)^{h-2} \leq n \leq 2v(2u+1)^{h-1},$
$2(u+1)^{h-2} \leq DB(n) \leq (2u+1)^{h-1}$

und bei $h(n) = h \geq 3$ außerdem
$1 + 2\sum_{i=0}^{h-3}(u+1)^i \leq IB(n) \leq \sum_{i=0}^{h-2}(2u+1)^i$, also
$1 + 2\frac{(u+1)^{h-2}-1}{u} \leq IB(n) \leq \frac{(2u+1)^{h-1}-1}{2u}.$

Enthält eine Datei, die nach dem hier beschriebenen Prinzip eines B*-Baums mit Parametern u und v gespeichert ist, $n \geq 2v+1$ Datensätze, so gilt für die Höhe $h(n)$ des Baums:

$$\frac{\log(n/(2v))}{\log(2u+1)} + 1 \leq h(n) \leq \frac{\log(n/(2v))}{\log(u+1)} + 2.$$

Im allgemeinen belegen Primärschlüsselwerte und die Verweise weniger Speicherplatz als komplette Datensätze. Daher kann man $u \geq v$ wählen.

Belegt beispielsweise ein Datensatz 200 Bytes, wobei 20 Bytes davon für den Primärschlüssel benötigt werden, so können bei einer Blockgröße von 2048 Bytes 10 Datensätze pro Datenblock abgelegt werden (der Platz für implementierungsabhängige Zusatzverweise ist bereits eingerechnet), d.h. $2v = 10$. Sieht man für einen Verweis innerhalb eines Indexblocks 4 Bytes vor, so ist er gefüllt, wenn er $2u = 84$ Primärschlüsselwerte und Verweise enthält; die restlichen Bytes seien für zusätzliche organisatorische Verweise reserviert. Bei $n = 15.000$ Datensätzen gilt dann für die Höhe $h(n)$ des B*-Baums und damit für die Anzahl physikalischer Blocktransfers, um einen Datensatz aufzusuchen, $2{,}64 \leq h(15.000) \leq 3{,}94$, also $h(n) = 3$. Bei $n = 150.000$ ist $3{,}164 \leq h(150.000) \leq 4{,}557$, also $h(n) = 4$. Bei einer weiteren Verzehnfachung der Anzahl an Datensätzen auf $n = 1.500.000$ wächst die Anzahl benötigter Blocktransfers, um einen Datensatz aufzusuchen, auf $h(n) = 5$, denn $4{,}556 \leq h(1.500.000) \leq 5{,}169$.

Die Anzahl wirklich benötigter Datenblöcke hängt vom Füllungsgrad eines Datenblockes und implizit von der Reihenfolge ab, in der Datensätze in die Datei aufgenommen bzw. aus ihr entfernt werden. Das gleiche gilt für die Anzahl benötigter Indexblöcke. Der "Preis", der letztlich für die schnelle Zugriffszeit auf Datensätze gezahlt werden muß, ist der Platzbedarf für die Indexblöcke, die neben den Datenblöcken in der Datei abgelegt werden müssen.

11 Typische Verfahren in der systemnahen Programmierung: Ein kleiner Betriebssystemkern

In diesem Kapitel werden die grundlegenden Abläufen in einem Betriebssystem gezeigt. Dabei werden die in Kapitel 10 beschriebenen Datenstrukturen verwendet. Da im praktischen Einsatz befindliche Betriebssystem wie UNIX (-Varianten), WINDOWS NT, Host-Betriebssysteme usw. aufgrund der an sie gestellten Anforderungen eine sehr große Komplexität aufweisen, wird die Darstellung der dort eingesetzten wesentlichen Mechanismen jedoch erschwert. Es müssen ja schwierige Funktionsbereiche wie Prozeß- und Speichermanagement, Dateiverwaltung, Gerätesteuerung, Ein/Ausgabemanagement, Netzwerksteuerung usw. teilweise auf unterschiedlichen Hardwareplattformen abgedeckt werden (Abbildung 11-1). Ein Blick auf die umfangreiche UNIX-Literatur und -Systemdokumentation bestätigt diesen Eindruck, auch wenn es inzwischen sehr handliche Implementationen, insbesondere im PC-Bereich, gibt. Aufgrund dieser Komplexität der Verfahren in konkreten Betriebssystemen wird im folgenden die Implementation eines kleinen und überschaubaren Betriebssystemkerns entwickelt, der es ermöglicht, unter MS-DOS einfache parallel ablaufende Prozesse zu realisieren, an denen wichtige Problemstellungen aus der Theorie nebenläufiger Prozesse demonstriert werden können (eine grundlegende Behandlung der hier angesprochenen Probleme findet man in der Literatur zu Betriebssystemen, z.B. in [TAN], [H/H] oder [M/O]). Der objektorientierte Ansatz und die Ausdrucksmöglichkeiten der Programmiersprache Pascal gewährleisten hierbei die Übersichtlichkeit der Verfahren. Die Auswahl für Methoden eines Betriebssystemkerns begründet sich aus dessen funktionaler Bedeutung innerhalb eines Betriebssystems.Im wesentlichen werden in diesem Kapitel die in Abbildung 11-1 fett umrahmten Funktionen behandelt.

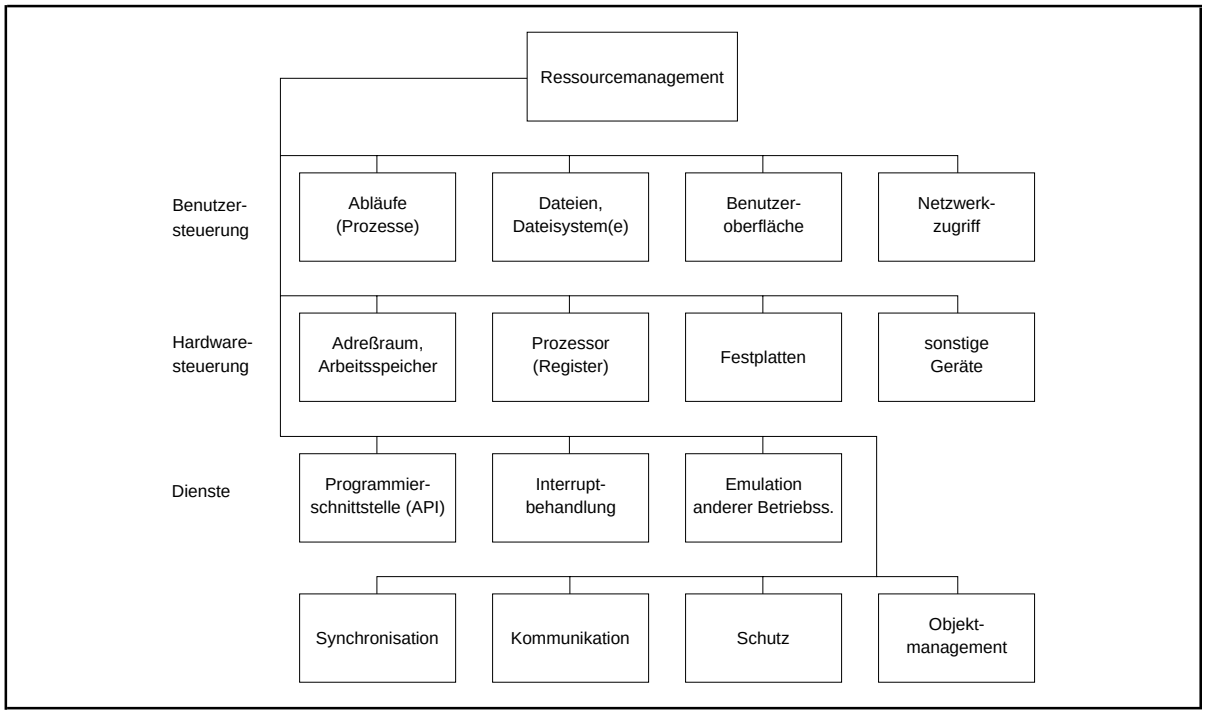


Abbildung 11-1: Aufgaben eines Betriebssystems

Der "klassische" Ansatz für das Layout eines Betriebssystems sieht eine Gliederung in logische Funktionsbereiche vor (Abbildung 11-2). Nach außen ist eine einheitliche Schnittstelle implementiert, über die die Anwendungsprozesse die Dienste des Betriebssystems in Anspruch nehmen können. Jeder Funktionsbereich wird durch eine Menge streng voneinander getrennter Module implementiert. Die Funktionsbereiche werden so definiert, daß die Schichten aufeinander aufbauen, d.h. aus jeder Schicht kann nur auf die Funktionen und Datenstrukturen einer darunterliegenden Schicht zugegriffen werden, so daß nur sehr wenige Codeteile unbegrenzten Zugriff auf die gesamte Hardware benötigen.

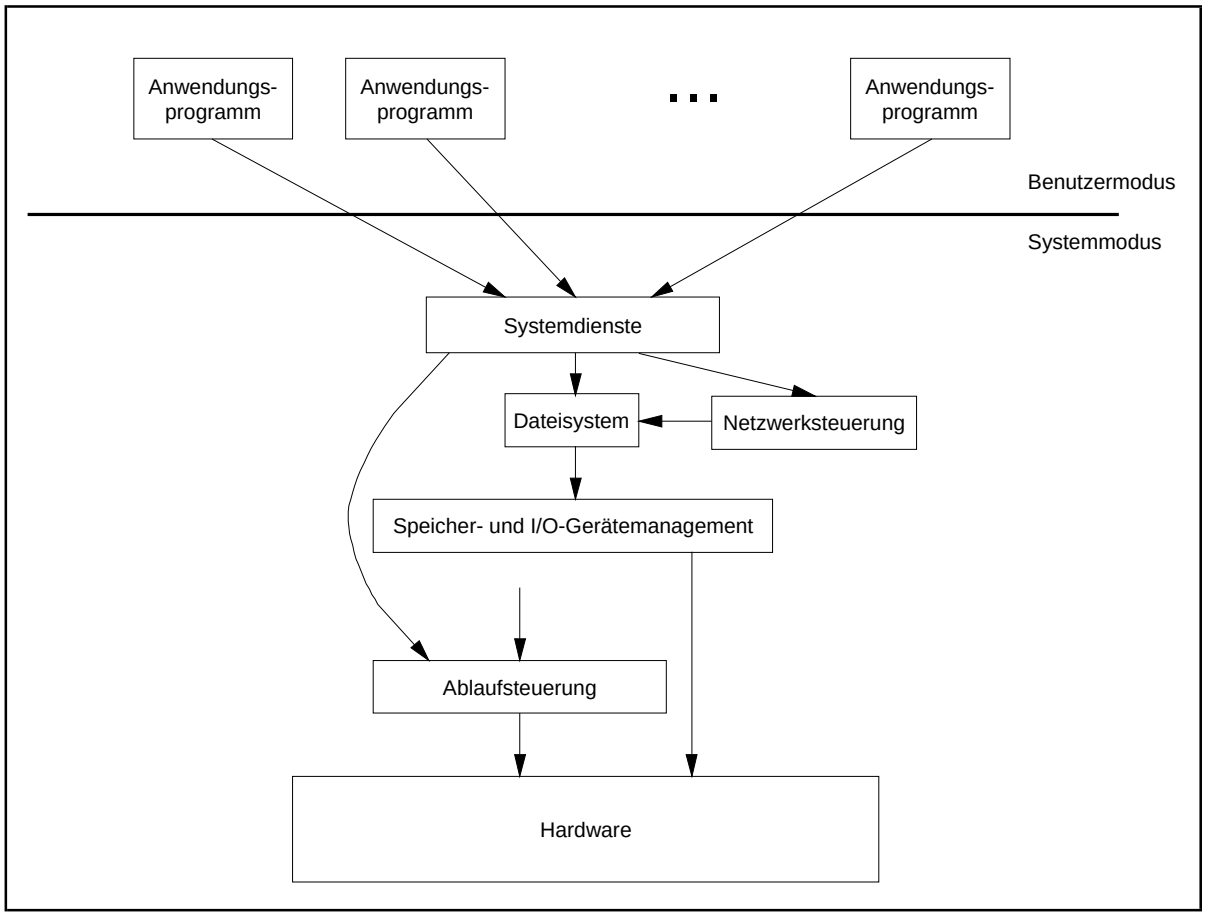


Abbildung 11-2: Klassische Betriebssystemstruktur

Neuere Betriebssystementwicklungen wie WINDOWS NT zeigen eher einen als Client-Server-Struktur bezeichneten Aufbau (Abbildung 11-3): Es wird versucht, den größten Teil der Betriebssystemdienste in Form von Anwendungsprozessen zu implementieren, die im Benutzermodus laufen, so daß nur ein minimaler Betriebssystemkern übrigbleibt, der den Systemmodus erfordert. Um einen Betriebssystemdienst anzufordern, sendet ein Benutzerprozeß (Klient, client), das auch ein Betriebssystemdienst sein kann, über den Betriebssystemkern eine **Nachricht** an den den Dienst implementierenden Anwendungsprozeß (server). Die Nachricht wird vom Betriebssystemkern an den Empfänger ausgeliefert; dieser führt den Dienst aus und sendet über den Betriebssystemkern eine Antwort an den Klienten zurück. Die so definierte Strukturierung des Betriebssystems bietet eine Reihe von Vorteilen:

- der den Systemmodus erfordernde Betriebssystemkern kann auf die wichtigsten Basisfunktionen wie Prozeßwechsel, Nachrichtenaustausch und Prozeßsynchronisation beschränkt werden

- fällt ein Betriebssystemdienst (Anwendungsprozeß) aus, wird das Funktionieren der übrigen Betriebssystemdienste nicht berührt

- in einer Mehrprozessorumgebung können einzelne Betriebssystemdienste leicht auf unterschiedliche Prozessoren verlagert werden, so daß eine Realisierung eines verteilten Betriebssystems ermöglicht wird

- nur der Betriebssystemkern ist teilweise noch maschinenabhängig, und die übrigen Betriebssystemdienste, die die Funktionalität des Betriebssystems ausmachen, sind leicht auf andere Hardwarearchitekturen portierbar, so daß eine derartige Struktur des Betriebssystem der Forderung nach einem offenen, d.h. auf vielen unterschiedlichen Hardwareplattformen einsetzbaren System, entgegenkommt.

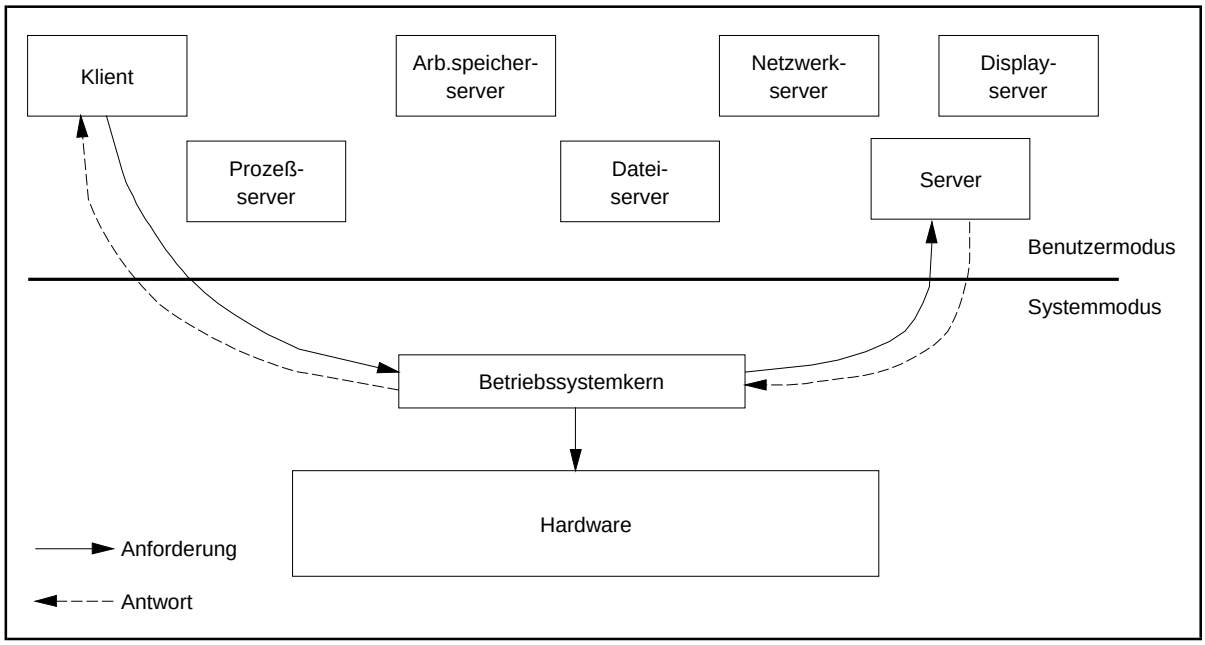


Abbildung 11-3: Client-Server-Betriebssystemstruktur

Typische Anwendungsfälle der systemnahen Programmierung kann man also an Abläufen im Betriebssystemkern kennenlernen. Dazu gehören Verfahren der Prozeßsteuerung und -synchronisation und der Interprozeßkommunikation. Im folgenden werden zur Konkretisierung daher eine Reihe von Pascal-Units vorgestellt, die ein einfaches Multitasking-Modell implementieren und unter MS-DOS ablaufen. Die zentrale Unit trägt die Bezeichnung `Multitsk`. Sie ist von der in [TIS] angegebenen Unit `MT` abgeleitet; geändert wurden im wesentlichen die interne Verwaltung von Prozessen (Anpassung an die in Kapitel 10 beschriebenen Datenstrukturen), die Benutzerschnittstelle (Vereinfachung) und die Mechanismen zur Realisierung der Prozeßsynchronisation (erweiterte zählende anstelle binärer

Semaphore, Ereignisvariablen) und der Interprozeßkommunikation. Entsprechend wurde auch der in [TIS] beschriebene (INTEL 80x86-) Assemblerteil auf die notwendigen Routinen reduziert.

Im folgenden werden die Begriffe Prozeß und Task synonym verwendet.

Gegenüber den Prozeßmodellen in "realen" Betriebssystemen werden einige ***wichtige Einschränkungen*** gemacht (im wesentlichen sollen ja nur die Prinzipien der Prozeßverwaltung, der Prozeßsynchronisation und Interprozeßkommunikation behandelt werden, und der REAL-Mode des INTEL-Prozessors ist mit seiner Beschränkung auf 1 MB adressierbaren Arbeitsspeicher für die Realisierung eines Multitaskings nicht sonderlich gut geeignet):

- Der Code des Betriebssystemkerns, d.h. die im folgenden vorgestellten Units, ist nur in soweit vor Zugriffen eines Anwenderprozesses geschützt, wie es das Sprachkonzept der Pascal-Units vorsieht. Alle Code- und Datenteile, auf die ein Anwenderprozeß keinen Zugriff zu haben braucht, sind in den `IMPLEMENTATION`-Teil einer Unit verlagert bzw. sind in den Objekttypdeklarationen als `PRIVATE` gekennzeichnet. Es werden also lediglich die syntaktischen Schutzmechansimen der zugrundeliegenden Programmiersprache verwendet, die sich an dieser Stelle einerseits als sehr mächtig und andererseits als leicht zu handhaben darstellen. Alle Funktionen des Betriebssystemkerns werden über eine definierte Schnittstelle angestoßen, so da ein Anwenderprozeß internerne Details nicht zu kennen braucht.

- Alle Prozesse laufen in derselben Systemumgebung. Konzeptionell bedeutet das, daß im Sinne der Terminologie des Kapitels 3.2 ***kein Multitasking-, sondern ein Multithreading-Modell*** implementiert wird. Alle Threads laufen innerhalb des einen, nämlich des durch MS-DOS dargestellten Prozesses. Sie teilen sich den Adreßraum, den das Speichermodell von MS-DOS liefert. Insbesondere reagieren alle Prozesse (hier: Threads) auf dieselben Interrupts. Interruptvektoren und -behandlungsroutinen werden bei einem Threadwechsel nicht ausgetauscht, wie dies üblicherweise bei einem Prozeßwechsel in einem "realen" Multitasking-Betriebssystemen stattfindet. Dadurch wird die interne Verwaltung vereinfacht und der benötigte Speicherplatz erheblich reduziert. ***Innerhalb dieses Kapitels soll trotzdem ab sofort von (parallel-laufenden) Prozessen die Rede sein, auch wenn es sich konzeptionell um Threads eines Prozesses handelt***, da die vorgestellten Mechanismen in ähnlicher Weise auch in Multitasking-Umgebungen vorkommen.

- Jeder Prozeß (eigentlich: Thread) kann mit seinem Anwender über Bildschirmfenster kommunizieren, d.h. jeder Prozeß kann einen Teil des Bildschirms für seine Ausgaben verwenden. Eingaben liest er über die Tastatur, die ihm zugeordnet ist, sobald er aktiv ist. Ein Prozeß kann gleichzeitig mehrere Bildschirmfenster öffnen, die überlappend dargestellt werden, jedoch ist pro Prozeß immer nur ein Fenster, das **Vordergrundfenster**, aktiv. Mit entsprechenden Funktionen wird ein Wechsel zwischen den Fenstern eines Prozesses ausgelöst, so daß eventuell ein anderes als das gerade im Vordergrund befindliche Fenster zum Vordergrundfenster und vor die anderen Fenster des Prozesses gesetzt wird. Zur Vereinfachung sollten ***Bildschirmfenster verschiedener Prozesse nicht überlappend*** definiert werden, da alle Bildschirmfenster, einschließlich der Vordergrundfenster, in denselben internen Bildschirmspeicher von MS-DOS umgesetzt werden und bei einem Prozeßwechsel

der Ausschnitt des Bildschirmspeichers, auf das das jeweilige Bildschirmfenster gerade abgebildet ist, nicht in einem prozeßeigenen Speicherbereich gesichert wird. Bei sich überlappenden Fenstern könnten verschiedene Prozesse auf dieselbe Bildschirmposition ausgeben, so daß sich Fensterinhalte transparent überlagern. Die zur Fensterbehandlung erforderliche Unit `MultiWin`, deren Schnittstelle unten beschrieben wird, basiert im wesentlichen auf den Units `WinMt` und `Win2Mt` aus [TIS] (die Unit `MultiWin` reduziert deren sehr umfangreiche Schnittstelle auf wenige hier benötigte Prozeduren und Funktionen).

- Zur Laufzeit findet keine Abbildung von prozeßeigenen virtuellen Adreßräumen auf den Arbeitsspeicher statt, sondern alle Prozeduren, die in den Prozessen ablaufen, werden statisch gebunden und geladen. Der komplette Funktionsbereich der virtuellen Speicherverwaltung bleibt also im Modell ausgeklammert. Konzeptionell bedeutet dies, daß der Prozeßadreßraum statisch in den Arbeitsspeicher abgebildet wird und eine feste Zuordnung von Arbeitsspeicherbereichen an Prozesse stattfindet, die während der Prozeßlebensdauer nicht geändert wird.

- Jedem Prozeß (eigentlich: Thread) wird vor Prozeßstart ein prozeßeigener Stack zugeordnet. Dieser kann während der Prozeßlebensdauer nicht in seiner Größe verändert werden. Alle Stacks liegen resident im Arbeitsspeicher. Aufgrund dieser privaten Stacks können unterschiedliche Prozesse parallel durch dieselben Prozeduren laufen. Kapitel 7.6 beschreibt, was programmiertechnisch dabei zu beachten ist.

- Jeder Prozeß (eigentlich: Thread) besitzt einen eigenen virtuellen Registersatz, der bei Prozeßwechsel auf seinem prozeßeigenen Stack gesichert wird.

Eine **Anwendung, die die Unit `Multitsk` einbindet**, enthält Daten, "normale" Prozeduren und parallel ablaufende Prozesse (Tasks). Die Daten und Prozeduren können von allen Prozessen verwendet werden, stellen also aus Sicht der Prozesse **globale, gemeinsam verwendbare Ressourcen** dar. Ein Zugriff auf diese Objekte bedarf also eventuell einer Prozeßsynchronisation.

Prinzipiell kann eine Anwendung zunächst sequentielle Prozeduraufrufe enthalten, anschließend parallele Prozesse einrichten und unter Kontrolle des Multitaskings in der Unit `Multitsk` ablaufen lassen und dann wieder zu einem sequentiellen Prozedurablauf zurückkehren. Dieser Zyklus ist mehrmals wiederholbar. Das Einrichten von Prozessen kann auch dann noch erfolgen, wenn bereits parallele Prozesse aktiv sind. Ein neuer Prozeß wird dann sofort in das Multitasking aufgenommen. Nach Beendigung aller parallelen Prozesse beendet sich die Kontrolle des Multitaskings, kann dann aber, nachdem neue Prozesse eingerichtet wurden, erneut angestoßen werden. In den anfänglichen bzw. abschließenden sequentiellen Teilen der Anwendung können z.B. Synchronisationshilfsmitteln eingerichtet und initialisiert bzw. wieder aus dem System entfernt werden.

Abbildung 11-4 zeigt den Aufbau des Quellcodes einer Anwendung, die zwei parallele Prozesse enthält, die jeweils mit den Prozeduren `prozess1` bzw. `prozess2` starten sollen.

```
PROGRAM Anwendung;

USES Multitsk { evtl. werden weitere Units benötigt,
                siehe Kapitel 11.2 };

{ --- hier stehen Deklarationen globaler Objekte, die auch in den
  --- parallelen Prozeduren verfügbar sind                          }

{$F+}
PROCEDURE prozess1;

  { --- Deklarationen lokaler Objekte für prozess1 }

  BEGIN { prozess1 }

    { --- Programmcode für prozess1 }

  END   { prozess1 };
{$F-}

{$F+}
PROCEDURE prozess2;

  { --- Deklarationen lokaler Objekte für prozess2 }

  BEGIN { prozess2 }

    { --- Programmcode für prozess2 }

  END   { prozess2 };
{$F-}

BEGIN { anwendung }

  { --- sequentielle Anweisungen, die die globalen
        Objekte von Anwendung verwenden, z.B. Einrichten
        von Synchronisationshilfsmitteln                 }

  { Prozesse einrichten: }
  MTCreateTask (prozess1, ...);
  MTCreateTask (prozess2, ...)

  { Multitasking starten; die eingerichteten Prozesse
    laufen parallel unter Kontrolle der Unit Multitsk ab: }
  MTStart (...);


  { --- sequentielle Anweisungen, die die globalen
        Objekte von Anwendung verwenden, z.B. Entfernen der
        zuvor eingerichteten Synchronisationshilfsmittel    }

END   { Anwendung }.
```

Abbildung 11-4: Prinzipieller Aufbau einer Anwendung mit der Unit `Multitsk`

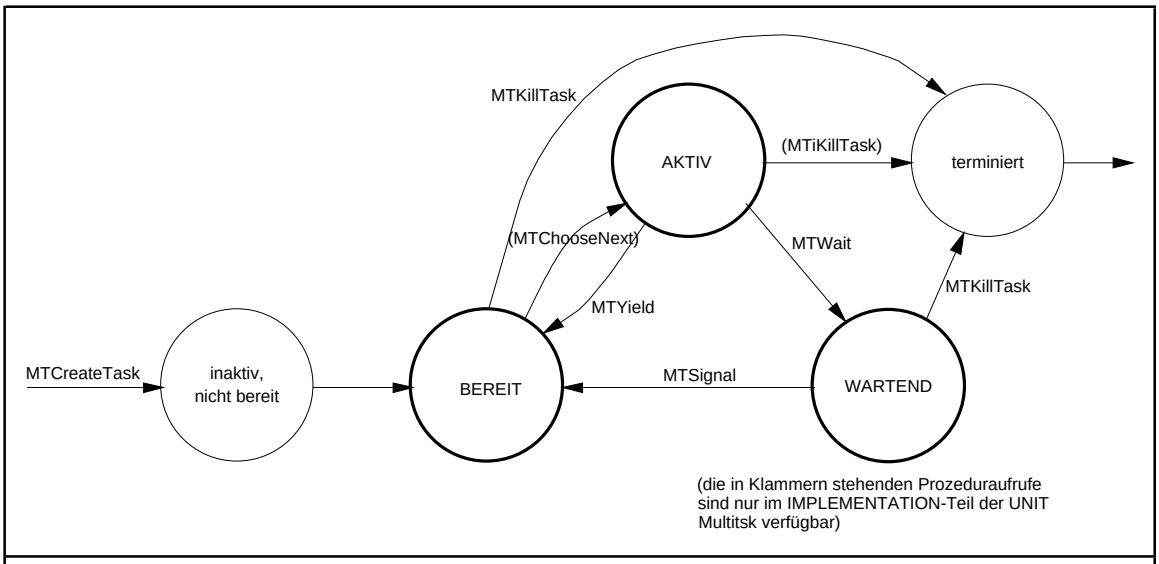
Die Einrichtung eines Prozesses bewirkt der Aufruf der Prozedur `MTCreateTask`, die im `INTERFACE`-Teil der Unit `Multitsk` deklariert ist (in Abbildung 11-4 sind nicht alle Parameter von `MTCreateTask` angegeben); dabei wird der Prozeß noch nicht gestartet, sondern zunächst dem Multitasking "bekannt" gemacht. Das Multitasking läuft durch Aufruf der Prozedur `MTStart` (ebenfalls aus der Unit `Multitsk`) an; von diesem Zeitpunkt an laufen beide Prozesse parallel durch ihre jeweiligen Prozeduren (beginnend bei `prozess1` bzw.

prozess2). Nach Beendigung beider Prozesse und der von ihnen eventuell zusätzlich ins Leben gerufenen weiteren Prozessen beendet sich `MTStart`. Das Ende des Prozedurlaufs von `MTStart` stellt gleichzeitig das Ende der Multitasking-Kontrolle dar.

Die Beschreibung in den folgenden Unterkapiteln gibt die wesentlichen Abläufe wieder, wobei gelegentlich einige Details (auch wenn für die Implementierung auf diese nicht verzichtet werden kann) zur besseren Übersichtlichkeit weggelassen wurden.

11.1 Ein vereinfachtes Prozeßmodell

Die Unit `Multitsk` geht von einem Prozeßmodell aus, in dem sich jeder Prozeß zu jedem Zeitpunkt in einem von fünf möglichen Prozeßzuständen befindet. Abbildung 11.1-1 erläutert die **Prozeßzustände** und zeigt die durch Prozeduraufrufe der Unit `Multitsk` ausgelösten möglichen Übergänge zwischen den Prozeßzuständen.



Die Prozeßzustände sind:

- **inaktiv, nicht bereit**: der Prozeß wird gerade in die Taskverwaltung aufgenommen und ist noch nicht aktiviert worden
- **BEREIT:** alle vom Prozeß benötigten Betriebsmittel bis auf die CPU stehen ihm zur Verfügung; ein anderer Prozeß ist gerade im Zustand AKTIV
- **AKTIV (laufend):** der Prozeß belegt gerade die CPU
- **WARTEND (nicht bereit):** der Prozeß wartet auf das Eintreten eines Ereignisses, z.B. auf eine Nachricht von einem anderen Prozeß oder auf die Zugriffsberechtigung auf ein Betriebsmittel; das Eintreten des Ereignisses wird von einem anderen Prozeß signalisiert
- **terminiert:** der Prozeß verläßt das System und wird aus der Prozeßverwaltung entfernt; vorher müssen eventuelle Querverbindungen zu anderen Prozessen oder Betriebsmitteln gelöst werden

Abbildung 11.1-1: Vereinfachtes Prozeßmodell

Die Zustände BEREIT und WARTEND unterscheiden sich im wesentlichen dadurch, daß ein Prozeß im Zustand BEREIT bei einem möglichen Prozeßwechsel überprüft wird, ob ihm jetzt die CPU zugeordnet werden kann (die Unit `Multitsk` setzt eine definierte **Schedulingstrategie** ein), während ein Prozeß im Zustand WARTEND auf keinen Fall als möglicher Kandidat für die nächste Zuweisung der CPU angesehen wird. Erst wenn das Eintreten des Ereignisses, auf das der Prozeß im Zustand WARTEND wartet, von einem anderen Prozeß signalisiert wird, wechselt er in den Zustand BEREIT.

Ein Prozeß im Zustand BEREIT kann für ein festgelegtes Zeitquantum Δ, eine **Zeitscheibe**, die CPU bekommen und ist dann im Zustand AKTIV. Falls er während dieser Zeit aufgrund seines Programmlaufs in einen Wartezustand (Zustand WARTEND) gerät, weil ihm ein zusätzliches Betriebsmittel, z.B. ein ausstehendes Ereignis, fehlt, wird sofort ein anderer Prozeß aus dem Zustand BEREIT aktiviert. Die Multitasking-Kontrolle stellt sicher, daß es immer mindestens einen Prozeß im Zustand BEREIT gibt, indem sie einen sogenannten **Idleprozeß** definiert, der immer dann aktiviert wird, wenn keine anderen Prozesse mehr im Zustand BEREIT sind, und der lediglich durch die Prozedur

```
{$F+}
PROCEDURE IdleTask( Zeiger : Pointer );

BEGIN { IdleTask }
  WHILE TRUE DO ;
END   { IdleTask };
{$F-}
```

läuft, die sich im `IMPLEMENTATION`-Teil der Unit `Multitsk` befindet. Gibt es mehrere Prozesse im Zustand BEREIT, wird ein Prozeß mit höchster **(interner) Priorität** ausgewählt. Diese bestimmt sich aus der **externen Priorität**, die jeder Prozeß bei Einrichtung erhält und die im Laufe der Prozeßlebensdauer veränderbar ist (Aufruf `MTChangePrio` im `INTERFACE`-Teil der Unit `Multitsk`). Der Idleprozeß hat minimale externe Priorität. Ein weiterer Einflußfaktor auf die interne Priorität ist der Zeitpunkt, an dem ein Prozeß zuletzt die CPU bekommen hat: Die interne Priorität eines Prozesses ist umso höher, je höher seine externe Priorität im Vergleich zu den anderen Prozessen im System ist und bei gleicher externer Priorität je weiter der Zeitpunkt zurückliegt, an dem er zuletzt im Zustand AKTIV war. Die Unit `Multitsk` verwendet als **logische Uhr** einen mit 0 initialisierten Zähler, der bei jedem Prozeßwechsel inkrementiert wird; der aktuelle Zählerstand wird beim Prozeß vermerkt, der vom Zustand BEREIT in den Zustand AKTIV wechselt. Diese logische Uhr befindet sich in der Form

```
CONST Aufrufe : Longint = 0;     { bisherige Aufrufe dieser Prozedur }
```

in der Prozedur `MTChooseNext` der Unit `Multitsk`. Nach Ablauf der Zeitscheibe Δ wird dem aktiven Prozeß die CPU entzogen, und er wechselt wieder in den Zustand BEREIT. Standardmäßig ist die Größe der Zeitscheibe $\Delta = 1/18.2$ sec. Sie kann bei Start des Multitaskings um einen Faktor verkleinert oder vergrößert werden (siehe Kapitel 11.3.3). Geht man davon aus, daß ein Anteil von *k* Prozent aller Prozeßwechsel nur aufgrund des Ablaufs der Zeitscheibe erfolgt und ein Prozeß, der von sich aus einen Prozeßwechsel initiiert, diesen

im Mittel nach Ablauf der halben Zeitscheibe auslöst, so ist die mittlere Länge einer Zeitscheibe

$$\bar{\Delta} = \alpha\Delta + \frac{1-\alpha}{2}\Delta \text{ mit } \alpha = k/100,$$

d.h. $\bar{\Delta} = \frac{\alpha+1}{2}\Delta$.

Die logische Uhr mit Datentyp `Longint` wird also nach Ablauf von $2.147.483.647 \cdot \bar{\Delta}$ sec. auf einen Zahlenbereichsüberlauf stoßen, so daß sich das Multitasking nach Ablauf dieser Zeitdauer mit einem Laufzeitfehler beendet[37]. Nimmt man als Zahlenbeispiel $\alpha = 0{,}9$ und $\Delta = 1/18.2$ sec., so kann eine Anwendung mit der Unit `Multitsk` maximal 112.093.926,63 sec., d.h. etwa 1297 Tage bzw. ca. 3,5 Jahre, laufen, bevor die logische Uhr überläuft.

Insgesamt wird eine Strategie verfolgt, die als eine **durch externe Prioritäten gesteuerte unterbrechende Schedulingstrategie mit Zeitscheibe (round robin preemptive scheduling)** bezeichnet wird; Prioritätskonflikte werden dabei nach der **LRU-Regel (least recently used)** gelöst.

Jeder Prozeß besitzt nach Aufnahme in die Multitasking-Kontrolle ein Datenobjekt vom Objekttyp `TD` (*T*ask-*D*eskriptor), das ihn beschreibt. Dieses Datenobjekt wird **TCB (Task Control Block) des Prozesses** genannt. Ein Anwenderprozeß hat keinen direkten Zugriff auf seinen TCB, trotzdem erleichtert die Kenntnis der TCB-Komponenten das Verständnis des Multitasking-Mechanismus (diese Details werden in Kapitel 11.3.2 beschrieben). Die Objekttypdeklaration des TCBs befindet sich in der Unit `TCB`, die von einer Anwendung aber *nicht* eingebunden wird. ***Ein Anwenderprozeß darf auf keinen Fall selbst die Werte seines eigenen TCBs auswerten oder verändern; dies ist ausschließlich den Prozeduren der Multitasking-Kontrolle vorbehalten.***

11.2 Anwenderschnittstellen des Multitaskings

Eine typische Anwendung wird einige parallel-laufende Prozesse beinhalten, die von der Unit `Multitsk` gesteuert werden und mit ihrer Umwelt über Fenster kommunizieren, die auf dem Bildschirm abgebildet sind. Zur Verwaltung von Bildschirmfenster kann die Unit `MultiWin` eingebunden werden (diese Unit benutzt die in [TIS] beschriebenen Unit `WinMt` mit der `INCLUDE`-Datei `Win2Mt.PAS`, wobei in der dortigen `USES`-Anweisung jetzt auf die Unit `Multitsk` Bezug genommen wird).

Weitere Units können in der Anwendung eingebunden werden, natürlich auch Units, die Anwenderfunktionalität beinhalten, also nicht zu den Units der hier beschriebenen Multitasking-Kontrolle gehören. Damit eine derartige Unit Gelegenheit hat, für jeden neu eingerichteten Prozeß Unit-spezifische Initialisierungsaktivitäten durchzuführen, kann man die Unit als **beim Multitasking kooperierende Unit** anmelden. Die **Registrierung (Anmeldung)** muß erfolgen, bevor der erste Prozeß mit `MTCreateTask` eingerichtet wird.

[37] Die Unit Multitsk fängt Laufzeitfehler ab, so daß eine Beendigung "geregelt" erfolgt.

Bis zu 5 Units können sich als kooperierende Units registrieren lassen. Eine kooperierende Unit kann außerdem festlegen, daß bei jedem Prozeßwechsel und bei Prozeßende Unit-spezifische Aktionen durchzuführen sind. Welche Aktionen das sind, wird bei der Registrierung als kooperierende Unit mitgeteilt, indem entsprechende Prozeduren als Parameter übergeben werden.

Einige Typdeklarationen sind erforderlich, wenn man mit kooperierenden Units arbeitet. Diese sind in der Unit `koopUNIT` zusammengefaßt, die daher bei Bedarf in die Anwendung eingebunden werden muß:

```
UNIT koopUNIT;

INTERFACE

CONST max_units  = 5;           { maximale Anzahl mitarbeitender Units }
      AnzLUnits  : INTEGER = 0; { Anzahl mitarbeitender Units          }

TYPE UpCreate = FUNCTION : Pointer;
     UpChange = PROCEDURE (Zeiger : Pointer);
     UpDelete = PROCEDURE (Zeiger    : Pointer;
                           RestTasks : INTEGER);
                                { Prozeduren mitarbeitender Units bei
                                  Einrichten einer Task, Taskwechsel
                                  bzw. Entfernen einer Task           }
     UNITproz = RECORD
                  CreateProz : UpCreate;
                  ChangeProz : UpChange;
                  DeleteProz : UpDelete;
                END;

VAR LUnits : ARRAY [1..max_units] OF UNITproz;
                                { Prozeduren registrierter
                                  mitarbeitender (linked) Units       }

IMPLEMENTATION
END.
```

Die Registrierung einer Unit als kooperierende Unit erfolgt über den Aufruf der `FUNCTION MTRegisterUNIT` (aus der Unit `Multitsk`). Als Parameter werden drei Prozeduren bzw. Funktionen vom Typ `UpCreate`, `UpChange` und `UpDelete` mitgegeben, die für die zu registrierende Unit im `ARRAY LUnits` eingetragen werden. Die Registrierung als kooperierende Unit kann z.B. im Initialisierungsteil dieser Unit erfolgen. Beispielsweise läßt sich die Unit `WinMt`, die zur Verwaltung von Bildschirmfenstern durch die Unit `MultiWin` angebunden wird, in ihrem Initialisierungsteil durch den Aufruf

```
MtRegisterUNIT (WinTaskCreate, WinTaskChange, WinTaskDelete);
```

registrieren, wobei die Parameter in der Unit `WinMt` deklariert sind. So liefert die Funktion

```
FUNCTION WinTaskCreate : Pointer;
```

die Adresse eines Datenblocks, in dem der Bildschirmfensternbereich eines Prozesses verwaltet wird. Der Code der beiden anderen Prozeduren bestimmt die prozeßspezifischen Aktivitäten bei Prozeßwechsel (Wechsel auf einen anderen Fensterbereich) bzw. Prozeßende (Freigabe des Fensterbereichs des Prozesses).

Bei der Einrichtung eines TCBs für einen Prozeß wird für den Fall, daß sich kooperierende Units haben registrieren lassen, die `FUNCTION` vom Typ `UpCreate` der jeweiligen Unit aufgerufen und der Rückgabewert in einer Komponente des TCBs abgelegt; für jede kooperierende Unit ist dort ein Eintrag vorgesehen. Entsprechende Aufrufe erfolgen bei Prozeßwechsel und Prozeßende: Jetzt werden für die registrierten Units die jeweiligen Prozeduren vom Typ `UpChange` bzw. `UpDelete` ausgeführt. Auf diese Weise können kooperierende Units asynchron Informationen an Prozesse bei Prozeßstart, -wechsel und -beendigung übergeben.

Zusammenfassend wird eine Anwendung, die mit Multitasking, Fensterverwaltung (eine kooperierende Unit) und eventuell weiteren kooperierenden Units arbeitet, eine `USES`-Anweisung der Form

```
USES Multitsk, MultiWin;
```

aufführen. Die Units `TCB` und `koopUNIT` werden nicht angegeben, da sie im `INTERFACE`-Teil der Unit `Multitsk` vorkommen.

Zu beachten ist, daß ein Anwenderprozeß eigentlich keine internen Details der Multitasking-Kontrolle kennen sollte. Aus programmiertechnischen Gründen ist jedoch die Identifizierung des gerade aktiven Prozesses in Form der Adresse seines TCBs in der globalen Variablen `AktTask` aus dem `INTERFACE`-Teil der Unit `Multitsk` verfügbar. Allerdings kann ein Anwenderprozeß auf die einzelnen Komponenten eines TCBs und dessen Methoden nicht zugreifen, da die TCB-Objekttypdeklaration in einer eigenen Unit abgelegt ist, die in eine Anwendung mit Multitasking nicht eingebunden wird.

Gelegentlich sprechen Anwendungsprozesses andere Prozesse an, um ihnen beispielsweise eine Nachricht zu übermitteln oder sie aus dem System zu entfernen. In diesem Fall ist es notwendig, eine **Prozeßidentifizierung** des angesprochenen Prozesses zu kennen. Obwohl die Unit `Multitsk` beim Einrichten eines Prozesses automatisch eine Prozeßidentifikation erzeugt und im TCB ablegt, wird diese nicht verwendet, sondern ***ein Prozeß identifiziert sich durch seine TCB-Adresse***, d.h. um einen Prozeß explizit anzusprechen, benötigt man dessen TCB-Adresse (Zeiger auf seinen TCB).

Die folgende Tabelle faßt die **Schnittstelle der Unit `Multitsk` (Task-Verwaltung)** zusammen.

Variable (im INTERFACE-Teil der Unit Multitsk)	Bedeutung
`AktTask: TaskPtr;`	Verweis auf den TCB der gerade aktiven Task. ***Der Inhalt dieser Variablen darf von einer Anwendung nicht verändert werden.***

Prozedur/Funktion (im INTERFACE-Teil der Unit Multitsk)	Bedeutung
`PROCEDURE MTStart (Tempo : INTEGER);`	Das Multitasking wird gestartet, falls bereits Tasks mit `MTCreate` eingerichtet wurden

Bedeutung des Parameters:

`Tempo` Zeitfaktor für die Zeitscheibengröße:
= 1: 18.2 Prozeßwechsel pro Sekunde
> 1: Verkleinerung der Zeitscheibe um den angegebenen Faktor (häufigere Prozeßwechsel)
≤ -1: Vergrößerung der Zeitscheibe um den Absolutbetrag des angegebenen Faktors (weniger häufige Prozeßwechsel)

`FUNCTION MTCreateTask (Task : TS; Prio : BYTE; SLen : INTEGER; Zeiger : Pointer ) : TaskPtr;`	Ein neuer Prozeß wird in das Multitasking aufgenommen. Der Rückgabewert ist die Adresse des TCBs der neu eingerichteten Task bzw. `NIL`, falls die Task nicht angelegt werden konnte

Bedeutung der Parameter:

`Task` FAR-Prozedur, mit der die Ausführung der Task startet

`Prio` externe Taskpriorität

`SLen` Größe des Task-Stacks (in Bytes)

`Zeiger` Zeiger auf einen Parametersatz (Einzelparameter oder `RECORD`-Typ), der dem Prozeß bei Prozeßstart mitgegeben werden kann (zur Parameterübergabe vgl. die Beschreibung am Ende von Kapitel 11.3.2)

`PROCEDURE MTKillTask (Task : TaskPtr);`	Eine Task wird aus der Task-Verwaltung entfernt

Bedeutung des Parameters:

`Task` Zeiger auf den TCB der zu entfernenden Task

`PROCEDURE MTYield;`	Die aufrufende Task gibt die Programmausführung freiwillig an die nächste Task im Zustand BEREIT ab

`PROCEDURE MTChangePrio ( Task : TaskPtr; NewPrio : Byte);`	Die Priorität einer Task wird geändert.

Bedeutung der Parameter:

`Task`	Zeiger auf den TCB der Task, deren Priorität geändert wird
`NewPrio`	neue externe Priorität

`PROCEDURE MTBlock;`	Der Taskwechselmechanismus wird ab sofort unterbunden. ***Die Prozedur sollte von einer Anwendung nicht aufgerufen werden.***
`PROCEDURE MTContinue;`	Der Taskwechselmechanismus wird ab sofort wieder aktiviert. ***Die Prozedur sollte von einer Anwendung nicht aufgerufen werden (außer sie hat vorher den Taskwechselmechanismus außer Kraft gesetzt).***
`PROCEDURE MTWait (Ptr : PSyncelement);`	Die aufrufende Task wird in den Zustand WARTEND versetzt und eine andere Task, die sich im Zustand BEREIT befindet, wird aktiviert.

Bedeutung des Parameters:

`Ptr`	Zeiger auf das Ereignis (Semaphor, Nachricht usw.), auf das die Task wartet

`PROCEDURE MTSignal (Task : TaskPtr);`	Einer wartenden Task (im Zustand WARTEND) wird das Eintreten des Ereignisses signalisiert, wodurch sie in den Zustand BEREIT versetzt wird.

Bedeutung des Parameters:

`Task`	Zeiger auf den TCB der wartenden Task

`FUNCTION MTRegisterUNIT (Create : UpCreate; Change : UpChange; Delete : UpDelete) : BOOLEAN;`	Die Unit, aus der der Aufruf erfolgt, wird als kooperierende Unit des Multitaskings registriert (siehe obigen Text). Der Rückgabewert ist `TRUE`, falls die Unit registriert werden konnte (maximale Anzahl registrierbarer Units = `max_units`, siehe Unit `koopUNIT`)

Bedeutung der Parameter:

`Create`	Prozedur, die bei Einrichtung einer Task in der kooperierenden Unit aufgerufen wird
`Change`	Prozedur, die bei jedem Taskwechsel in der kooperierenden Unit aufgerufen wird
`Delete`	Prozedur, die beim Entfernen einer Task in der kooperierenden Unit aufgerufen wird

Die Prozeduren und Funktionen der **Unit `MultiWin` zur Verwaltung von Bildschirmfenstern** zeigt die folgende Tabelle. Die vollständige Benutzerschnittstelle der Unit WinMt mit der `INCLUDE`-Datei `Win2Mt`, die durch die Unit `MultiWin` angebunden wird, kann man in [TIS] nachlesen.

Typen (im INTERFACE-Teil der Unit MultiWin)	Bedeutung
`Rahmentyp = (einfacher_Rahmen, doppelter_Rahmen, gepunkteter_Rahmen, voller_Rahmen);`	Verschiedene Fensterumrahmungstypen;

Prozedur/Funktion (im INTERFACE-Teil der Unit MultiWin)	Bedeutung
`FUNCTION TaskWinOpen (x1, y1, x2, y2 : Byte; Rahmen : Rahmentyp; Titel : STRING ) : INTEGER;`	Ein Bildschirmfenster (Puffer) wird eingerichtet und ein Handle (Identifizierung) auf dieses Fenster zurückgegeben.

Bedeutung der Parameter:

`x1, y1, x2, y2`	Bildschirmkoordinaten (`x` = Spaltennummer, `y` = Zeilennumer; die Koordinaten der linken oberen Bildschirmecke lauten `x` = 1, `y` = 1)
`Rahmen`	Art der Fensterumrahmung
`Titel`	Fensterbeschriftung

`PROCEDURE TaskWinClose (Redraw : BOOLEAN);`	Das zuletzt geöffnete Bildschirmfenster wird geschlossen. Dabei wird vorausgesetzt, daß noch mindestens ein Fenster geöffnet ist.

Bedeutung des Parameters:

`Redraw`	= `TRUE` : der Inhalt des Bildschirmbereichs unter dem zu schließenden Fenster wird wieder auf den Bildschirm gebracht

`FUNCTION TaskWinInFront (Handle : INTEGER) : BOOLEAN;`	Ein Fenster des Prozesses wird zum Vordergrundfenster. Bei erfolgreicher Ausführung liefert die Funktion den Wert `TRUE`. Sie liefert den Wert `FALSE`, falls zu wenig Speicher zur Ausführung der Funktion zur Verfügung stand.

Bedeutung des Parameters:

`Handle`	Handle des Fensters (siehe `TaskWinOpen`), das zum Vordergrundfenster werden soll

`PROCEDURE TaskWinSetCursor;`	Der Bildschirmcursor wird in das Vordergrundfenster der Task positioniert und sichtbar gemacht.
`PROCEDURE TaskWinHideCursor;`	Der Bildschirmcursor wird vom Bildschirm entfernt.

`PROCEDURE TaskWinGotoXY (x, y : INTEGER);`	Die Prozedur ersetzt die Prozedur `GotoXY` der Unit `Crt`. Sie legt die Ausgabeposition für den nächsten Aufruf der `Write/Writeln`-Prozedur fest.

Bedeutung der Parameter:

`x`	Ausgabespalte
`y`	Ausgabezeile

`FUNCTION TaskWinWhereX : INTEGER;`	Die Funktion ersetzt die Funktion `WhereX` der Unit `Crt`. Sie liefert die Ausgabespalte des nächsten `Write/Writeln`-Befehls.
`FUNCTION TaskWinWhereY : INTEGER;`	Die Funktion ersetzt die Funktion `WhereY` der Unit `Crt`. Sie liefert die Ausgabezeile des nächsten `Write/Writeln`-Befehls.
`PROCEDURE TaskWinWriteStr (txt : STRING; attr : Byte; backgrd : Byte);`	Eine Zeichenkette wird mit den angegebenen Attributen (Textattribut und Hintergrund) an die aktuelle Cursorposition geschrieben.

Bedeutung der Parameter:

`txt`	auszugebender Text
`attr`	Attribute des Texts[38]
`backgrd`	Attribute des Hintergrunds

`PROCEDURE TaskWinClrScr;`	Der Bildschirm wird gelöscht.

Abbildung 11.2-1 benennt alle Units, die zur Realisierung einer Multitasking-Anwendung im vorliegenden Modell eingebunden werden müssen. Dabei sind bereits Units angegeben, die zur Prozeßsynchronisation mit Semaphoren (Unit `Semaphor`, Kapitel 11.4.1) bzw. Ereignisvariablen (Unit `Ereignis`, Kapitel 11.4.2) und für die Interprozeßkommunikation (Unit `IPK`, Kapitel 11.5) verwendet werden.

---

[38]

```
CONST
{ Foreground and background
  color constants }

  Black         = 0;
  Blue          = 1;
  Green         = 2;
  Cyan          = 3;
  Red           = 4;
  Magenta       = 5;
  Brown         = 6;
  LightGray     = 7;

{ Foreground color constants }

  DarkGray      = 8;
  LightBlue     = 9;
  LightGreen    = 10;
  LightCyan     = 11;
  LightRed      = 12;
  LightMagenta  = 13;
  Yellow        = 14;
  White         = 15;

{ Add-in for blinking }

  Blink         = 128;
```

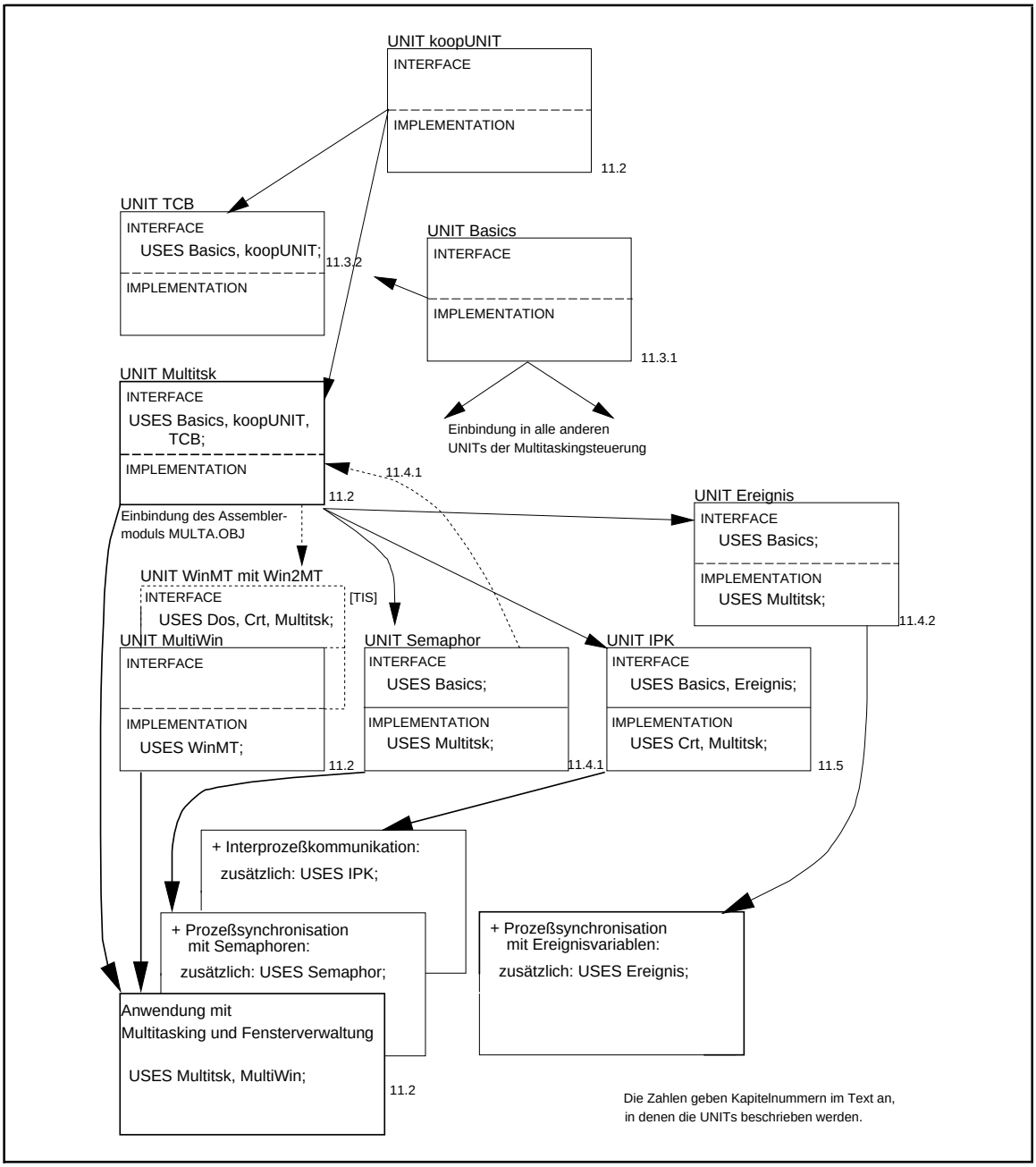


Abbildung 11.2-1: Anwenderschnittstelle des Multitaskings

Eine ***Anwendung, die nur Multitasking mit Fensterverwaltung durchführt***, enthält beispielsweise die Anweisung

```
USES Multitsk, MultiWin;
```

eine ***Anwendung mit Multitasking, Fenster für die einzelnen Prozesse, Prozeßsynchronisation mit Semaphoren und Interprozeßkommunikation*** enthält entsprechend die Anweisung

```
USES Multitsk, MultiWin, Semaphor, IPK;
```

Die einzelnen Units führen z.T. USES-Anweisungen in ihrem INTERFACE- und im IMPLEMENTATION-Teil auf. Zu beachten ist, daß eine Anwendung einige Units nicht einbindet, obwohl sie implizit verwendet werden. Dazu gehören die Units TCB, koopUNIT und die Unit Basics, die Objekttypen zur Realisierung von Listen, FIFO-Warteschlangen und Synchronisationshilfsmittel definiert.

11.3 Prozeßverwaltung, Prozeßwechsel und Scheduling

In diesem Kapitel werden einige wichtige internen Abläufe der Prozeßverwaltung, nämlich die funktionalen Zusammenhänge, die dabei verwendeten Datenstrukturen und insbesondere der Prozeßwechsel, überblicksmäßig beschrieben.

Es wird wieder die in Abbildung 11-4 gezeigte Anwendung betrachtet, d.h. in einem Rahmenprogramm sind alle Prozeduren enthalten, die von den einzelnen Prozessen durchlaufen werden, und zusätzlich sind die Unit Multitsk und eventuell weitere benötigte Units eingebunden. Vor dem ersten Aufruf der Funktion MTCreateTask zur Einrichtung von Prozessen hat dann das Laufzeitlayout das in Abbildung 11.3-1 gezeigte schematische Aussehen (vgl. Kapitel 2.1 und 3.4).

Vor der Beschreibung der Abläufe in der Multitasking-Kontrolle (Kapitel 11.3.3) werden einige verwendete grundlegende Datenobjekttypen (Kapitel 11.3.1) und die Details des TCBs eines Prozesses (Kapitel 11.3.2) betrachtet.

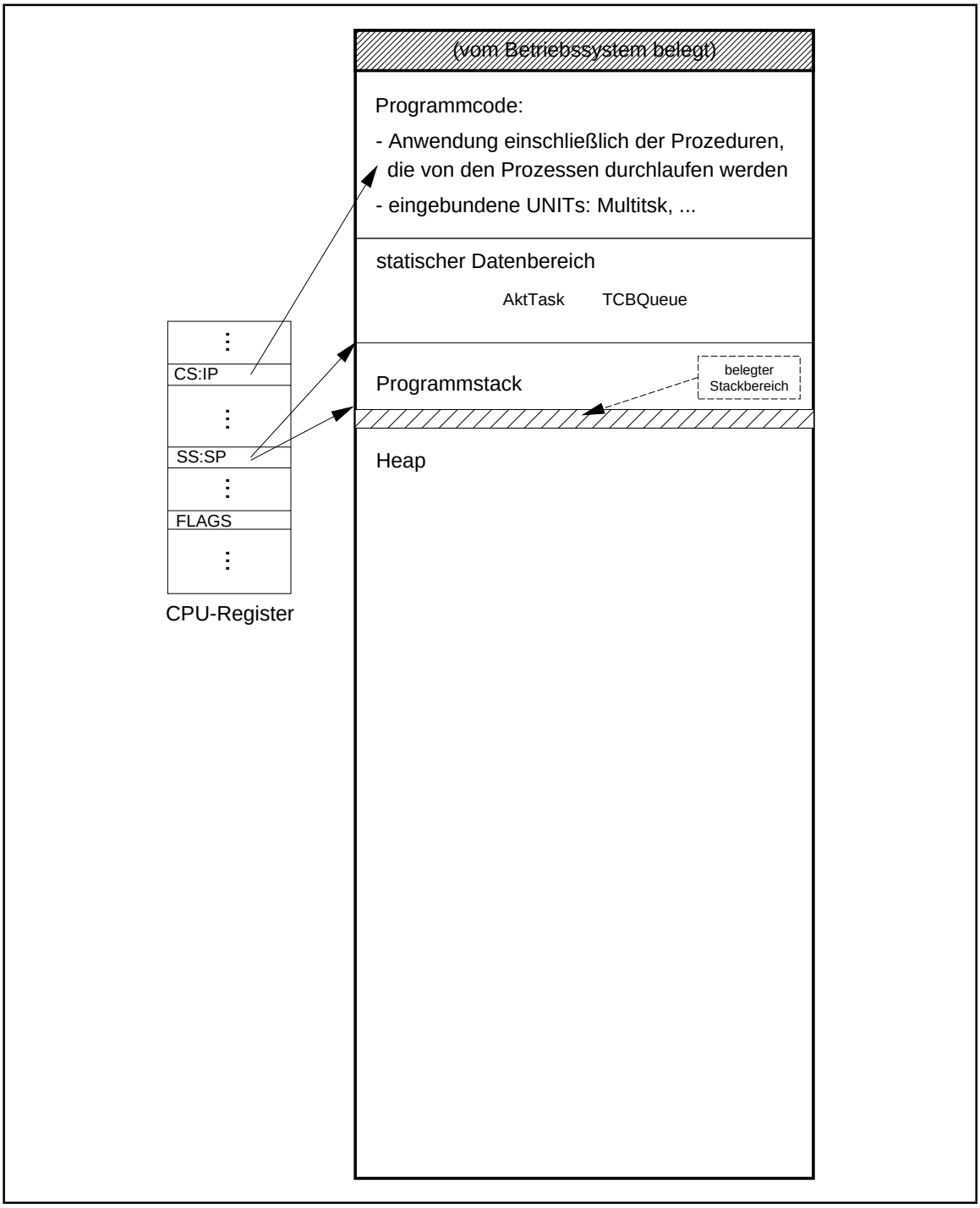


Abbildung 11.3-1: Laufzeitlayout einer Anwendung mit Multitasking vor dem Start der Multitaskingsteuerung (schematische Darstellung)

11.3.1 Die Unit Basics

Einige Objekttypdeklarationen werden in allen Units der Multitasking-Steuerung benötigt und sind daher in einer Unit namens `Basics` zusammengefaßt. Es sind die Objekttypen

```
TYPE TListe = OBJECT ...;
     TFIFO  = OBJECT (TListe)...;

     TSyncelement = ...;
```

Der Objekttyp `TSyncelement` wird in Kapitel 11.4 beschrieben. Die Objekttypen `TListe` bzw. `TFIFO` entsprechen weitgehend den in Kapitel 10.1.1 bzw. 10.1.2 beschriebenen Objekttypen. Die wichtigsten Änderungen sind: Eine Liste besitzt vorwärts- und rückwärtsverkettete Einträge; ein Eintrag in eine Liste kann einen beliebigen Datentyp haben, denn er wird über eine Komponente `entry_ptr` vom Typ `Pointer` adressiert; `TFIFO` ist nun von `TListe` abgeleitet; beide Objekttypen wurden um einige Methoden erweitert. **Alle Methoden der Multitasking-Kontrolle verwenden die hier beschriebenen Objekttypen.**

```
TYPE
     { Liste: }
     TList_flag = (alle, erstes, c_first, c_next);
     PListe = ^TListe;
     TListe = OBJECT
                first      : Pointer;
                           { Anfang der Verkettung der Verweise auf
                             Einträge der Liste                    }
                last       : Pointer;
                           { Ende der Liste                        }
                lastchosen : Pointer;
                           { Eintrag, der beim letzten Aufruf der
                             Methode choose ermittelt wurde        }
                anz        : INTEGER;
                           { Anzahl der Listeneinträge             }
                CONSTRUCTOR init;
                PROCEDURE insert (entry_ptr : Pointer);
                PROCEDURE delete (flag      : TList_flag;
                                  entry_ptr : Pointer);
                FUNCTION choose (param : TList_flag) : Pointer;
                FUNCTION count : INTEGER;
                DESTRUCTOR done; VIRTUAL;
              END;

     { FIFO-Warteschlange: }
     PFIFO = ^TFIFO;
     TFIFO = OBJECT (TListe)
               PROCEDURE delete (VAR entry_ptr : Pointer);
               PROCEDURE delete_entry (entry_ptr : Pointer);
             END;
```

Die Methoden beider Objekttypen haben folgende Bedeutung:

Methode	Bedeutung
`CONSTRUCTOR TListe.Init;`	Die Liste wird eingerichtet.
`PROCEDURE TListe.insert (entry_ptr : Pointer);`	Ein neuer Eintrag wird in die Liste aufgenommen.
Bedeutung des Parameters: `entry_ptr` Zeiger auf den neuen Eintrag	
`PROCEDURE TListe.delete (flag : TList_flag; entry_ptr: Pointer);`	Ein Listeneintrag wird entfernt.
Bedeutung der Parameter: `flag` Angabe, ob alle Vorkommen des Eintrags entfernt werden sollen (`flag = alle`) bzw. nur das erste Vorkommen (`flag = erstes`) `entry_ptr` Zeiger auf das zu entfernende Element	
`FUNCTION TListe.choose (param : TList_flag) : Pointer;`	Bei jedem Aufruf wird ein Listeneintrag ermittelt, wobei der Parameter `param` angibt, von welcher Stelle innerhalb der Liste der Listeneintrag genommen werden soll.
Bedeutung des Parameters: `param` = `c_first`: erster Listeneintrag = `c_next` : der dem letzten Aufruf von `choose` folgende Listeneintrag	
`FUNCTION TListe.count : INTEGER;`	Die Funktion liefert die aktuelle Anzahl an Listeneinträgen.
`DESTRUCTOR TListe.done;`	Die Liste wird entfernt.

`PROCEDURE TFIFO.delete (VAR entry_ptr : Pointer);`	Ein Eintrag wird gemäß dem FIFO-Prinzip aus der FIFO-Warteschlange entfernt
Bedeutung des Parameters: `entry_ptr` Zeiger auf das aus der FIFO-Warteschlange entfernte Element; falls die FIFO-Warteschlange leer ist, wird der Wert `NIL` zurückgegeben.	
`PROCEDURE TFIFO.delete_entry (entry_ptr : Pointer);`	Ein genau spezifizierter Eintrag wird aus der FIFO-Warteschlange entfernt. Der Aufruf entspricht dem Aufruf `TListe.delete (erstes, entry_ptr);`
Bedeutung des Parameters: `entry_ptr` Zeiger auf das aus der FIFO-Warteschlange zu entfernende Element.	

Auf eine Darstellung der Implementierung der Methoden wird hier verzichtet, da ähnliche Mechanismen in Kapitel 10.4.1 und 10.4.2 beschrieben werden. Zur Realisierung der doppelten Verkettung innerhalb der Liste ist der dort aufgeführte Objekttyp `TListentry` zu

```
TYPE PListentry = ^TListentry;
     TListentry = RECORD
                    entry_ptr : Pointer;
                    next      : PListentry;  { Vorwärtsverkettung   }
                    last      : PListentry;  { Rückwärtsverkettung  }
                  END;
```

erweitert worden.

11.3.2 Der Task Controll Block (TCB) eines Prozesses

Die Multitasking-Kontrolle identifiziert einen Prozeß durch einen Zeiger (vom Typ `TaskPtr`) auf seinen TCB.

Jeder Prozeß besitzt **seinen eigenen Stack**, der zur Realisierung seiner Prozeduraufrufe und zur temporären Zwischenspeicherung der Registerinhalte verwendet wird, wenn der Prozeß die CPU aufgrund eines Prozeßwechsels (aus dem Zustand AKTIV in den Zustand BEREIT oder WARTEND) an eine andere Task abgeben muß. Die Registerinhalte werden also beim Prozeßwechsel nicht in einem virtuellen Registersatz (vgl. Kapitel 3.2), der Teil des TCB sein könnte, sondern im prozeßeigenen Stack gesichert.

Bei Prozeßeinrichtung (Aufruf von `MTCreateTask` im `INTERFACE`-Teil der Unit `Multitsk`) wird der TCB des Prozesses über seinen Konstruktor initialisiert. Diese Initialisierung erfolgt durch Aufruf des entsprechenden Konstruktors `Init` im Prozedurrumpfs von `MTCreateTask`. Die beim `Init`-Aufruf zu verwendenden Aktualparameter werden von den Aktualparametern aus dem `MTCreateTask`-Aufruf übernommen.

Bei der Initialisierung des TCBs wird Platz für den Stack des Prozesses auf dem Heap mit Hilfe der Pascal-Prozedur `GetMem` reserviert. Die Stackgröße eines einzurichtenden Prozesses wird vom Anwender festgelegt. Die Segmentadresse des Stacks und der Stackpointer, d.h. der Offset des obersten Bytes relativ zum Anfang des Stacks, sowie die Stacklänge werden im TCB eingetragen (Komponenten `Stack`, `SP` bzw. `StackLen`).

Die aktuelle externe Prozeßpriorität wird in die TCB-Komponente `Priority` eingetragen.

Der aktuelle Prozeßzustand wird nicht explizit festgehalten, sondern er ergibt sich implizit je nach Wert einiger Einträge im TCB bzw. Aufenthaltsort des Prozesses im System: Belegt der Prozeß gerade die CPU, d.h. werden Anweisungen seiner Prozeduren ausgeführt, ist er im Zustand AKTIV. Immer nur ein Prozeß kann zu einem Zeitpunkt in diesem Zustand sein. Der Zustand WARTEND ist dadurch gekennzeichnet, daß der Prozeß auf das Eintreten eines Ereignisses wartet, das durch den Eintrag `WaitPtr` in seinem TCB adressiert wird. Steht dort der Wert `NIL` und ist der Prozeß nicht AKTIV, so befindet er sich im Zustand BEREIT und wird als potentieller Kandidat für den nächsten Prozeßwechsel angesehen.

Ein Prozeß betritt während seines Ablaufs eventuell einen oder mehrere, u.U. logisch ineinander geschachtelte sogenannte kritische Abschnitte (siehe Kapitel 11.4), in denen er exklusives Zugriffsrecht auf globale Ressourcen besitzt. Zur Realisierung einer hierbei erforderlichen Prozeßsynchronisation werden gelegentlich Semaphore (Kapitel 11.4.1) eingesetzt. Terminiert ein Prozeß vorzeitig oder regulär, noch bevor er den exklusiven Zugriff auf Ressourcen beendet, müssen diese explizit freigegeben werden. Andere Prozesse könnten ja ihrerseits auf die exklusive Zugriffsmöglichkeit der Ressourcen warten. Die bei Prozeßterminierung noch freizugebenden Ressourcen, die der Prozeß zuvor mit Hilfe von Semaphoren reserviert hatte, werden über die Einträge in der FIFO-Liste gefunden, die durch die TCB-Komponente PassedSem adressiert ist. Weitere Details werden in Kapitel 11.4.1 beschrieben.

Zur Realisierung des Nachrichtenaustauschs mit anderen Prozessen enthält der TCB in der Komponente Ports einen Verweis auf eine Liste von Datenobjekten, die seine Verbindungen zur "Außenwelt" beschreiben und als Ports bezeichnet werden. Die Details des Nachrichtenaustauschs werden in Kapitel 11.5 behandelt.

Jeder Prozeß erhält bei Einrichtung von der Prozedur `MTCreateTask` als Prozeßidentifikation eine aufsteigend vergebene eindeutige Nummer, die in die TCB-Komponente `TaskID` eingetragen, jedoch von der Multitasking-Kontrolle zur Zeit nicht genutzt wird.

Die TCB-Komponente `StartTime` gibt den Zeitpunkt an, an dem der Prozeß zuletzt aktiviert wurde. Der Zeitpunkt wird als aktueller Wert der logischen Uhr (typisierte Konstante mit Bezeichner Aufrufe in der Prozedur `MTChooseNext` der Unit `Multitsk`) festgehalten.

Damit einzelne Abläufe des Betriebssystems eventuell gezielt Informationen an einen Prozeß übermitteln können, enthält der TCB die Komponente `Parameter`, deren Wert durch die Methoden `PutParameter` bzw. `GetParameter` gesetzt bzw. gelesen werden kann.

Im TCB befindet sich ein weiterer Eintrag mit Bezeichner `UNITData`. In diesem `ARRAY` werden die Rückgabewerte festgehalten, die kooperierende Units bei der Prozeßeinrichtung durch Aufruf der Unit-spezifischen Funktionen `UNITproz.CreateProz` ermitteln.

Der Objekttyp `TD` verwendet in seiner Deklaration einen Prozedurtyp `TS`. Die Deklarationen beider Typen lauten:

```
TYPE TS = PROCEDURE (Zeiger : Pointer);

     TD  = OBJECT     { Task-Descriptor }
             Stack     : Pointer;
                         { Segmentadresse des Stacks          }
             SP        : WORD; { Stackpointer bei Taskunterbrechung }
             StackLen  : INTEGER;
                         { Größe des Stacks in Byte           }
             Priority  : BYTE; { externe Priorität            }
```

```
  WaitPtr   : PSyncelement;
                    { Zeiger auf zu erwartendes Ereignis,
                      z.B. Semaphor, Nachricht usw.      }
  PassedSem : PFIFO;{ Zeiger auf Semaphore, die passiert
                      wurden, ohne den durch sie
                      geschützten Bereich wieder zu
                      verlassen                          }
  Ports     : PFIFO;{ Zeiger auf Kommunikationsports     }
  TaskId    : INTEGER;
                    { Prozeßnummer                       }
  StartTime : Longint;
                    { Zeitpunkt des letzten Starts       }
  Parameter : Pointer;
                    { Parameter für Betriebssystemabläufe}
  UNITData  : ARRAY [1..max_units] OF Pointer;
                    { Unit-individuelle Information, die
                      von den mitarbeitenden Units über
                      ihre CreateProz erzeugt wird       }
  CONSTRUCTOR Init (Startproc : TS;
                    EndProc   : Pointer;
                    Prio      : BYTE;
                    SLen      : INTEGER;
                    Id        : INTEGER;
                    Zeiger    : Pointer );
  PROCEDURE ChangePrio (NewPrio : BYTE);
  PROCEDURE SetWaiting (Ptr : PSyncelement);
  PROCEDURE SetReady (VAR Ptr : PSyncelement);
  PROCEDURE PutParameter (Ptr : Pointer);
  FUNCTION GetParameter : Pointer;
  DESTRUCTOR Done;
END;
```

Da die Unit `Multitsk` auf die Komponenten des Objekttyps `TD` zugreifen muß, werden diese nicht als `PRIVATE` deklariert. Alle Methoden des TCBs werden nur aus der Unit `Multitsk` angestoßen. Die Struktur des TCBs eines Prozesses darf für einen Anwenderprozeß nicht sichtbar sein. Die Unit `TCB` wird daher in eine Anwendung nicht eingebunden. Andererseits wird eine Task ja durch einen Verweis auf ihren TCB identifiziert. Die entsprechende Typdeklaration

```
TYPE TaskPtr = ^TD;
```

befindet sich daher im `INTERFACE`-Teil der Unit `Multitsk`. Als Konsequenz dieses Ansatzes (der ausschließlichen Verwendung der Unit Multitsk auf Anwenderebene) ergibt sich, daß einige Funktionen, die konzeptionell als Methoden des TCBs anzusehen sind, in nicht-objektorientierter Weise als Prozeduren/Funktionen im `INTERFACE`-Teil der Unit `Multitsk` implementiert sind. Auf den TCB gibt es ja keinen Zugriff. Typischerweise sind z.B. die Operationen zur Änderung einer Prozeßpriorität bzw. zur Entfernung eines Prozesses aus der Task-Verwaltung Methoden des TCBs; im vorliegenden Ansatz gibt es für diese Operationen die "klassischen" Prozeduren `MTChangePrio` bzw. `MTKillTask` in der Anwenderschnittstelle der Unit `Multitsk`.

Die folgende Tabelle faßt die Methoden und die Bedeutung der Parameter des Objekttyps `TD` zusammen:

Methode	Bedeutung
`CONSTRUCTOR TD.Init (Startproc : TS; EndProc : Pointer; Prio : BYTE; SLen : INTEGER; Id : INTEGER; Zeiger : Pointer );`	Der TCB eines Prozesses und sein Stack werden eingerichtet und initialisiert (siehe nachfolgende Beschreibung).

Bedeutung der Parameter:

`Startproc`	Prozedur, mit der der Prozeß startet
`Endproc`	Zeiger auf eine Prozedur, die nach Prozeßbeendigung abläuft
`Prio`	externe Prozeßpriorität
`SLen`	Länge des zu reservierenden prozeßeigenen Stacks in Bytes
`Id`	numerische Prozeßidentifikation
`Zeiger`	Zeiger auf einen Parametersatz (Einzelparameter oder `RECORD`-Typ), der dem Prozeß bei Prozeßstart mitgegeben werden kann

`PROCEDURE TD.ChangePrio (NewPrio : BYTE);`	Der Prozeß erhält eine neue Priorität.

Bedeutung des Parameters:

`NewPrio`	Neue Prozeßpriorität

`PROCEDURE TD.SetWaiting (Ptr : PSyncelement);`	Der Prozeß wird aus dem Zustand AKTIV in den Zustand WARTEND versetzt.

Bedeutung des Parameters:

`Ptr`	Zeiger ($\neq$ `NIL`) auf das zu erwartende Ereignis

`PROCEDURE TD.SetReady (VAR Ptr : PSyncelement);`	Der Prozeß wird aus dem Zustand WARTEND in den Zustand BEREIT versetzt, weil das Ereignis eingetroffen ist.

Bedeutung des Parameters:

`Ptr`	Zeiger auf das Ereignis, auf das bisher gewartet wurde

`PROCEDURE TD.PutParameter (VAR Ptr : Pointer);`	Die Komponente Parameter wird gesetzt.

Bedeutung des Parameters:

`Ptr`	Wert für die Komponente Parameter

`FUNCTION TD.GetParameter : Pointer;`	Der Wert der Komponente Parameter wird zurückgegeben.
`DESTRUCTOR TD.Done;`	Eventuelle Verbindungen zur Interprozeßkommunikation werden gelöst (siehe Kapitel 11.5), der für den Prozeß reservierte Stack wird freigegeben und der TCB aus dem System entfernt.

Bei der Einrichtung eines TCBs werden bereits Einträge im Stack des Prozesses vorgenommen: es sind nacheinander nach absteigenden Adressen die Zeiger auf den Parametersatz bei Prozeßstart und auf die Prozedur, die bei Prozeßende ablaufen soll, der Inhalt des FLAGS-Registers bei Prozeßstart, die Adresse der Startprozedur und die mit 0 initialisierten Registerinhalte. Auf diese Weise enthält der Stack vor dem Start des Prozesses bereits einen Eintrag, der genauso strukturiert ist wie ein Eintrag, der dort bei einem Prozeßwechsel abgelegt wird. Bei einem Prozeßwechsel wird nämlich im Stack des Prozesses, der den Zustand AKTIV verläßt, (in dieser Reihenfolge) der Inhalt des FLAG-Registers, die Prozeßfortsetzungsadresse und die gegenwärtigen Inhalte der allgemeinen Daten- und Adreßregister der CPU (vgl. Abbildung 2.1-1) eingetragen. Bei Prozeßstart entspricht der Prozeßfortsetzungsadresse die Startadresse des Prozesses, und die Register enthalten den Anfangswert 0.

Die Adreßverkettungen nach Ablauf eines `CONSTRUCTOR`-Aufrufs `TD.init` zeigen Abbildung 11.3.2-1.

Zu beachten sind noch die Auswirkungen, die sich durch die vom Konstruktor `TD.Init` an unterster Stelle im Stack, also an den höchsten Adressen, eingetragenen Werte ergeben. Diese Werte sind in Abbildung 11.3.2-1 durch `MTaTaskEnded` und `@Data` benannt. `MTaTaskEnded` wird einheitlich durch `MTCreateTask` für jeden einzurichtenden Prozeß an den Konstruktor übergeben; `@Data` ist der Wert, der für den neuen Prozeß im `MTCreateTask`-Aufruf im Formalparameter `Zeiger` als Aktualparameter gesetzt ist.

Unmittelbar vor Eintritt in die Startprozedur, die ja das Hauptprogramm des Prozesses darstellt und dessen Ablaufende auch das Prozeßende ist, hat der Prozeßwechselmechanismus der Multitasking-Kontrolle alle Stackeinträge, bis auf die Einträge `MTaTaskEnded` und `@Data` entfernt, so daß sie jetzt auf oberster Position im Stack liegen. Da eine Prozedur bei ihrem Start erwartet, daß der Aufrufer im Stack die Rücksprungadresse zum Aufrufer und darunter (in Richtung aufsteigender Adressen) die Aktualparameter für den Prozeduraufruf abgelegt hat (vgl. Abbildung 7.5-2), ist `MTaTaskEnded` aus Sicht der Startprozedur die Rücksprungadresse zu einem Aufrufer, den es aber eigentlich gar nicht gibt. `@Data` fungiert als Aktualparameter für den Parameter `Zeiger` im Aufruf der Startprozedur, die den Typ

```
TYPE TS = PROCEDURE (Zeiger : Pointer);
```

hat, also einen call-by-value-Parameter vom Typ `Pointer` erwartet.

Folglich verzweigt die Kontrolle am Ende des Ablaufs der Startprozedur eines jeden Prozesses, d.h. bei der regulären Prozeßbeendigung, in die Prozedur `MTaTaskEnded`. Diese befindet sich im Assemblerteil der Unit `Multitsk` und bewirkt, daß der Prozeß aus dem System entfernt wird (Aufruf `MTKillTask`) und daß für den Fall, daß sich gerade der insgesamt letzte Prozeß beendet, das Multitasking überhaupt beendet wird. In diesem Fall erfolgt ein Rücksprung zum Aufrufer von `MTStart`.

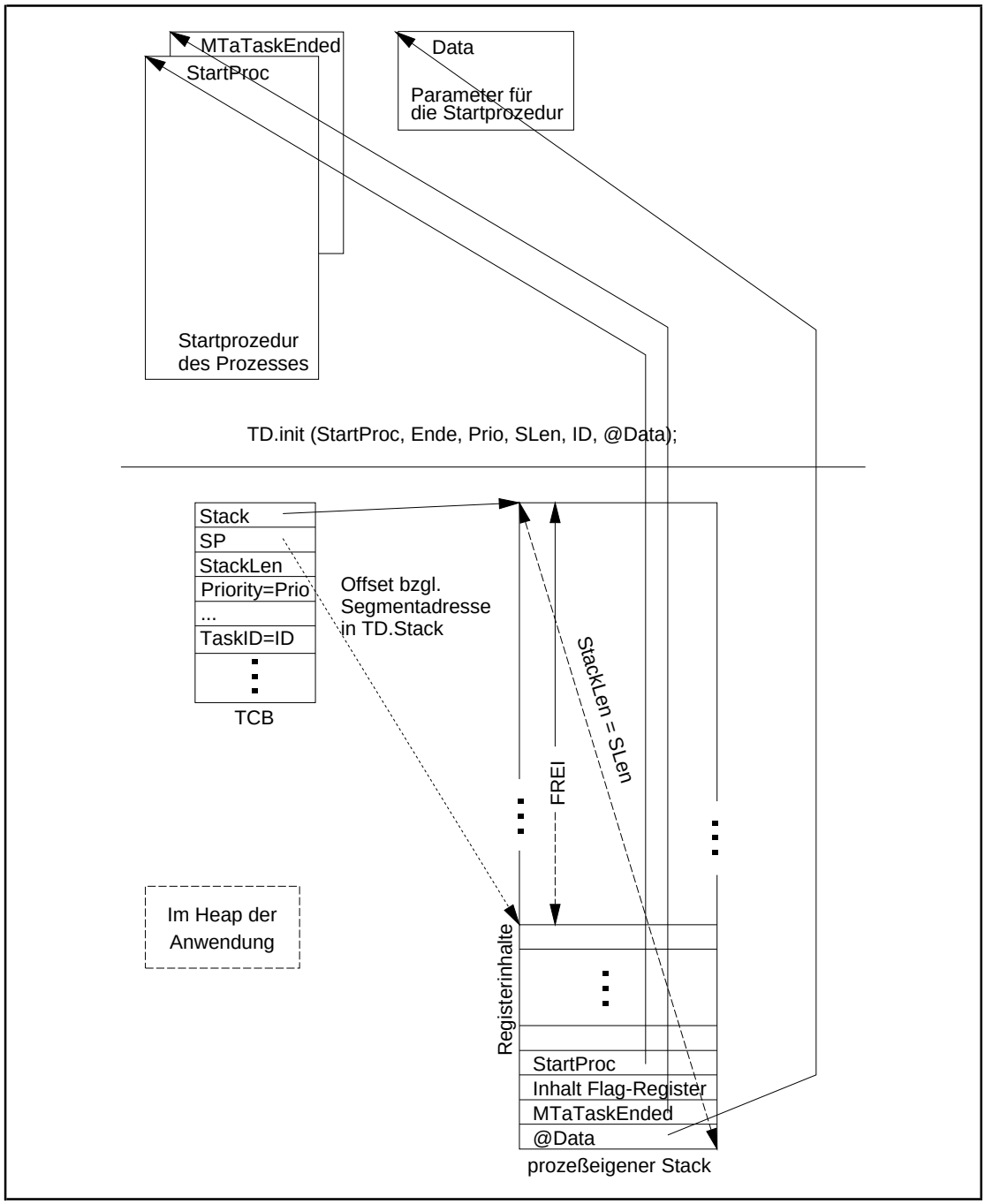


Abbildung 11.3.2-1: Methode `TD.init`

Die folgenden Beispiele zeigen, wie man der Startprozedur eines Prozesses auch Parameterwerte übergeben kann, die nicht vom Datentyp `Pointer` sind. Dabei soll angenommen werden, daß die Startprozedur durch

```
{$F+}
PROCEDURE prozess_start (param : Pointer);
...
BEGIN { prozess_start }
  ...
END   { prozess_start };
{$F-}
```

deklariert wird. Soll der Startprozedur der Aktualparameter

```
VAR AktParam : Pointer;
```

übergeben werden, der auch innerhalb der Prozedur als Wert vom Typ `Pointer` interpretiert wird, so kann der entsprechende Aufruf zum Einrichten des Prozesses

```
MTCreateTask (prozess_start, ..., AktParam);
```

lauten. Möchte man allerdings der Startprozedur einen Aktualparameter mit einem anderen Datentyp, etwa vom Typ `INTEGER`, übergeben, so kann man sich eines Tricks bedienen: Da ein Datenobjekt vom Typ `INTEGER` ein Wort belegt, der Aktualparameter in der Startprozedur aber als Datenobjekt vom Typ `Pointer` zwei hintereinanderliegende Worte, nämlich den Segment- und den Offsetteil einer Adresse, kann der Aufrufer den zu übergebenden Aktualparameter in ein Datenobjekt vom Typ `Pointer` einkleiden und dann an die Startprozedur über den `MTCreateTask`-Aufruf weiterleiten. Die Startprozedur interpretiert dann den Aktualparameter nicht als Datenobjekt vom Typ `Pointer`, sondern als zwei Worte, von denen eines den übergebenen Aktualparameter enthält. Die Einkleidung des Aktualparameters in ein Datenobjekt vom Typ `Pointer` kann mit Hilfe der PASCAL-Standardfunktion **`Ptr`** erfolgen. Diese konvertiert zwei Angaben, die als Segment- und Offsetteil einer Adresse interpretiert werden, in ein entsprechendes Datenobjekt vom Typ `Pointer`. Das Aufrufformat von `Ptr` lautet:

```
FUNCTION Ptr (Seg, Ofs : Word) : Pointer;
```

Konkret lautet der entsprechende Codeteil:

```
VAR AktParam : INTEGER;
...
MTCreateTask (prozess_start, ..., Ptr (0, AktParam));
```

Der Aktualparameter wird also im Offsetteil der Adresse abgelegt, der auf dem Stack des Prozesses nach aufsteigenden Adressen gesehen vor dem Segmentteil eingetragen wird (d.h. der Segmentteil liegt logisch unter dem Offsetteil im Stack). Der Code zur Übernahme des Aktualparameters in der Startprozedur in die lokale Variable `i` vom Typ `INTEGER` kann dann etwa lauten:

```
{$F+}
PROCEDURE prozess_start (param : Pointer);

  VAR i : INTEGER;
  ...
BEGIN { prozess_start }
  i := INTEGER (param);
  ...
END   { prozess_start };
{$F-}
```

Soll ein Aktualparameter vom RECORD-Typ an die Startprozedur übergeben werden, kann man folgende sprachliche Konstruktion wählen:

```
...
TYPE Tparam = RECORD
                ...
              END;

...
VAR AktParam : Tparam;
...
{$F+}
PROCEDURE prozess_start (param : Pointer);

  VAR rec : Tparam;
  ...
BEGIN { prozess_start }
  rec := Tparam (param^);
  ...
END   { prozess_start };
{$F-}
...
MTCreateTask (prozess_start, ..., @aktptr);
...
```

11.3.3 Abläufe der Multitasking-Kontrolle

Die Multitasking-Steuerung beginnt mit dem Aufruf der Prozedur MTStart. Es ist eine reine Assemblerprozedur (der Quellcode befindet sich zusammen mit weiteren Assemblerprozeduren in der Datei MULTA.ASM; die übersetzte Objektcodedatei heißt MULTA.OBJ). Sie überprüft, ob bereits Prozesse mit MTCreateTask eingerichtet wurden; beim ersten Aufruf von MTCreateTask wurde gleichzeitig der Idleprozeß ins Leben gerufen.

Die TCBs aller mit MTCreateTask eingerichteten Prozesse werden in einer über die Zeigervariable

```
VAR TCBQueue : PListe; { Verweis auf die Liste der verketteten TCBs }
```

adressierten Liste gehalten. Bei Einrichtung eines neuen Prozesses wird dessen TCB-Adresse hier eingehängt. Die globale Variable

```
VAR AktTask   : TaskPtr;     { Verweis auf den TCB der aktiven Task }
```

zeigt auf den TCB der gerade aktiven Task. In der im `IMPLEMENTATION`-Teil der Unit `Multitsk` verborgenen Variablen

```
CONST AnzTask   : INTEGER = 0;   { Anzahl angelegter Tasks          }
```

wird die Anzahl existierender Prozesse festgehalten.

Existieren noch keine Prozesse, beendet sich `MTStart` sofort, so daß im folgenden davon ausgegangen werden kann, daß Prozesse im System eingerichtet worden sind.

Zunächst durchläuft `MTStart` eine Initialisierungsphase, deren wichtigste Funktion die Vorbereitung des Zeitscheibenmechanismus ist. Hierbei wird u.a. die Interruptbehandlungsroutine zum Timerinterrupt `$08` (siehe Abbildung 2.2-2) durch eine von zwei möglichen eigenen Interruptbehandlungsroutinen ersetzt je nachdem, ob der Wert des Parameters `Tempo` im `MTStart`-Aufruf ≥ 1 oder ≤ -1 ist. Auf diese Weise wird der Multitaskingmechanismus an den zyklisch vom System erzeugten Timerinterrupt angeschlossen. ***Die Größe der Zeitscheibe*** Δ ***ist der zeitliche Abstand zwischen zwei Timerinterrupts***: Der Timerinterrupt kommt von dem im Rechner eingebauten Timerbaustein, der bei den INTEL 80x86-Rechnern mit einer festen Frequenz von 1.193.182 Hz betrieben wird (vgl. [BAU]). Der Timerbaustein wird mit einem Startwert (Datentyp `Word`) geladen, der dann in dieser Frequenz dekrementiert wird. Beim Nulldurchgang löst der Timerbaustein den Interrupt aus. Der "normale" Startwert ist $FFFF_{16} = 65.535_{10}$, d.h. die normale Interruptfrequenz ist 1.193.182/65.535 = 18,207 Interrupts pro Sekunde. Die "normale" Zeitscheibe ist damit gleich $\Delta = \Delta_0 = 1/18{,}207$ sec. Die Behandlungsroutine zum Interrupt `$08`, die von `MTStart` für die normale Interruptbehandlungsroutine eingesetzt wird, wenn der Wert t des Parameters `Tempo` >1 ist, lädt den Timerbaustein mit dem kleineren Startwert 65.536 DIV t, so daß der Abstand zwischen den einzelnen Timerinterrupts kürzer ist; die Zeitscheibe ist nun $\Delta \approx 1/t \cdot \Delta_0$. Für $t = 1$ wird der Startwert des Timerbausteins nicht geändert. Die Behandlungsroutine zum Interrupt `$08`, die von `MTStart` für die normale Interruptbehandlungsroutine eingesetzt wird, wenn der Wert t des Parameters `Tempo` ≤ -1 ist, programmiert den Timerbaustein nicht um, sondern wertet nur jeden $|t|$-ten Interrupt aus, so daß die Zeitscheibe auf den Wert $\Delta = |t| \cdot \Delta_0$ vergrößert wird.

Nach Installation der neuen Behandlungsroutine zum Timerinterrupt `$08` (Abschluß der Initialisierungsphase) wählt `MTStart` den ersten zu aktivierenden Prozeß aus und verzweigt an die Startadresse des Prozesses.

Der Auswahlvorgang des zu aktivierenden Prozesses ist in der Prozedur `MTChoose-Next` implementiert, die auch später zu jedem Prozeßwechsel angestoßen wird. Eine derartige Prozedur, die die Auswahl einer definierten **Prozessorzuteilungsstrategie (Schedulingstrategie)** implementiert, heißt **Scheduler**. `MTChooseNext` implementiert die in Kapitel 11.1 beschriebene zeitscheibengesteuerte Round-Robin--LRU-Schedulingstra- tegie. Sie wertet dazu die in den TCBs der eingerichteten Prozesse festgehaltenen externen Prioritäten und Zeitpunkte des letzten Prozeßstarts aus und

bestimmt die TCB-Adresse des Prozesses mit der höchsten internen Priorität. Der Aufrufer von `MTChooseNext` sorgt dafür, daß der Taskwechselmechanismus während des Ablaufs der Prozedur unterbrochen ist. `MTChooseNext` bestimmt bei jedem Aufruf einen Wert für die Variable `AktTask`:

```
PROCEDURE MTChooseNext;

CONST Aufrufe : Longint = 0;       { bisherige Aufrufe dieser Proz. }

VAR LookTask : TaskPtr;                 { aktuell verglichene Task  }
    MaxTask  : TaskPtr;     { Zeiger auf TCB mit höchster Priorität }
    AktPrio  : INTEGER;             { Priorität der aktuellen Task  }
    MaxPrio  : INTEGER;         { größte bisher gefundene Priorität }
    MinTime  : Longint;                 { früheste gestartete Task }
    i        : INTEGER;

BEGIN { MTChooseNext }
  { die Task suchen, der nicht geblockt ist, im Augenblick
    die höchste Priorität besitzt und die im Vergleich mit Tasks
    gleicher Priorität am längstem nicht mehr ausgeführt wurde.     }

  MaxPrio  := -1;                 { jede Priorität ist höher als -1 }
  MinTime  := Max_Longint;
  LookTask := TCBQueue^.choose (c_first);

  WHILE LookTask <> NIL DO
    BEGIN
      WITH LookTask^ DO
        BEGIN
          IF WaitPtr = NIL
          THEN { Task wartet auf kein Ereignis }
               IF Priority > MaxPrio
               THEN BEGIN { neue höhere Priorität }
                      MaxPrio := Priority;
                      MaxTask := LookTask;
                      MinTime := StartTime;
                    END
               ELSE IF (Priority = MaxPrio)
                       AND
                       ((StartTime < MinTime)
                          OR ((StartTime = MinTime)
                              AND (LookTask <> AktTask)))
                     THEN BEGIN
                            MaxTask := LookTask;
                            MinTime := StartTime;
                          END;
        END { WITH };
      LookTask := TCBQueue^.choose (c_next);
    END;

  IF MaxTask <> AktTask
  THEN BEGIN
        { kooperierende Units über den Prozeßwechsel informieren }
         FOR i := 1 TO AnzLUnits DO
            LUnits[i].ChangeProz (MaxTask^.UnitData[i]);
         AktTask := MaxTask;
         AktTask^.StartTime := Aufrufe;
       END;
  Inc( Aufrufe );
END { MTChooseNext };
```

Durch die Einrichtung eines Prozesses hat jeder Prozeß seinen eigenen Stack erhalten, der physikalisch im Heap der Anwendung liegt. Abbildung 11.3.3-1 zeigt das **Laufzeitlayout der Anwendung** mit den wichtigsten Adreßverkettungen, nachdem einige Prozesse eingerichtet wurden.

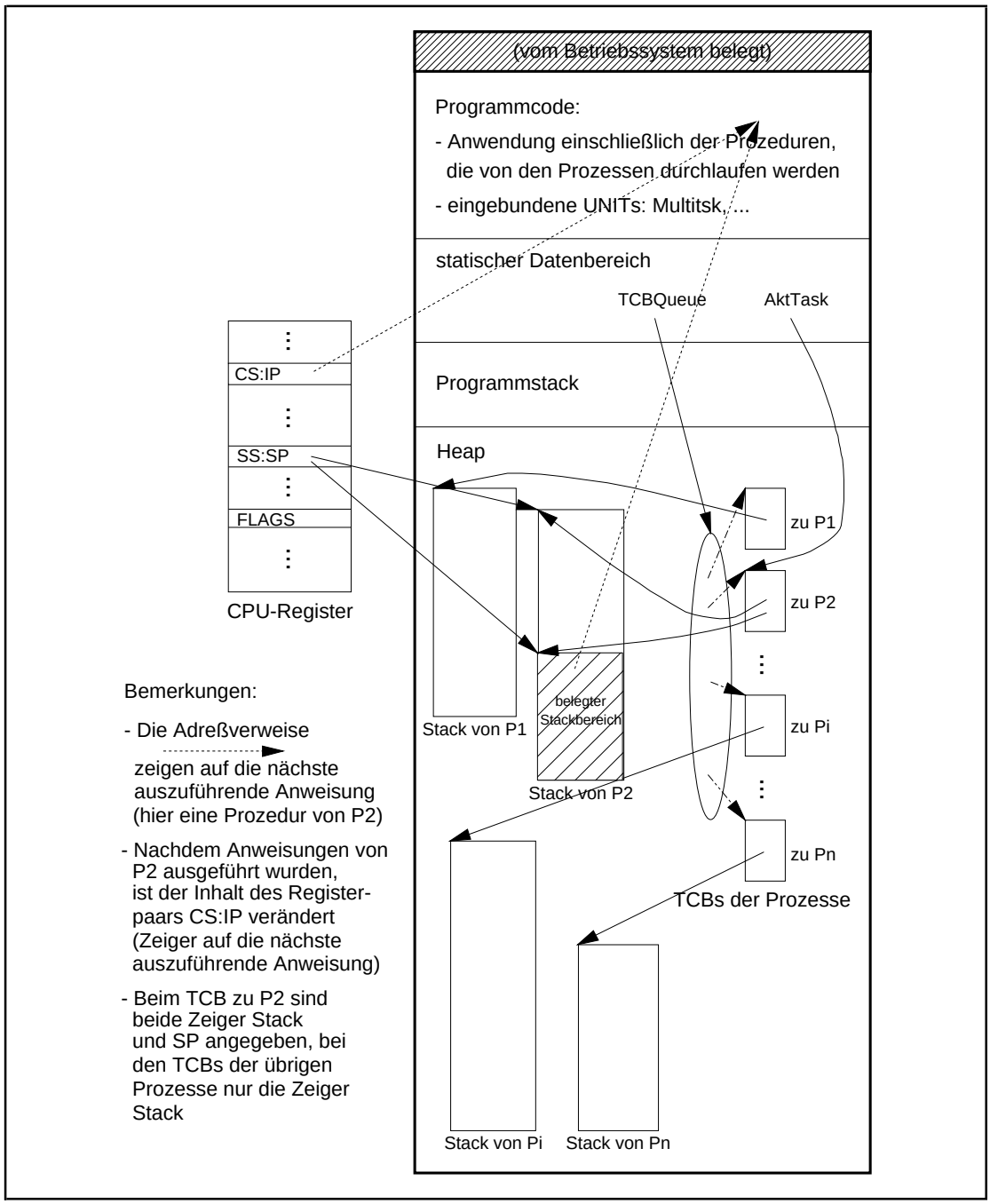


Abbildung 11.3.3-1: Laufzeitlayout einer Anwendung mit Multitasking nach Einrichtung einer Prozesse nach dem Start der Multitaskingsteuerung (schematische Darstellung)

Dabei wird angenommen, daß als aktive Task bereits der mit P2 bezeichnete Prozeß ausgewählt und unmittelbar vor der Ausführung seiner nächsten Anweisung steht. Die Inhalte der CPU-Register beziehen sich jetzt auf diesen Prozeß. Insbesondere adressiert das CPU-Registerpaar SS:SP (Stacksegmentregister SS, Stackpointer SP, vgl. Abbildung 2.1-1) den Stack des aktiven Prozesses, so daß sich alle Stackoperationen wie Unterprogrammsprünge mit der Ablage von Aktivierungsrecords und der Einrichtung lokaler Daten (siehe Kapitel 7.5) auf den Stack des aktiven Prozesses (und nicht etwa auf den Programmstack) beziehen. Das CPU-Befehlszählerregister (CS:IP) adressiert den nächsten im aktiven Prozeß auszuführenden Befehl, und der Inhalt des Anzeigenregisters FLAGS enthält die prozeßeigenen Anzeigenwerte des aktiven Prozesses.

Sobald der Timerinterrupt eintrifft, weil die Zeitscheibe Δ abgelaufen ist, wird das gerade laufende Programm unterbrochen, MTStart übernimmt wieder die Kontrolle und leitet einen Prozeßwechsel ein, falls der Prozeßwechselmechanismus nicht durch einen vorhergehenden Aufruf von `MTBlock` gerade deaktiviert ist. Ein Prozeßwechsel wird auch eingeleitet, wenn der laufende Prozeß diesen durch einen Aufruf von `MTYield` explizit veranlaßt oder falls er den ausgeschalteten Prozeßwechselmechanismus mit `MTContinue` wieder aktiviert und in der Zwischenzeit seine Zeitscheibe Δ abgelaufen ist (diese Situation wird daran erkannt, daß seit Deaktivierung des Prozeßwechselmechanismus ein Timerinterrupt eingetroffen ist, der aber nicht weiter behandelt wurde).

Beim Prozeßwechsel werden die Registerinhalte im Stack des zu deaktivierenden Prozesses gesichert. Ein derartiger Stackeintrag hat den in Abbildung 11.3.3-2 gezeigten Aufbau. Insbesondere wird die Fortsetzungsadresse des Prozeßlaufs an die Adresse $[SS:(SP+12_{16})]^{\wedge}$ (also im Stack des zu deaktivierenden Prozesses) abgelegt. Die TCB-Adresse des zu aktivierenden Prozesses wird von `MTChooseNext` in `AktTask` eingetragen, und der neue prozeßeigene Stack wird adressiert (im Assemblercode von `MULTA.ASM`). Die Stacksegmentadresse findet man dabei in

```
AktTask^.Stack
```

und den Offset des obersten Stackeintrags in

```
AktTask^.SP
```

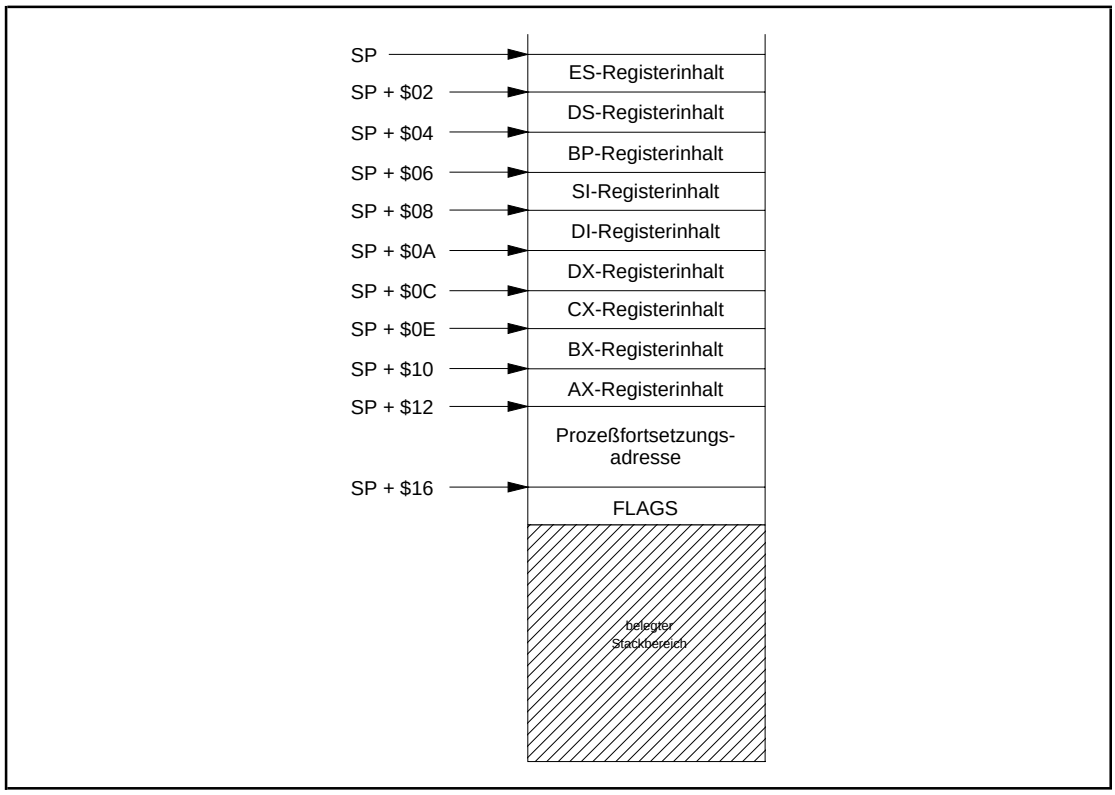


Abbildung 11.3.3-2: Stackeintrag bei Prozeßwechsel

Die Registerinhalte werden aus dem Stack des zu aktivierenden Prozesses geladen; die Prozeßfortsetzungsadresse kommt dabei in die CPU-Register CS:IP. Dabei werden die Registerinhalte vom Stack entfernt; SP in Abbildung 11.3.3-2 zeigt also jetzt auf den schraffierten Teil. Anschließend erfolgt ein Sprung an die Adresse in CS:IP und damit in das Programm des zu aktivierenden Prozesses.

Der Prozeßwechselmechanismus, der auch **Dispatcher** genannt wird, ist in 80x86-Assembler geschrieben, da in seinem Ablauf direkt auf Registerinhalte des Rechners Bezug genommen wird. Dies ist mit den Sprachmitteln einer höheren Programmiersprache nicht möglich.

Spätestens beim nächsten Timerinterrupt erfolgt wieder eine Verzweigung in die von `MTStart` geladene Interruptbehandlungsroutine, und es findet ein erneuter Prozeß-wechsel statt. Abbildung 11.3.3-3 zeigt in Fortsetzung zu Abbildung 11.3.3-1 die Situation nach Auswahl eines neuen zu aktivierenden Prozesses, hier Pi, unmittelbar vor Sprung an dessen Prozeßfortsetzungsadresse.

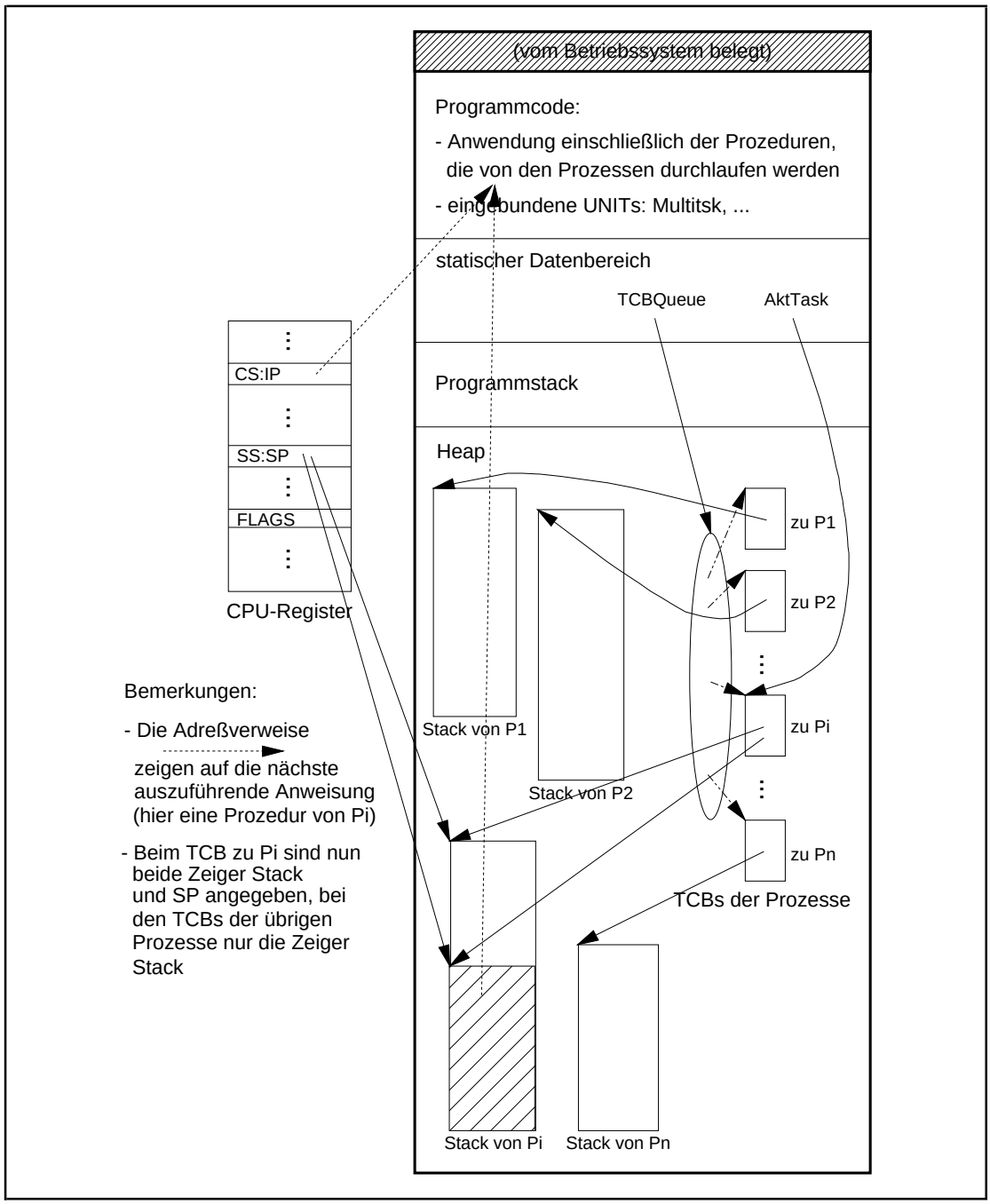


Abbildung 11.3.3-3: Laufzeitlayout nach einem Prozeßwechsel (Fortsetzung von Abbildung 11.3.3-1)

11.3.4 Alternative Schedulingstrategien

Die in `MTChooseNext` implementierte durch externe Prioritäten gesteuerte unterbrechende **Round-Robin-Schedulingstrategie**, deren Prioritätskonflikte nach der LRU-Regel gelöst werden, ist nur eine von mehreren in Betriebssystemen üblichen Strategien. Bei gleichen externen Prioritäten erhalten hierbei alle Prozesse im Zustand BEREIT den gleichen Anteil an der CPU-Zeit. Allerdings steigt mit der Zahl der durch

die CPU zu bedienenden Prozesse die Wartezeit eines jeden Prozesses, da ja die CPU-Verfügbarkeit gleichmäßig und gerecht auf alle Prozesse im Zustand BEREIT aufgeteilt wird. Bei einer großen Anzahl an Prozessen ergibt sich dabei ein eventuell unbefriedigendes Zeitverhalten einzelner Prozesse. Bei unterschiedlichen externen Prioritäten werden immer die Prozesse höchster Priorität bedient, so daß ein Prozeß niedrigerer Priorität nur dann die CPU erhält, wenn alle Prozesse höherer Priorität im Zustand WARTEND oder terminiert sind. Daher können Prozesse hoher Priorität Prozesse niedrigerer Priorität von der CPU-Zuteilung ausschließen.

Alternative Schedulingstrategien lassen sich leicht in die Unit `Multitsk` einbauen. Dazu muß lediglich die Prozedur `MTChooseNext` durch eine andere, die neue Strategie implementierende Prozedur ausgetauscht werden. Im folgenden werden einige in Betriebssystemen gebräuchliche Strategien informell beschrieben. Dabei wird davon ausgegangen, daß nicht alle Prozesse zur gleichen Zeit in das System eintreten, sondern eventuell auch erst nach Start der Multitasking-Kontrolle eingerichtet werden. Durch Hinzufügen eines weiteren Formalparameter, der den Zeitpunkt des Prozeßstarts relativ zum Start von `MTStart` bestimmt, in das Aufrufformat der Prozedur `MTChooseNext` kann dieses Verhalten auch simuliert werden.

Bei der **Dynamic-Priority-Round-Robin-Schedulingstrategie** erhält jeder Prozeß bei Eintritt in das System die interne Priorität 0. Sie wächst linear mit der Rate $\alpha > 0$, bis die Priorität so groß ist wie die kleinste Priorität bereits im System befindlicher Prozesse. Von nun an wächst die Prozeßpriorität linear mit der Rate $\beta > 0$, wobei $\alpha \geq \beta \geq 0$ gilt. Der Prozeß höchster Priorität wird wie bei Round Robin für die Dauer von höchstens einer Zeitscheibe Δ bedient.

Bei der **Multi-Level-Feedback-Schedulingstrategie** werden n Prioritätsstufen $\pi_n, \pi_{n-1}, \ldots, \pi_2, \pi_1$ absteigender Priorität vereinbart. Für jede Prioritätsstufe π_i sei T_i die maximale CPU-Zeit, die ein Prozeß auf dieser Prioritätsstufe insgesamt bekommen kann. Jeder neu in das System ankommende oder jeder aus dem Zustand WARTEND in den Zustand BEREIT wechselnde Prozeß bekommt die Prioritätsstufe π_n. Sobald ein Prozeß in Prioritätsstufe π_i die CPU-Zeit T_i seit Ankunft auf Stufe π_i verbraucht hat, geht er zurück in den BEREIT-Zustand, und zwar in Prioritätsstufe π_{i-1}. Prozesse in Prioritätsstufe π_1 werden nicht zurückgesetzt. Prozesse werden nach nichtleeren absteigenden Prioritätsstufen und dort gemäß First-in-First-out oder Round Robin mit einer festen Zeitscheibe Δ bedient.

Grundlage der folgenden Strategie ist eine **Überwachungsfunktion)** $h(t)$, die eine Beziehung zwischen der Verweildauer eines Prozesses im System und der gewünschten Bedienzeit angibt, die ein Prozeß zu einem Zeitpunkt seit seiner Ankunft im System erhalten haben sollte. $h(t)$ ist die Zeitdauer, für die ein Prozeß, der t Zeiteinheiten im System ist, die CPU erhalten soll. Ein Prozeß heißt **kritisch**, wenn die nach t Zeiteinheiten seit Eintritt in das System tatsächlich erhaltene Prozessorzeit kleiner als $h(t)$ ist (Abbildung 11.3.4-1). Einem kritischen Prozeß wird die CPU für die Dauer einer Zeitscheibe

Δ zugeteilt. In realen Betriebssystemen, z.B. SNI BS2000, werden häufig komplexe Überwachungsfunktionen eingesetzt, in die die gegenwärtige Betriebsmittelbelastung, unterschiedliche Prozeßtypen, externe Prioritäten usw. einbezogen werden. Um zu vermeiden, daß bei geringer Systemlast Prozesse unterbrochen werden, die vor ihrer Vorgabe liegen, ohne daß ein kritischer Prozeß diese Unterbrechung nötigt macht, und daß bei hoher Systemlast Prozesse, die hinter ihrer Vorgabe liegen, zu früh unterbrochen werden, empfiehlt sich die Beachtung folgender Regel: Falls ein Prozeß, der gerade eine Zeitscheibe und die CPU erhalten hat, hinter seiner Vorgabe $h(t)$ liegt oder falls ein Prozeß höchster Priorität vor seiner Vorgabe liegt, wird dem zuletzt bearbeiteten Prozeß eine weitere Zeitscheibe zugeordnet.

Die Überwachungsfunktion in Abbildung 11.3.4-1 hat die Eigenschaft, daß sie periodisch große Steigungen aufweist und innerhalb einer Periode sublinear wächst. Dadurch verhält sich ein lange im System befindlicher Prozeß immer wieder wie ein Prozeß, der gerade das System betreten hat. Erst kurz unter Multitasking-Kontrolle befindliche Prozesse werden bevorzugt, allerdings wird einem lange im System befindlichen Prozeß immer wieder die Möglichkeit gegeben, ebenfalls in den Zustand AKTIV zu gelangen (und sich dann eventuell regulär zu beenden). Die Funktion in Abbildung 11.3.4-1 berechnet sich aus der logarithmisch wachsenden Funktion

$f(t)$ = Anzahl signifikanter Bits, d.h. ohne führende 0_2-Bits, in der Binärdarstellung von t

wie folgt: Es sei $m \in \mathbf{N}$ eine vorgegebene Konstante. Für $t \in \mathbf{N}$ mit $k \cdot m \leq t < (k+1) \cdot m$ ist

$h(t) = k f(m) + f(t - km)$.

Offensichtlich ist $k = t$ DIV m.

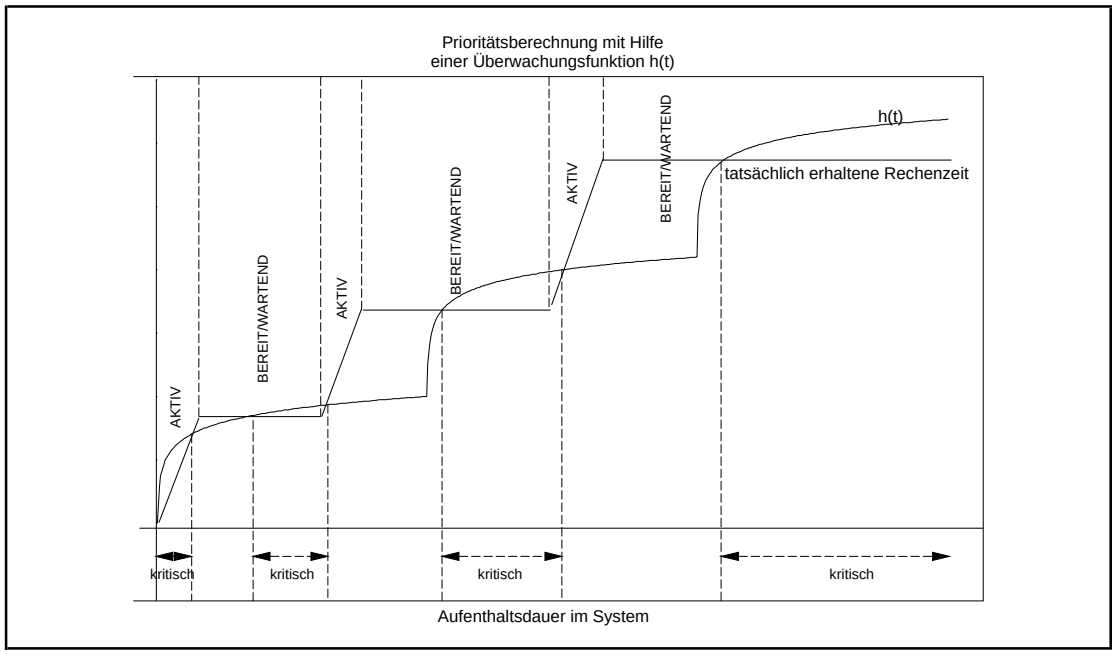


Abbildung 11.3.4-1: Überwachungsfunktion zur Prioritätsberechnung

11.4 Prozeßsynchronisation

Wenn mehrere Prozesse ein und dieselbe Ressource gleichzeitig verwenden wollen, ist eine **Prozeßsynchronisation** erforderlich. Das Prozeßkonzept mit seinen Prozeßwechselmechanismen verhindert beispielsweise, daß verschiedene Prozesse zur gleichen Zeit den (Hardware-) Registersatz besitzen (dabei wird sogar noch eine mehr oder weniger "faire" Zuweisungsstrategie beachtet). Andere knappe Resourcen, für die es Zuteilungsmechanismen geben muß, sind gemeinsame Datenobjekte, Arbeitsspeicherbereioche, Peripheriespeicher oder Geräte.

Im folgenden Beispiel soll es möglich sein, daß zwei Prozesse den Inhalt eines gemeinsamen Datenobjekts mit Bezeichner *a* lesen und ändern. Dabei ist zunächst notwendig, das Datenobjekt vor "gleichzeitigem" Zugriff und "gleichzeitiger" Änderung zu schützen. Auch ist das Ergebnis der Berechnungen ohne zusätzliche Synchronisation von der Reihenfolge abhängig, in der die Ablaufsteuerung den Prozessen die CPU zuweist.

Prozeß P_1	Prozeß P_2
`{ Initialisierung }` `a := 10;`	

`...`
`a := a * a { 1.1 };`
`Write (a) { 1.2 };`
`...`
	`...`
	`a := a / 2 { 2.1 };`
	`Write (a) { 2.2 };`
	`...`

zeitliche Reihenfolge der ausgeführten Anweisungen (verursacht durch unterschiedliche Schedulingstrategien)	Ergebnis in P_1	Ergebnis in P_2
`1.1 - 1.2 - 2.1 - 2.2`	100	50
`1.1 - 2.1 - 1.2 - 2.2`	50	50
`1.1 - 2.1 - 2.2 - 1.2`	50	50
`2.1 - 2.2 - 1.1 - 1.2`	25	5
`2.1 - 1.1 - 2.2 - 1.2`	25	25

Das Beispiel legt folgende Definition nahe: Ein **kritischer Abschnitt innerhalb (eines Programms) eines Prozesses** ist ein Programmabschnitt, in dem auf Ressourcen wie Daten (schreibend und/oder lesend) oder Geräteleistungen zugegriffen wird, die von mehreren parallel-laufenden Prozessen gemeinsam benutzt werden. Diese Art des Zugriffs nennt man **konkurrierenden Zugriff**; die Prozesse werden als **konkurrierende Prozesse** bezeichnet. Im Bespiel ist sind die Codezeilen mit Nummern 1.1 und 1.2 bzw. 2.1 und 2.2 jeweils in kritische Abschnitte der beiden Prozesse zu legen.

Die Betriebsmittel lassen sich in **gemeinsam-benutzbare (shareable)** und **nicht gemeinsam-benutzbare (non-shareable) Betriebsmittel** einteilen. Beispielsweise kann Programmcode gemeinsam-benutzbar sein. Beispiele nicht gemeinsam-benutzbarer Betriebsmitteln sind Datenspeicher (Arbeitsspeicher, Peripherie, Datenbankabschnitte usw.), auf die mehrere Prozesse lesend und schreibend zugreifen.

Das Problem der Prozeßsynchronisation tritt selbstverständlich auch auf, wenn die einzelnen Prozesse in unterschiedlichen Rechnern (Rechnernetze) bzw. CPUs (Multiprozessorsysteme) laufen.

Als generelle Regel gilt:

Nicht gemeinsam-benutzbare Betriebsmittel dürfen von konkurrierenden Prozessen nur in kritischen Abschnitten belegt werden.

Die Koordination konkurriender Prozesse ist Aufgabe der **Prozeßsynchronisation** des Betriebsystems, das hierzu Hilfsmittel zur Verfügung stellt (vgl. z.B. [TAN]).

Wesentliche **Anforderungen an die Prozeßsynchronisation** sind:

(1) Konkurrierende Prozesse mit kritischen Abschnitten, in denen sie auf gemeinsame Ressourcen zugreifen wollen, müssen daran gehindert werden, die kritischen Abschnitte gleichzeitig (evtl. quasi-simultan) zu durchlaufen (**gegenseitiger Ausschluß, mutual exclusion**).

(2) Die Synchronisationsmechanismen dürfen nicht auf Annahmen über die Ausführungsgeschwindigkeiten der einzelnen Prozesse beruhen.

(3) Ein Prozeß, der sich außerhalb eines kritischen Bereichs aufhält oder stoppt, darf andere Prozesse nicht daran hindern, ihren kritischen Bereich zu betreten.

(4) Prozesse dürfen nicht unbegrenzt auf Betriebsmittel zu warten gezwungen werden, indem ein Prozeß seinen kritischen Bereich wiederholt betritt, während andere Prozesse zum Eintritt in ihren kritischen Bereich keine Chance bekommen (**Problem des "Verhungerns", starvation**).

(5) Falls mehrere Prozesse jeweils ihren kritischen Bereich betreten wollen, darf die Entscheidung darüber, welcher von ihnen den Vorrang bekommt, nicht unendlich lange verzögert werden.

Eine Prozeßsynchronisation der einzelnen Prozesse ohne Zuhilfenahme von Betriebsystemdiensten, also "in Eigenregie", ist technisch möglich, aber nicht empfehlenswert, da dabei u.U. Randbedingungen für den korrekten Ablauf der Prozeßsynchronisation nicht eingehalten werden. Die meisten (Multitasking-) Betriebssysteme stellen als Hilfsmittel Ereignisvariablen, Semaphore und Interprozeßkommunikation bereit. Auch wenn diese

Hilfsmittel **prinzipiell funktional äquivalent** sind, d.h. eine Synchronisationsmethode läßt sich mit den Mitteln einer anderen Synchronisationsmethode nachbilden, so erscheint gelegentlich aus Sicht einer Anwendung das eine oder andere Synchronisationshilfsmittel praktikabler zu sein. Einige der Techniken werden im folgenden in das beschriebene Prozeßmodell integriert.

Eine typische Synchronisationsaufgabe zeigt der Code in Abbildung 11.4-1. Zunächst sind noch keine Synchronisationshilfsmittel zur Gewährleistung der Ausschlußbedingung eingebaut (einige entsprechenden Stellen sind bereits durch einen Pfeil markiert). Die gezeigte Anwendung enthält t zyklisch ablaufende parallele Prozesse, die jeweils kritische und unkritische Abschnitte besitzen. Der kritische Abschnitt in Prozeß i ($1 \leq i \leq t$) umfaßt mehrere Anweisungen und wird mit

`kritischer_Abschnitt_`*`i`*`;`

bezeichnet. Entsprechend heißt der aus mehreren Anweisungen bestehende unkritische Abschnitt in Prozeß *i*

`unkritischer_Abschnitt_`*`i`*`;`

Der Prozeßzyklus von Prozeß *i* wird durch eine Endebedingung

`endebedingung_`*`i`*

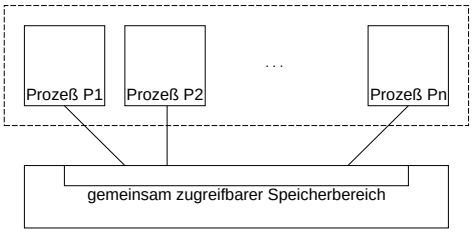
kontrolliert, die gegebenenfalls von Prozeß *i* auf TRUE gesetzt wird. Als *Ausschlußbedingung* wird gefordert, daß sich maximal $a > 0$ Prozesse gleichzeitig in ihrem kritischen Abschnitt aufhalten dürfen. Der Fall $a = 1$ bedeutet gegenseitigen Auschluß der Prozesse bezüglich ihrer kritischen Bereiche.

```
PROGRAM Syncaufgabe;

USES Multitsk;

CONST a = ...;
{ ------------------------------------------------------------- }
{$F+}
PROCEDURE prozess1;

  VAR endebedingung_1 : BOOLEAN;

  BEGIN { prozess1 }
    ...
    endebedingung_1 := FALSE;
    REPEAT
      BEGIN
        { ===> hier ist der Einbau von Synchronisations-
               hilfsmitteln erforderlich                      }
        kritischer_Abschnitt_1;

        unkritischer_Abschnitt_1;
      END
    UNTIL endebedingung_1;
    ...
  END   { prozess1 };
{$F-}
{ ------------------------------------------------------------- }
  ...
{ ------------------------------------------------------------- }
{$F+}
PROCEDURE prozesst;

  VAR endebedingung_t : BOOLEAN;

  BEGIN { prozesst }
    ...
    endebedingung_t := FALSE;
    REPEAT
      BEGIN
        { ===> hier ist der Einbau von Synchronisations-
               hilfsmitteln erforderlich                      }
        kritischer_Abschnitt_t;

        unkritischer_Abschnitt_t;
      END
    UNTIL endebedingung_t;
    ...
  END   { prozesst };
{$F-}
{ ------------------------------------------------------------- }
BEGIN { Syncaufgabe }
  { ===> Synchronisationshilfsmittel initialisieren             }
  { Prozesse einrichten: }
  MTCreateTask (prozess1, ...);
  ...
  MTCreateTask (prozesst, ...);
  { Multitasking starten; die eingerichteten Prozesse
    laufen parallel unter Kontrolle der Unit Multitsk ab: }
  MTStart (...);
  { ===> Entfernen der zuvor eingerichteten
         Synchronisationshilfsmittel                          }
END   { Syncaufgabe }.
```

Abbildung 11.4-1: Ein Beispiel zur Prozeßsynchronisation

Den Methoden konkurrierender Programmierung liegen zwei unterschiedliche **Modellvorstellungen** zugrunde, je nachdem, ob die Prozesse auf einen gemeinsamen Speicherbereich zugreifen können oder nicht (Abbildung 11.4-2). Das erste Modell eignet sich dabei gut zur Beschreibung der Prozeßsynchronisation in Multitasking-Betriebssystemen mit einer oder wenigen CPUs und gemeinsamen Arbeitsspeicher, das zweite Modell beschreibt allgemeinere Mechanismen, die auch in verteilten Systemen einsetzbar sind.

Modell 1: Konkurrierende Prozesse haben Zugriff auf einen gemeinsamen Datenbereich. In diesem Datenbereich können sie oder das Betriebssystem **gemeinsame Variablen zur Prozeßsynchronisation** anlegen.



Modell 2: Konkurrierende Prozesse haben keinen Zugriff auf einen gemeinsamen Datenbereich. Die einzige Möglichkeit zur Prozeßsynchronisation besteht über den **Austausch von Nachrichten (message passing)**. Derartige Systeme heißen auch **verteilte Systeme**. Der Nachrichtenaustausch erfolgt über ein Netzwerk oder über Dienste des Betriebssystems. Alle Methoden in verteilten Systemen sind natürlich auch in Modell 1 anwendbar, falls das Betriebssystem oder das verwendete Rechnersystem entsprechende Dienste zum Nachrichtenaustausch zur Verfügung stellt.

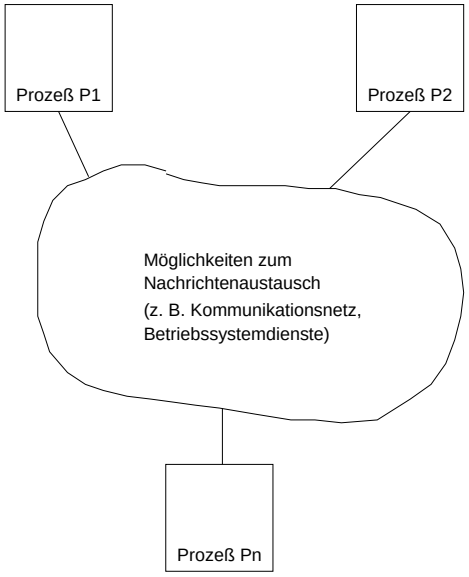


Abbildung 11.4-2: Grundmodelle der Prozeßsynchronisation

Eine Auswahl von Methoden im Modell 1 wird in den folgenden Unterkapiteln beschrieben. Dabei wird auf Methoden verzichtet, die in einigen Programmiersprachen durch ent-

sprechende Sprachkonstrukte implementiert sind (z.B. das Rendezvouskonzept). Methoden für Modell 2, deren hauptsächliches Wesensmerkmal die Interprozeßkommunikation ist, werden in Kapitel 11.5 behandelt.

Alle im folgenden behandelten Synchronisationsmechanismen verwenden Synchronisationselemente, deren Objekttypen von einem allgemeinen Objekttyp in der Unit `Basics` abgeleitet werden. Dieser allgemeine Objekttyp ist

```
{ allgemeines Synchronisationselement: }
PSyncelement = ^TSyncelement;
TSyncelement = OBJECT
                 list : PFIFO; { Warteschlange des
                                 Synchronisationselements    }
                 CONSTRUCTOR init;
                 PROCEDURE final_handling (taskid : Pointer);
                                                    VIRTUAL;
                      { typabhängige Behandlung eines
                        Eintrags bei Entfernung aus dem
                        Synchronisationselement              }
                 PROCEDURE Killtask (taskid : Pointer);
                      { Entfernung eines Prozesses aus der
                        Warteschlange des Synchronisations-
                        elements                             }
                 DESTRUCTOR done (taskid : Pointer); VIRTUAL;
               END;
```

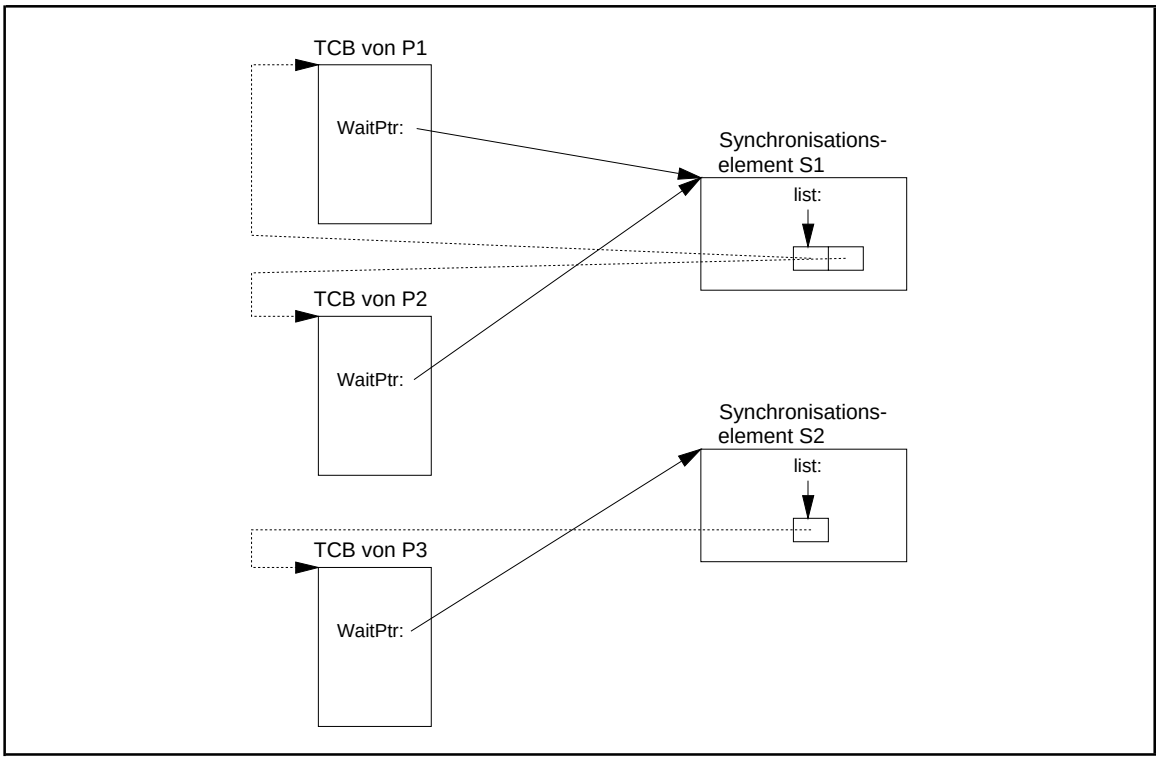


Abbildung 11.4-3: Vor Synchronisationselementen wartende Prozesse

Ein Synchronisationselement enthält also mindestens eine FIFO-Warteschlange (aus der Unit `Basics`), in der die TCBs der Prozesse eingehängt werden, die vergeblich versucht

haben, das Synchronisationselement zu passieren, und nun im Zustand WARTEND davorstehen. Im TCB eines derartigen Prozesses steht in der Komponente `WaitPtr` die Adresse des Synchronisationselements. Der Formalparameter
`taskid`
in den Methoden des Objekttyps `TSyncelement` dient der Übergabe eines Verweises auf den TCB eines Prozesses in den Methoden von Objekttypen, die von `TSyncelement` abgeleitet werden und dort eventuell einen derartigen Parameter benötigen. Abbildung 11.4-3 zeigt drei Prozesse P1, P2 und P3 und zwei Synchronisationselemente S1 und S2. Die Prozesse P1 und P2 warten vor S1, P3 wartet vor S2. Die Pfeile deuten die Adreßverkettungen zwischen den beteiligten TCBs und den FIFO-Warteschlangen der Synchronisationselemente an.

Wichtig ist, daß der aus `TSyncelement` abgeleitete Objekttyp eines spezialisierten Synchronisationselements bei Bedarf eine eigene *virtuelle* Methode namens

```
PROCEDURE final_handling (taskid : Pointer); VIRTUAL;
```

enthält, in der die Aktionen in Bezug auf das spezielle Synchronisationselement kodiert werden, die bei Entfernung eines Prozesses aus diesem Synchronisationselement ausgeführt werden sollen. Die Methode

```
TSyncelement.final_handling
```

enthält nämlich keinen Code. Eine derartige Methode wird als **abstrakte Methode** bezeichnet. Insgesamt lautet die Implementation der Methoden des Objekttyps
`TSyncelement`:

```
CONSTRUCTOR TSyncelement.init;
  BEGIN { TSyncelement.init }
    list    := New (PFIFO, init);
  END   { TSyncelement.init };


PROCEDURE TSyncelement.final_handling (taskid : Pointer);
  BEGIN { TSyncelement.final_handling }
  END   { TSyncelement.final_handling };


PROCEDURE TSyncelement.Killtask (taskid : Pointer);
  BEGIN { TSyncelement.Killtask }
    list^.delete_entry (taskid);
    final_handling (taskid);
  END   { TSyncelement.Killtask };


DESTRUCTOR TSyncelement.done (taskid : Pointer);

  BEGIN { TSyncelement.done }
    Dispose (list, done);
  END   { TSyncelement.done };
```

11.4.1 Semaphore

Das Konzept der Semaphore wurde von E.W. Dijkstra 1965 vorgeschlagen und seither in mehreren Varianten weiterentwickelt (vgl. [M/O]). Ein **Semaphor** ist ein im Betriebssystem implemetiertes Datenobjekt, das das Betriebssystem **für alle Prozesse im gemeinsam zugreifbaren Speicherbereich verwaltet**. Ein Semaphor schützt kritische Abschnitte konkurrierender Prozesse.

Im folgenden soll entsprechend der Synchronisationsaufgabe aus Abbildung 11.4-1 angenommen werden, daß sich bis zu a Prozesse gleichzeitig (parallel) in ihren kritischen Abschnitten bewegen dürfen, z.B. daß bis zu $a > 0$ Prozesse eine Ressource simultan belegen können. ***Bevor ein Prozeß seinen kritischen Abschnitt betreten möchte, "erbittet er die Erlaubnis hierzu", indem er eine mit* P *benannte Methode des den kritischen Abschnitt schützendes Semaphors anstößt.*** Wenn mehr als a Prozesse ihren kritischen Abschnitt betreten wollen, werden sie daran mit Übergang in den Prozeßzustand WARTEND gehindert, bis einer der a Prozesse, die sich gerade in ihrem kritischen Abschnitt aufhalten, diesen verlassen; dann wird ein weiterer Prozeß in seinen kritischen Abschnitt eingelassen[39]. Alle auf das Passieren eines Semaphors vergeblich wartenden Prozesse reihen sich in die zum Semaphor gehörende FIFO-Warteschlange vor dem Semaphor ein. ***Ein Prozeß gibt das Verlassen seines kritischen Abschnitts dadurch bekannt, daß er eine mit* V *benannte Methode des den kritischen Abschnitt schützenden Semaphors aufruft.***

Die hier beschriebene Semaphorvariante heißt **zählendes Semaphor**.

Semaphore können genauso in verteilten Systemen definiert werden, die über eine gemeinsame Steuerung verfügen. In verteilten Systemen, in denen eine derartige zentralisierte Kontrolle nicht existiert, sind verfeinerte Semaphormechanismen erforderlich (auf mehrere Prozesse verteilte Semaphore[40]).

---

[39] Die Situation ist vergleichbar mit einem Parkhaus, das a Stellplätze enthält und am Eingang mit einer Ampel (ital.: *il semaforo*) versehen ist, die solange auf grün steht, wie noch nicht alle Stellplätze im Parkhaus belegt sind. Sobald das Parkhaus vollständig gefüllt ist, schaltet die Ampel auf rot, und weitere ankommende Fahrzeuge müssen sich in eine (first-in-first-out-) Warteschlange vor dem Parkhaus einreihen. Jedes das Parkhaus verlassende Fahrzeug gibt die Eintrittsberechtigung für ein auf Einlaß wartendes Fahrzeug (die Ampel schaltet für ein wartendes Fahrzeug auf grün).

[40] Vgl. Andrews, G.R.: Paradigms for Process Interaction in Distributed Programs, ACM Computing Surveys 23:1, 1991, S.49-90.

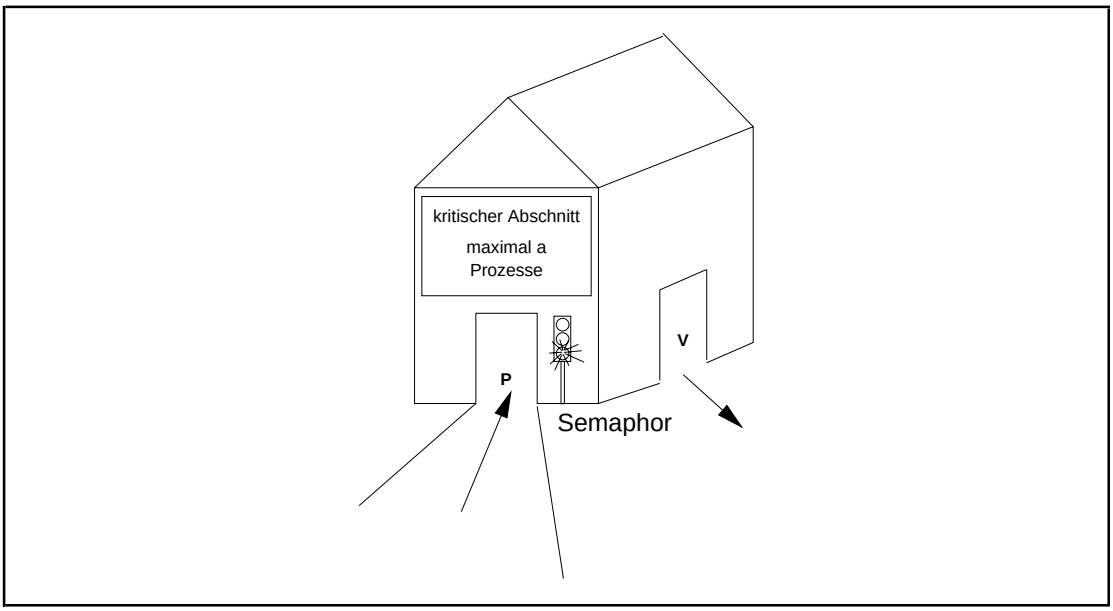


Abbildung 11.4.1-1: Semaphor

Im folgenden wird zunächst die Implementierung des "klassischen" Semaphorkonzepts vorgestellt, das dann um Lösungsvorschläge erweitert wird, die den Problemen, die sich aus einem Prozeßabbruch durch Aufruf der Prozedur `MTKillTask` ergeben, entgegenwirken können. Anschließend werden Erweiterungen des Semaphorkonzepts diskutiert.

Zur Implementierung und Integration des Konzepts zählender Semaphore in das beschriebene Prozeßmodell wird die Unit `Semaphor` definiert. Sie enthält im `INTERFACE`-Teil die Deklaration des Objekttyps `Tsemaphor`, der sich vom Objekttyp `TSyncelement` ableitet und zur Deklaration eines Semaphors verwendet wird:

```
TYPE PSemaphor = ^TSemaphor;
     Tsemaphor =
       OBJECT (TSyncelement)
          PRIVATE
            count : INTEGER;     { Semaphorzähler                    }
          PUBLIC
            CONSTRUCTOR init (a : INTEGER);
            PROCEDURE P;
            PROCEDURE V;
            PROCEDURE final_handling (taskid : Pointer); VIRTUAL;
            DESTRUCTOR close; VIRTUAL;
        END;
```

In der **Implementierung** besteht ein **Semaphor** im wesentlichen aus einem (Semaphor-) Zähler (Komponente `count`), aus dessen Wert sich zu jedem Zeitpunkt berechnen läßt, wieviele Prozesse das Semaphor passiert haben und sich gegenwärtig in ihrem kritischen Abschnitt aufhalten und wieviele Prozesse noch auf das Passieren des Semaphors warten. Dieser Zähler wird bei Initialisierung des Semaphors auf den Maximalwert *a* gesetzt. Vor Eintritt eines Prozesses in seinen kritischen Abschnitt ruft er die Methode `Tsemaphor.P` für das den kritischen Abschnitt schützende Semaphor auf. Der Semaphorzähler wird dekrementiert. Falls bei dieser Operation ein negativer Zählerwert

entsteht, wird der aufrufende Prozeß blockiert (Überführung in den Prozeßzustand WARTEND). Sobald ein Prozeß seinen kritischen Abschnitt verläßt und die Methode `Tsemaphor.V` aktiviert, wird der Zähler inkrementiert. Die Adressen der TCBs derjenigen Prozesse, die vergeblich das Semaphor passieren wollten und nun vor dem Semaphor warten, stehen in der über die von `TSyncelement` geerbten Komponente `list` adressierten FIFO-Warteschlange.

Die Korrektheit des Synchronisationsmechanismus mit Hilfe von zählenden Semaphoren ergibt sich aus folgender **Konsistenzbedingung**, die jeweils nach Beendigung eines P- und V-Methodenaufrufs gilt:

Ist *c* der gegenwärtige Wert des Semaphorzählers `count` und *a* sein Anfangswert aus dem `init`-Aufruf, so besteht der Zusammenhang

c + Anzahl der Prozesse in ihren kritischen Abschnitten

\+ Anzahl der Prozesse in der Warteschlange

$= a$.

Insbesondere gilt

$c < 0 \Leftrightarrow$ Anzahl der Prozesse in der Warteschlange > 0.

Die Konsistenzbedingung weist auf eine Problematik im Semaphorkonzept hin: Soll ein Prozeß (durch `MTKillTask`) aus dem System entfernt werden, der sich gerade in der Warteschlange eines Semaphors befindet, weil er vergeblich versucht hat, das Sempahor zu passieren, so ist bereits der Semaphorzählerwert verringert worden. Würde man lediglich diesen wartenden Prozeß aus der Semaphorwarteschlange und dem System entfernen, so ist nun die Konsistenzbedingung verletzt. Die Prozedur `MTKillTask` trägt dieser Situation Rechnung, indem sie für einen zu entfernenden, vor einem Synchronisationselement wartenden Prozeß die virtuelle Methode `KillTask` des Synchronisationselements aufruft:

```
IF Task^.WaitPtr <> NIL
THEN { der Prozeß wartet in einem Synchronisationselement }
     Task^.WaitPtr^.Killtask (Task);
```

`TSyncelement.Killtask` entfernt den Prozeß aus der Warteschlange des Synchronisationselements (hier: des Semaphors) und ruft dann die virtuelle Methode `TSyncelement.final_handling` auf (vgl. Kapitel 11.4). Bei einem Semaphor ist hierin die Inkrementierung des Semaphorzählers implementiert, so daß nach Entfernen des Prozesses die Konsistenzbedingung wieder erfüllt ist.

Die Schnittstellenbeschreibung des Objekttyps `TSemaphor` lautet:

Methode	Bedeutung
`CONSTRUCTOR Tsemaphor.init (a : INTEGER);`	Das Semaphor wird eingerichtet.
Bedeutung des Parameters: `a` Anfangswert des Semaphorzählers	
`PROCEDURE Tsemaphor.P;`	Das Semaphor wird passiert, falls nicht bereits *a* (= Wert beim `init`-Aufruf) Prozesse das Semaphor passiert und nicht wieder "freigegeben" haben. In diesem Fall wird der aufrufende Prozeß unterbrochen (Zustand WARTEND).
`PROCEDURE Tsemaphor.V;`	Das Semaphor wird "freigegeben": einem anderen Prozeß (falls es ihn überhaupt gibt) wird erlaubt, das Semaphor zu passieren.
`PROCEDURE Tsemaphor.final_handling (taskid : Pointer); VIRTUAL;`	Die Prozedur definiert die Abschlußbehandlung bei Entfernung eines vor dem Semaphor wartenden Prozesse durch `MTKillTask`. Die Prozedur darf aus einem Prozeß heraus nicht direkt aufgerufen werden.
Bedeutung des Parameters: `taskid` Zeiger auf den TCB des zu entfernenden Prozesses.	
`DESTRUCTOR TSemaphor.close;`	Das Semaphor wird aus dem System entfernt.

Der folgende Abdruck zeigt die Implementierung der Methoden des Objekttyps `Tsemaphor`. Zu beachten ist, daß aus Konsistenzgründen während der Abfragen und Änderungen der Komponenten eines Semaphors keine Prozeßwechsel zulässig sind. Daher wird der Interruptmechanismus kurzfristig und unter Kontrolle des Betriebssystems außer Kraft gesetzt.

```
CONSTRUCTOR Tsemaphor.init (a : INTEGER);

BEGIN { Tsemaphor.init }
  INHERITED init;
  count := a;
END   { Tsemphor.init };


PROCEDURE Tsemaphor.P;

BEGIN { Tsemaphor.P }
  MTBlock;
  count := count - 1;
  IF count < 0 THEN BEGIN
                      list^.insert (AktTask); { eigenen TCB in die
                                                Warteschlange eintragen       }
                      MTWait (@Self); { Übergang in den Zustand
                                        WARTEND; der Taskwechsel wird
                                        automatisch durchgeführt      }
                    END
                ELSE MTContinue;
END   { Tsemaphor.P };
```

```
PROCEDURE Tsemaphor.V;

VAR task : TaskPtr;

BEGIN { Tsemaphor.V }
  MTBlock;
  count := count + 1;
  IF count <= 0 THEN BEGIN
                       list^.delete(Pointer(task));
                       MTSignal (task);
                     END;
  MTContinue;
END   { Tsemaphor.V };


PROCEDURE Tsemaphor.final_handling (taskid : Pointer);

BEGIN { Tsemaphor.final_handling }
  count := count + 1;
END   { Tsemaphor.final_handling };


DESTRUCTOR Tsemaphor.close;

BEGIN { Tsemaphor.close }
  INHERITED done (NIL);
END   { Tsemaphor.close };
```

In der Implementierung von `Tsemaphor.P` wird der in Kapitel 9.2 erwähnte implizite Parameter `Self` einer Methode in Pascal verwendet. Die Prozedur `MTWait` erwartet als Aktualparameter einen Verweis auf das Synchronisationselement, vor dessen Passieren der Prozeß in den Zustand WARTEND gerät (vgl. Kapitel 11.2). Mit `@Self` wird genau dieser Verweis benannt.

Abbildung 11.4.1-1 zeigt eine Lösung der in Abbildung 11.4-1 formulierten Grundaufgabe des gegenseitigen Ausschlusses von Prozessen mit Hilfe eines einzigen Semaphors.

```
PROGRAM Syncaufgabe;

USES Multitsk, Semaphor;

CONST a = ...;
VAR sem : Psemaphor;
{ ---------------------------------------------------------------- }
PROCEDURE prozess1;

  VAR endebedingung_1 : BOOLEAN;

  BEGIN { prozess1 }
    ...
    endebedingung_1 := FALSE;
    REPEAT
      BEGIN
        sem^.P;
        kritischer_Abschnitt_1;
        sem^.V;
        unkritischer_Abschnitt_1;
      END
    UNTIL endebedingung_1;
    ...
  END   { prozess1 };
{ ---------------------------------------------------------------- }
...
{ ---------------------------------------------------------------- }
PROCEDURE prozesst;

  VAR endebedingung_t : BOOLEAN;

  BEGIN { prozesst }
    ...
    endebedingung_t := FALSE;
    REPEAT
      BEGIN
        sem^.P;
        kritischer_Abschnitt_t;
        sem^.V;
        unkritischer_Abschnitt_t;
      END
    UNTIL endebedingung_t;
    ...
  END   { prozesst };
{ ---------------------------------------------------------------- }

BEGIN { Syncaufgabe }

  sem := New (Psemaphor, init (a));

  { Prozesse einrichten: }
  MTCreateTask (prozess1, ...);
  ...
  MTCreateTask (prozesst, ...)

  { Multitasking starten; die eingerichteten Prozesse
    laufen parallel unter Kontrolle der Unit Multitsk ab: }
  MTStart (...);

  Dispose (sem, close);

END   { Syncaufgabe }.
```

Abbildung 11.4.1-1: Ein Beispiel zur Prozeßsynchronisation (vgl. Abbildung 11.4-1)

Das Beispiel in Abbildung 11.4.1-1 zeigt ein weiteres Problem des Semaphorkonzepts: Es sei $a = 1$, d.h. das durch `sem` adressierte Semaphor realisiert einen gegenseitigen Ausschluß bezüglich der kritischen Abschnitte. Angenommen, Prozeß 1 ruft nun `MTKillTask` auf, um Prozeß *t* aus dem System zu entfernen. Befindet sich Prozeß *t* gerade in seinem unkritischen Abschnitt, so kann er aus dem System entfernt werden. Wartet Prozeß *t* vor dem Semaphor in der Warteschlange, so kann er ebenfalls aus dem System entfernt werden; die Abschlußbehandlung in der Methode `Tsemaphor.final_handling` sichert die Konsistenzbedingung des Semaphors. Befindet Prozeß *t* sich aber gerade in seinem kritischen Abschnitt (und Prozeß 1 folglich in seinem unkritischen Abschnitt) und wird er aus dem System entfernt, so kann Prozeß 1 seinen kritischen Abschnitt nicht mehr betreten, da das durch sem adressierte Semaphor noch sperrt. Während des Ablaufs der Prozedur `MTKillTask` muß also das durch Prozeß *t* gesperrte Semaphor wieder freigegeben werden[41]. Es empfiehlt sich daher, daß sich ein Prozeß alle diejenigen Semaphore "merkt", auf die er erfolgreich eine P-Operation angewendet hat, ohne bisher die korrespondierende V-Operation abzusetzen. Der TCB eines Prozesses enthält hierzu die bisher noch nicht beschriebene Komponente

```
TD.PassedSem : PFIFO;
```

Sobald ein Prozeß ein Semaphor passiert, wird die Adresse des Semaphors in die durch `TD.PassedSem` adressierte Warteschlange eingetragen. Der Eintrag wird wieder entfernt, wenn der Prozeß eine korrespondierende V-Operation auf das Semaphor absetzt. Dadurch, daß die entsprechenden Semaphoradressen in einer Warteschlange verwaltet werden, können auch ineinandergeschachtelte (korrekt eingesetzte) P-V-Aufrufpaare berücksichtigt werden. Entsprechend werden die Implementationen von `Tsemaphor.P` und `Tsemaphor.V` abgeändert (die Änderungen sind hervorgehoben):

```
PROCEDURE Tsemaphor.P;

BEGIN { Tsemaphor.P }
  MTBlock;
  count := count - 1;
  IF count < 0 THEN BEGIN
                      list^.insert (AktTask); { eigenen TCB in die
                                                Warteschlange eintragen       }
                      MTWait (@Self); { Übergang in den Zustand
                                        WARTEND; der Taskwechsel wird
                                        automatisch durchgeführt      }
                      MTBlock;
                    END;

  { Semaphorkennung in die Liste der passierten Semaphore eintragen:}
  AktTask^.PassedSem^.insert (@Self);
  MTContinue;
END   { Tsemaphor.P };
```

[41] Wird das Semaphor eingesetzt, um z.B. eine exklusive Sperre auf eine Ressource, wie eine exklusive Schreibsperre auf einen Datenbereich, zu realisieren, so muß vor Entfernen des Prozesses diese Sperre wieder freigegeben werden, damit insgesamt im System ein Deadlock verhindert wird.

```
PROCEDURE Tsemaphor.V;

VAR task : TaskPtr;

BEGIN { Tsemaphor.V }
  MTBlock;
  count := count + 1;
  IF count <= 0 THEN BEGIN
                       list^.delete(Pointer(task));
                       MTSignal (task);
                     END;
  { Semaphorkennung aus der Liste der passierten Semaphore
    entfernen:}
  AktTask^.PassedSem^.delete_entry (@Self);
  MTContinue;
END   { Tsemaphor.V };
```

Wird nun ein Prozeß mit Hilfe der Prozedur `MTKillTask` aus dem System entfernt, der noch Einträge in seiner durch `PassedSem` adressierten Liste enthält (er hat also Semaphore mit der P-Methode passiert, aber noch nicht wieder freigegeben), so werden die Semaphorzähler dieser Semaphore gezielt aktualisiert und eventuell vor dem Semaphor wartende Prozesse wieder aktiviert. Der entsprechende Codeteil in der in `MTKillTask` internen Prozedur `MTDelTask` lautet (der Verweis `Task` zeigt auf den TCB des zu entfernenden Prozesses):

```
  PROCEDURE MTDelTask (Task : TaskPtr);

  VAR i        : INTEGER;
      port_ptr : PSyncelement;
      ptr      : Pointer;

  BEGIN { MTDelTask }
    ...
    IF Task^.WaitPtr <> NIL
    THEN { der Prozeß wartet in einem Synchronisationselement }
         Task^.WaitPtr^.Killtask (Task);
    { passierte Semaphore freigeben: }
    Task^.PassedSem^.delete (ptr);
    WHILE ptr <> NIL DO
      BEGIN
        PSemaphor(ptr)^.final_handling (NIL);
        Task^.PassedSem^.delete (ptr);
      END;
    ...
  END   { MTDelTask };
```

Zur Typisierung der Variablen `ptr` mit dem Datentyp `PSemaphor` muß in den `IMPLEMENTATION`-Teil der Unit `Multitsk` die Anweisung

```
USES Semaphor;
```

eingebaut werden (zur Vermeidung zyklischer `USES`-Referenzen wird der `IMPLEMENTATION`- und nicht der `INTERFACE`-Teil genommen). Man sieht, daß hier die Methode `Tsemaphor.final_handling` direkt, also nicht über `Tsemaphor.KillTask`, aufgerufen wird. In der Methode `Tsemaphor.final_handling` wird dieser Aufrufweg am Aktualparameter `NIL` erkannt. Auch diese Methode wird also angepaßt:

```
PROCEDURE Tsemaphor.final_handling (taskid : Pointer);

VAR task : TaskPtr;

BEGIN { Tsemaphor.final_handling }
  MTBlock;
  count := count + 1;
  IF (taskid = NIL) AND (count <= 0)
  THEN BEGIN
         list^.delete(Pointer(task));
         MTSignal (task);
       END;
  MTContinue;
END   { Tsemaphor.final_handling };
```

Die so abgeänderten Semaphor-Methoden erhalten die Konsistenzbedingung eines Semaphors im Falle des Entfernens von Prozessen in Situationen, in denen ein Prozeß zu jedem Aufruf der P-Methode auch selbst einen korrespondierenden Aufruf der V-Methode absetzt bzw. absetzen wird. Die durch seine TCB-Komponente `PassedSem` adressierte Liste wird dann immer wieder durch den Aufruf der V-Methode bereinigt. Für das Problem des gegenseitigen Ausschlusses, wie es in Abbildung 11.4.1-1 beschrieben wird, trifft dieses zu. Es gibt jedoch andersartige Anwendungen, in denen die korrespondierenden P-V-Aufrufpaare auf mehrere Prozesse verteilt sind. Beispielsweise führt der Produzentprozeß im unten beschriebenen Produzent/Konsument-Problem ausschließlich die P-Aufrufe auf ein Semaphor aus, während die zugehörigen V-Aufrufe vom Konsumentprozeß kommen. Ein Verbindung zwischen Produzent- und Konsumentprozeß besteht hier nicht. In dieser Situation wird die durch PassedSem adressierte Warteschlange im Produzentprozeß aufgrund der von ihm durchgeführten P-Aufrufe immer länger, ohne daß sie durch die korrespondierenden V-Aufrufe, die ja in einem anderen Prozeß liegen, wieder gekürzt werden könnte. Hier ist es angebracht, doch bei der ursprünglichen Implementation der P-Methode zu bleiben (mit dem Wissen, daß das Entfernen von Prozessen problematisch sein kann). Die Menge der Methoden eines Semaphors wird daher um eine weitere Methode mit Bezeichner `PnoControl` erweitert. Sie beinhaltet genau die ursprüngliche P-Methode, d.h. es werden keine Semaphoraufrufe in der durch `PassedSem` adressierten Liste protokolliert, und ist mit der oben beschriebenen abgeänderten V-Methode kompatibel.

Methode	Bedeutung
...	
`PROCEDURE Tsemaphor.PnoControl;`	wie die Methode `P`, nur wird der Semaphoraufruf beim Prozeß nicht registriert
...	

Die Objekttypdeklaration eines Semaphors lautet nun:

```
TYPE Psemaphor = ^Tsemaphor;
     Tsemaphor =
       OBJECT (TSyncelement)
          PRIVATE
            count : INTEGER;     { Semaphorzähler                    }
```

```
        PUBLIC
          CONSTRUCTOR init (a : INTEGER);
                    { das Semaphor wird eingerichtet und sein
                      Zähler mit a initialisiert                }
          ...
          PROCEDURE PnoControl;
                    { wie die Methode P, nur wird der
                      Semaphoraufruf beim Prozeß nicht registriert}
          ...
      END;
```

Die Methode `PnoControl` wird implementiert durch

```
PROCEDURE Tsemaphor.PnoControl;

BEGIN { Tsemaphor.PnoControl }
  MTBlock;
  count := count - 1;
  IF count < 0 THEN BEGIN
                    list^.insert (AktTask); { eigenen TCB in die
                                              Warteschlange eintragen       }
                    MTWait (@Self); { Übergang in den Zustand
                                      WARTEND; der Taskwechsel wird
                                      automatisch durchgeführt      }
                  END
               ELSE MTContinue;
END   { Tsemaphor.PnoControl };
```

Die Betriebssystem-Literatur beschreibt eine Reihe **klassischer Synchronisationsprobleme**, die mit dem Einsatz von Semaphoren gelöst werden (vgl. [TAN], [H/H]). Einige Beispiele sind:

- das **Produzent/Konsument-Problem**: Ein Prozeß (**Produzent**) erzeugt sequentiell Sätze. Sobald ein Satz produziert worden ist, wird er in einen Puffer eingetragen, der ***maximal n Sätze*** aufnehmen kann und ringförmig und nach dem FIFO-Prinzip organisiert ist. Neue Einträge werden dabei fortlaufend im Puffer abgelegt, bis sein Ende erreicht ist. Dann werden die folgenden Einträge wieder am Pufferanfang eingefügt. Ein weiterer Prozeß (**Konsument**) entfernt sequentiell Sätze aus dem Puffer, um sie weiter zu verarbeiten (auszudrucken o.ä.). Beide Prozesse arbeiten zyklisch und parallel. Dabei ergibt sich ein Synchronisationsproblem mit folgenden Randbedingungen:
  - -- Beide Prozesse werden unabhängig voneinander gestartet; es ist nicht festgelegt, welcher Prozeß zuerst losläuft.
  - -- Über die Ausführungsgeschwindigkeit beider Prozesse wird nichts vorausgesetzt. Sie können unterschiedlich schnell laufen, und das Produzieren bzw. Konsumieren verschiedener Sätze kann zeitlich variieren.
  - -- Ein Pufferüberlauf muß verhindert werden, d.h. der Produzent darf höchstens *n* Sätze schreiben, die der Konsument noch nicht verarbeitet hat.
  - -- Ein Pufferunterlauf muß verhindert werden, d.h. der Konsument darf nicht aus einem leeren Puffer lesen wollen.

  Eine Variante des Produzent/Konsument-Problems ergibt sich bei einem *unbeschränkten Puffer* in einem **Client/Serversystem**: Als Klienten bezeichnete Prozesse geben Aufträge an einen Server, der diese Aufträge abwickelt. Den Klientenaufträgen entsprechen die produzierten Sätze im Modell. Der Produzent repräsentiert die Menge aller Klienten, der Konsument die Rolle des **Servers**. Der Server wartet auf das Eintreffen eines Auftrags und wickelt ihn ab, sobald er ankommt, und benötigt dazu eine

vom Auftrag abhängige Zeitspanne. Kommen während der Auftragsabwicklung weitere Aufträge an, so werden sie in den Puffer eingereiht. Nachdem alle derzeit anstehenden Aufträge bearbeitet sind und der Puffer leer ist, wartet der Server auf neue Aufträge.

- das **Leser/Schreiber-Problem**: Es gibt eine Anzahl zyklisch ablaufender Prozesse, die sich jeweils pro Zyklus in einem unkritischen Abschnitt aufhalten und in einem kritischen Abschnitt auf einen Datenbereich zugreifen. Dabei sind die Prozesse in zwei Klassen eingeteilt:

  Jeder Prozeß der ersten Klasse (**Leser**) darf sich den Zugriff auf den Datenbereich mit einer beliebigen Anzahl anderer Prozesse dieser Klasse teilen. Jeder Prozeß der zweiten Klasse (**Schreiber**) benötigt exklusiven Zugriff auf den Datenbereich.

  Dabei gelte zusätzlich eine der beiden folgenden **Vorrangregeln**:
  (i) Kein Leser soll beim Betreten seines kritischen Abschnitts warten müssen, wenn nicht bereits ein Schreiber seinen kritischen Abschnitt betreten hat. Ein neu hinzugekommener Leser kann einen Schreiber überholen, der darauf wartet, daß andere Leser ihren kritischen Abschnitt freigeben. Die Leser werden also bevorzugt.
  (ii) Die Schreiber sollen so schnell wie möglich Zugang zu ihren kritischen Abschnitten erhalten. Alle Leser, die im Moment, in dem der Schreiber seinen Anspruch auf Zugriff auf den Datenbereich anmeldet, dürfen ihre Arbeit in ihrem kritischen Abschnitt beenden. Nach dem Schreiber an ihren kritischen Abschnitten ankommende Leser sollen am Eintritt gehindert werden, solange der Schreiber nicht mit seinem kritischen Abschnitt fertig ist. Die Schreiber werden also bevorzugt.

  Ein praktisches Beispiel für das Leser/Schreiber-Problem ist ein Platzbuchungssystem, in dem sowohl reine Abfragen als auch Änderungen eines Datenbestands (Datenbank) vorkommen. Im Gegensatz zum Produzent/Konsument-Problem ist das Schreiber/Leser-Problem asymmetrisch.

- das **Philosophenproblem**: Eine Anzahl $n \geq 5$ Philosophen sitzen an einem runden Tisch. Sie haben nichts anderes zu tun, als zu denken und zu essen. Wenn ein Philosoph keine Lust mehr zum Denken oder einfach Hunger hat, fängt er an zu essen. Hat er seinen Hunger gestillt, so hat sein Gehirn neue Nahrung, und er fängt wieder an zu denken.

  Um den Nahrungsbedarf der Philosophen zu stillen, steht auf dem Tisch eine große Schüssel mit Reis, aus der ein Philosoph seine eigene Schale (Teller) füllt. Jeder Philosoph besitzt nur ein Eßstäbchen, obwohl er zum Essen eigentlich zwei Stäbchen benötigt[42]. Daher werden die Stäbchen zwischen die Schalen gelegt, aus denen die Philosophen ihren Reis essen, und jeder Philosoph darf außer seinem eigenen Stäbchen (rechts von seinem Schälchen) auch das seines Nachbarn (links von seinem Schälchen verwenden[43].

[42] Nur etwa 45% aller Menschen essen mit Messer und Gabel.

[43] Man sieht: Die Philosophen haben weder große kulinarische Ansprüche (sie essen nur Reis), noch sind sie sehr reich (jeder besitzt nur ein Stäbchen), noch haben sie hygienische Bedenken (sie teilen sich das Eßgeschirr mit ihrem jeweiligen Nachbarn).

Ein praktisches Beispiel für das Philosophenproblem ist eine Datenübertragung zwischen Rechnern, die durch Leitungen zu einem Ring verbunden sind. Um Daten von einem Rechner des Rings zum übernächsten weiterzugeben, müssen dem dazwischenliegenden Rechner Leitungen auf beiden Seiten exklusiv bereitstehen.

Hierbei gibt es einige Synchronisationsprobleme (in der Terminologie der Philosophen):

-- Ein Philosoph muß warten, bis zwei ihm benachbarte Stäbchen auf dem Tisch liegen (bis er also über zwei "ihm benachbarte" Betriebsmittel exklusiv verfügen kann)
-- Zwei Philosophen dürfen nicht gleichzeitig zum selben Stäbchen greifen (es besteht gegenseitiger Ausschluß bezüglich der Betriebsmittel)
-- Nicht jeder Philosoph darf sein eigenes, d.h. rechtes Stäbchen aufnehmen und dann auf das seines Nachbarn warten ("Deadlock")
-- Es sind Gerechtigkeitsüberlegungen anzustellen, damit alle Philosophen gleich satt werden und keiner verhungert.

Exemplarisch werden eine Realisierung des Produzent/Konsument-Problems und des Philosophenproblems in der beschriebenen Multitasking-Umgebung erläutert. Das Leser/Schreiberproblem ist als Übung vorgesehen (vgl. auch Kapitel 12.2).

Zunächst zum Produzent/Konsument-Problem! Der Datenaustausch zwischen Produzent und Konsument erfolgt über einen **beschränkten Puffer**, ein Datenobjekt vom Typ TPuffer, wie es in Kapitel 10.1.4 beschrieben wird. Der dort verwendete Datentyp Pentry wird jedoch durch den allgemeineren Datentyp Pointer ersetzt, so daß sich die im `INTERFACE`-Teil der Unit `Puffer` stehende `USES`-Anweisung erübrigt.

Das Programm in Abbildung 11.4.1-3 zeigt eine **Implementierung des Produzent/Konsument-Problems**. Der Produzentprozeß, dessen TCB durch die Variable `produzent_task` adressiert wird, läuft durch die Prozedur `produzent`. Der Konsumentprozeß, identifizierbar über seine TCB-Adresse in der Variablen `konsument_task`, läuft durch die Prozedur `konsument`. Der zyklische Puffer wird über die Pointervariable `austauschbereich` adressiert. Außerdem werden drei Semaphore verwendet, deren Adressen in `ausschluss`, `leer` und `belegt` stehen.

Das erste Semaphor gewährleistet gegenseitigen Ausschluß beim Zugriff auf den Puffer, die beiden anderen Semaphoren verhindern Über- und Unterlauf des Puffers. Das durch leer adressierte Semaphor kann passiert werden, wenn es noch freie Plätze im Puffer gibt, d.h. es kontrolliert die Anzahl freier Plätze im Puffer. Symmetrisch dazu kontrolliert das durch belegt adressierte Semaphor die Anzahl belegter Plätze im Puffer. Zu beachten ist, daß für diese beiden Semaphore anstelle der P-Methode die Methode `PnoControl` gewählt wurde. Nach der Produktion von jeweils 10 Sätzen wird nach einer Bestätigung der Prozeßfortsetzung gefragt; nach der Produktion von maximal 100 Sätzen wird der Produzentprozeß beendet, nachdem er den Konsumentprozeß abgebrochen hat. Die Veränderung der Prozeßpriorität im Produzentprozeß vor dem Lesen der Fortsetzungsbestätigung auf den Wert der Priorität des Konsumentprozesses bewirkt, daß der Konsumentprozeß weiterläuft, auch wenn der Produzent noch auf die Antwort des Anwenders wartet. Die Produktionsdauer eines Satzes wird durch das Pascal-Statement

```
Delay (100);
```

im Produzentprozeß simuliert. Ein entsprechendes Pascal-Statement bildet die Bearbeitungsdauer eines Satzes im Konsumentprozeß nach.

```
PROGRAM prodkom;

USES Crt,
     Multitsk, MultiWin, Semaphor, Puffer;

CONST n = 4;
      produzent_prio = 100;
      konsument_prio = 20;

TYPE Peintrag = ^STRING;

VAR produzent_task   : TaskPtr;
    konsument_task   : TaskPtr;

    ausschluss       : PSemaphor;
    leer             : PSemaphor;
    belegt           : PSemaphor;

    austauschbereich : PPuffer;
    init_OK          : BOOLEAN;


{$F+}
PROCEDURE produzent (zeiger : Pointer);

  VAR idx       : INTEGER;
      WinHandle : INTEGER;
      abbruch   : BOOLEAN;
      eintrag   : Peintrag;
      antwort   : CHAR;
      txt       : STRING;

  BEGIN { produzent }
    WinHandle := TaskWinOpen( 30, 0, 79, 11, einfacher_Rahmen,
                              ' Produzent' );
    abbruch := FALSE;
    idx     := 1;

    WHILE (NOT abbruch) AND (idx <= 100) DO
      BEGIN
        { Neuen Satz produzieren: }
        New (eintrag);
        Str (idx, eintrag^);
        eintrag^ := eintrag^ + '. Satz';
        Writeln (' ---> ' + eintrag^ + ' produzieren');
        Delay (100);
```

```
        leer^.PnoControl;
        ausschluss^.P;
        austauschbereich^.insert (eintrag);
        Writeln ('      Satz Nr. ', idx, ' eingetragen');
        ausschluss^.V;
        belegt^.V;

        IF (idx MOD 10) = 0
        THEN BEGIN
               Writeln ('Weiter? (J/N)');
               TaskWinSetCursor;
               MTBlock;
               MTChangePrio (AktTask, konsument_prio);
               MTContinue;
               Readln (antwort);
               MTBlock;
               MTChangePrio (AktTask, produzent_prio);
               MTContinue;
               IF UpCase (antwort) <> 'J' THEN abbruch := TRUE;
             END;
        Inc(idx);
      END;

    MTKillTask (konsument_task);
    Writeln ('1 : E N D E');
  END   { produzent };
{$F-}

{$F+}
PROCEDURE konsument (zeiger : Pointer);

  VAR idx       : INTEGER;
      WinHandle : INTEGER;
      eintrag   : Peintrag;

  BEGIN { konsument }
    WinHandle := TaskWinOpen( 30, 12, 79, 24, doppelter_Rahmen,
                              ' Konsument' );

    WHILE TRUE DO
      BEGIN
        belegt^.PnoControl;
        ausschluss^.P;
        austauschbereich^.delete (Pointer(eintrag));
        ausschluss^.V;
        leer^.V;

        { Satz konsumieren: }
        Writeln ('Gelesen: ', eintrag^);
        Dispose (eintrag);
        Delay (1000);
      END;

  END   { konsument };
{$F-}
```

```
BEGIN { Prodkom }
  TaskWinClrScr;

  austauschbereich := New (PPuffer, init (n, init_OK));
  IF NOT init_OK
  THEN Writeln ('Fehler bei der Initialisierung des Puffers')
  ELSE BEGIN
         ausschluss := New (PSemaphor, init (1));
         leer       := New (PSemaphor, init (n));
         belegt     := New (PSemaphor, init (0));

         produzent_task :=
             MTCreateTask (produzent, produzent_prio,  30000, NIL);
         konsument_task :=
             MTCreateTask (konsument, konsument_prio, 30000, NIL);

         MTStart (1);

         Dispose (ausschluss, close);
         Dispose (leer, close);
         Dispose (belegt, close);
         Dispose (austauschbereich, done);
       END;

END   { Prodkom }.
```

Abbildung 11.4.1-3: Produzent/Konsument-Problem

Die im folgenden beschriebene **Implementierung des Philosophenproblems** orientiert sich an der klassischen Lösung. Die $n \geq 3$ Philosophen werden mit Hilfe von n Semaphoren (die Philosophen werden von 0 bis n-1 durchnumeriert) synchronisiert, die man sich als zwischen den im Kreis plazierten Philosophen angeordnet denken kann. Die Semaphoren werden durch Adressen angesprochen, die in Form eines Felds
`privat : ARRAY[0..n-1] OF PSemaphor`
implementiert sind. Zusätzlich wird ein Feld
`c : ARRAY[0..n-1] OF philosophische_Aktion`
definiert. Hierbei ist
`TYPE philosophische_Aktion = (denken, hungrig_sein, essen);`
Die Einträge im Feld c geben an, welche Aktion ein Philosoph gerade ausführt, insbesondere, ob den Wunsch hat, seine Nahrungsaufnahme zu beginnen; in diesem Fall bekommt der entsprechende Eintrag in c den Wert hungrig_sein. Eine Prozedur test untersucht diesen Wunsch der Nachbarphilosophen und gibt ihnen eventuell statt. Dabei wird von der Regel Gebrauch gemacht, daß ein satter und denkender Philosoph seine beiden Eßstäbchen freigegeben, also zur Zeit kein Eßstäbchen aufgenommen hat; ein speisender Philosoph besitzt zwei Eßstäbchen; ein hungriger Philosoph nimmt eventuell bereits ein Eßstäbchen auf.

Zusätzlich zu den Philosophen-Tasks gibt es eine Task, adressiert durch
`check_task`,
die zyklisch prüft, ob der Gesamtablauf abgebrochen werden soll; die Philosophentasks laufen jeweils durch dieselbe nichtterminierende zyklische Prozedur, so daß sie explizit mit `MTKill-Task` beendet werden müssen. Weitere Details sind dem Code in Abbildung Abbildung 11.4.1-4 zu entnehmen; Codeteile, die besondere Beachtung verdienen, sind fett gedruckt. Der Code ist für die Simulation von n Philosophen vorgesehen, wobei die ersten fünf Philosophen (mit Nummern 0 bis 4) jeweils ein eigenes Bildschirmfenster besitzen, während sich alle weiteren Philosophen, falls es sie gibt, ein Bildschirmfenster teilen.

```
PROGRAM philosophen;

USES Crt,
     Multitsk, MultiWin, Semaphor;

CONST n          = 5;        { Anzahl der Philosophen          }
      zeitfaktor = 5;        { Zeitverzögerung beim Essen und
                               Denken (Simulation)             }
      task_prio  = 1;
      txt_lng    = 12;

TYPE index_typ             = 0..n-1;
     philosophische_Aktion = (denken, hungrig_sein, essen);
     koordinaten           = RECORD
                               x : BYTE;
                               y : BYTE;
                             END;

CONST  Rahmen : ARRAY [0..5] OF koordinaten
                 = (         { Philosophen-"Fenster";
                               für die ersten 5 Philosophen:  }
                     (x : 32; y :  2),
                     (x : 53; y :  9),
                     (x : 48; y : 17),
                     (x : 17; y : 17),
                     (x : 11; y :  9),
                     (x : 11; y :  2)
                    );
       aktionstext :
            ARRAY[philosophische_Aktion] OF STRING[txt_lng]
            = ('  D E N K T ',
               'ist hungrig ',
               '  I S S T   ');
       textattribut :
            ARRAY[philosophische_Aktion] OF Byte
            = (White, White, Yellow + Blink);
       backgroundattribut :
            ARRAY[philosophische_Aktion] OF Byte
            = (Black, Black, Red);

VAR c          : ARRAY [index_typ] OF philosophische_Aktion;
    privat     : ARRAY [index_typ] OF Psemaphor;
    ausschluss : Psemaphor;  { Zugriffskontrolle auf c und privat }
    phil_task  : ARRAY [index_typ] OF TaskPtr;
                             { Philosophen                       }
    check_task : TaskPtr;    { Abbruchkriterium                  }
    idx        : index_typ;

{ ------------------------------------------------------------- }

FUNCTION myMOD (i : INTEGER; n : INTEGER): INTEGER;

BEGIN { myMOD }
  WHILE i < 0 DO i := i + n;
  myMOD := i MOD n;
END   { myMOD };
```

```
PROCEDURE test (i : INTEGER);

BEGIN { test }
  IF (c[i] = hungrig_sein)
     AND (c[myMOD(i+1, n)] <> essen)
     AND (c[myMOD(i-1, n)] <> essen)
  THEN BEGIN
         c[i] := essen;
         privat[i]^.V
       END;
END   { test };
{ ------------------------------------------------------------- }
{$F+}
PROCEDURE philosoph (zeiger : Pointer);

  VAR nr      : INTEGER;
      txt     : STRING;
      Fenster : INTEGER;     { Fensterhandle des Philosophen  }
      aktion  : STRING[txt_lng];

  PROCEDURE taetigkeit (art : philosophische_Aktion);

    VAR txt   : STRING;
        dauer : INTEGER;

  BEGIN { taetigkeit }
    IF nr <= 4 THEN txt := ' '
               ELSE Str (nr, txt);
    IF nr > 5 THEN TaskWinGotoXY (12, 3);
    CASE art OF
    denken       : dauer := (Random (12) + 6) * 20 * zeitfaktor;
    hungrig_sein : dauer := 0;
    essen        : dauer := (Random (6) + 4) * 50 * zeitfaktor;
    END;
    TaskWinWriteStr (txt + ' ' + aktionstext[art],
                     textattribut[art], backgroundattribut[art]);
    Delay (dauer);
  END   { taetigkeit };

BEGIN { philosoph }
  nr := INTEGER(zeiger);     { Philosophennummer ermitteln      }
  Str (nr, txt);
  IF nr <= 4 THEN txt := 'Philosoph ' + txt
             ELSE txt := 'weitere Philosophen';
  { Fenster des Philosophen aufbauen
    (eigenes Fenster bis zum 5. Philosophen, für alle weiteren
     Philosophen ein gemeinsames Fenster):                      }
  IF nr <= 5
  THEN Fenster := TaskWinOpen (Rahmen[nr].x, Rahmen[nr].y,
                               Rahmen[nr].x + 15, Rahmen[nr].y + 4,
                               doppelter_Rahmen, txt);

  WHILE TRUE DO
    BEGIN
      taetigkeit (denken);

      taetigkeit (hungrig_sein);

      ausschluss^.P;
      c[nr] := hungrig_sein;
                          { Belegen eines Betriebsmittels         }
      test (nr);          { Belegen des zweiten Betriebsmittels,
                            falls es verfügbar ist                }
      ausschluss^.V;
```

```
      privat[nr]^.P;    { Warten auf das zweite Betriebsmittel,
                          falls es nicht bereits verfügbar ist  }
      taetigkeit (essen);

      ausschluss^.P;
      c[nr] := denken;  { Betriebsmittel freigeben              }
      { Nachbarprozesse anstoßen, wenn sie auf Betriebsmittel
        warten:                                                 }
      test (myMOD (nr+1, n));
      test (myMOD (nr-1, n));
      ausschluss^.V;
    END;
END   { philosoph };
{$F-}
{ ------------------------------------------------------------- }
{$F+}
PROCEDURE abbruch (zeiger : Pointer);

VAR fenster : INTEGER;
    antwort : CHAR;
    flag    : BOOLEAN;
    idx     : index_typ;

BEGIN { abbruch }
  fenster := TaskWinOpen (1, 22, 21, 24,
                          einfacher_Rahmen, 'Abbruch? (j/n)');
  flag := FALSE;
  WHILE NOT flag DO
    BEGIN
      TaskWinSetCursor;
      Readln (antwort);
      IF UpCase (antwort) = 'J' THEN flag := TRUE;
    END;
  FOR idx := 0 TO n-1 DO
    IF phil_task[idx] <> NIL THEN MTKillTask (phil_task[idx]);
END   { abbruch };
{$F-}
{ ------------------------------------------------------------- }

BEGIN { philosophen }
  Randomize;
  TaskWinClrScr;
  TaskWinHideCursor;
  FOR idx := 0 TO n-1 DO phil_task[idx] := NIL;

  ausschluss := New (PSemaphor, init (1));
  FOR idx := 0 TO n-1 DO
    BEGIN
      privat[idx]    := New (PSemaphor, init(0));
      c[idx]         := denken;
      phil_task[idx] := MTCreateTask
                          (philosoph, task_prio,
                           2000, Ptr (0, idx));
    END;
  check_task := MTCreateTask (abbruch, task_prio, 2000, NIL);

  MTStart (50);

  FOR idx := 0 TO n-1 DO Dispose (privat[idx], close);
  Dispose (ausschluss, close);

  TaskWinClrScr;
END   { philosophen }.
```

Abbildung 11.4.1-4: Philosophenproblem

Das bisher beschriebene Semaphorkonzept ist in Betriebssystemen in unterschiedlichen Varianten implementiert, z.B. als

- **Additives Semaphor**: Der Semaphorzähler kann mit einem Aufruf der jeweiligen P- oder V-Methode um mehr als 1 verändert werden

- **Semaphor mit Wartezeitbegrenzung**: Falls der Prozeß länger als eine vom Prozeß vorgegebene Zeitspanne (Timeout) auf das Passieren des Semaphors warten muß, wird er mit einer entsprechenden Rückmeldung fortgesetzt

- **Semaphor mit Alternative**: Falls der Prozeß bei Aufruf der P-Methode das Semaphor blockiert vorfindet, wird eine vom Prozeß definierte Prozedur durchlaufen bzw. ein "Ausweich-Prozeß" angestoßen, anstelle zu warten

- **Mehrfachoperationen**: Der Aufruf der P- bzw. V-Methode wirkt sich gleichzeitig auf mehrere Semaphore aus

- **Binäres Semaphor**: Der Semaphorzähler kann nur die Werte 0 oder 1 annehmen, so daß bei der Implementierung eines Semaphors im Betriebssystems für den Zähler ein Bit ausreicht.

Exemplarisch wird eine Realisierungsmöglichkeit des Konzepts **Semaphor mit Alternative** beschrieben. Die Prozedur, die aufgerufen wird, falls das zu passierende Semaphor den aufrufenden Prozeß in den Zustand WARTEND versetzen würde, habe den Datentyp

```
TYPE alt_proc = PROCEDURE (akt_count : INTEGER);
```

Diese Deklaration ist am Anfang des `INTERFACE`-Teils der Unit `Semaphor` eingefügt. Die Objekttypdeklaration `Tsemaphor` wird um eine Methode `PnoWait` erweitert, der als Aktualparameter die Alternativprozedur mitgegeben wird:

Methode	Bedeutung
...	
`PROCEDURE Tsemaphor.PnoWait (proc : alt_proc);`	wie die Methode `P`, nur wird die Prozedur `proc` aufgerufen, falls das Semaphor sperren würde, und der Prozeß nicht unterbrochen
Bedeutung des Parameters: `proc` Alternativprozedur	
...	

Die Änderungen in der Unit `semaphor` lauten:

```
TYPE alt_proc = PROCEDURE (akt_count : INTEGER);

     Psemaphor = ^Tsemaphor;
     Tsemaphor =
       OBJECT (TSyncelement)
          PRIVATE
            count : INTEGER;     { Semaphorzähler                  }
          PUBLIC
            CONSTRUCTOR init (a : INTEGER);
                      { das Semaphor wird eingerichtet und sein
                        Zähler mit a initialisiert                 }
            ...
            PROCEDURE PnoWait (proc : alt_proc);
                      { wie die Methode P, nur wird die Prozedur proc
                        aufgerufen, falls das Semaphor sperren würde,
                        und der Prozeß nicht unterbrochen          }
            ...
        END;
```

und

```
PROCEDURE Tsemaphor.PnoWait (proc : alt_proc);

BEGIN { Tsemaphor.PnoWait }
  MTBlock;
  IF count <= 0
  THEN BEGIN { das Semaphor sperrt }
         MTContinue;
         proc (count);
       END
  ELSE BEGIN
         count := count - 1;
         { Semaphorkennung in die Liste der passierten Semaphore
           eintragen:}
         AktTask^.PassedSem^.insert (@Self);
         MTContinue;
       END;
END   { Tsemaphor.PnoWait };
```

11.4.2 Ereignisvariablen

Einen sehr einfacher Synchronisationsmechanismus bieten Ereignisvariablen. Ein Prozeß wartet eventuell auf das Eintreten eines Ereignisses oder einer Auswahl von Ereignissen und stößt dann, je nachdem, welches der spezifizierten Ereignisse eingetreten ist, eine entsprechende Aktion an. Ein **Ereignis (event)** ist in diesem Zusammenhang ein binärer Wert, das eingetreten sein kann oder nicht, und wird vom Prozeß definiert und in seiner Bedeutung interpretiert und vom Betriebssystem (der Multitasking-Komponente) verwaltet.

Ein typisches Beispiel findet man in Multiuser-Systemen, in denen jeweils ein Prozeß ("Eingabeprozeß") mit der Entgegennahme von Eingaben eines Anwenders verbunden ist. Ein Ereignis ist hier die erneute Eingabe (Texteingabe, Mausklick, Funktionstaste usw.) des jeweiligen Anwenders. Pro Anwender könnte man einen eigenen Eingabeprozeß definieren, müßte dann aber viele sehr ähnlich ablaufende Eingabeprozesse verwalten. Stattdessen kann man auch einen einzigen Eingabeprozeß definieren, der auf das Ereignis "Eingabe von irgendeinem Anwender" wartet und dann entsprechend der

Eingabe reagiert. Dieser Prozeß wartet also auf das Eintreten mindestens eines Ereignisses aus einer Menge von möglichen Ereignissen, d.h. der Prozeß reagiert, sobald eines der definierten Ereignisse eingetreten ist. Es handelt sich hierbei um eine logische ODER-Verknüpfung von Ereignissen. Entsprechend kann man sich auch eine logische UND- und eine logische NICHT-Verknüpfung von Ereignissen vorstellen: es wird auf das Eintreten aller Ereignisse in einer Menge von Ereignissen gewartet (logische UND-Verknüpfung) bzw. auf das Eintreten von Ereignissen, die nicht zu einer definierten Ereignismenge gehören (logische NICHT-Verknüpfung).

Eine **Ereignisvariable** definiert eine Menge von Ereignissen, auf deren Eintreten man warten kann, und zwar entweder auf das Eintreten mindestens eines dieser definierten Ereignisse (**logische ODER-Verknüpfung von Ereignissen**), das Eintreten aller dieser definierten Ereignisse (**logische UND-Verknüpfung von Ereignissen**) oder das Eintreten eines Ereignisses aus der Komplementmenge der angegebenen Ereignisse (**logische NICHT-Verknüpfung von Ereignissen**). Entsprechend **sperrt die Ereignisvariable**, falls bei logischer ODER-Verknüpfung der Ereignisse noch nicht mindestens ein definiertes Ereignis eingetreten ist bzw. falls bei logischer UND-Verknüpfung der Ereignisse noch nicht alle Ereignisse eingetreten sind bzw. falls bisher noch kein Ereignis eingetreten ist, das nicht zur angegebenen Ereignismenge gehört. Es ist zu beachten, daß ein Ereignis der durch die Ereignisvariable definierten Ereignismenge eingetreten sein kann, bevor überhaupt ein Prozeß darauf wartet. Die Reaktion eines Prozesses, die aufgrund des Eintretens von Ereignissen ausgelöst wird, versetzt das auslösende Ereignis in den Zustand "nicht eingetreten" zurück, d.h. Ereignisse werden durch den bisher auf sie wartenden Prozeß "verbraucht" (zurückgesetzt).

Eine weitere Detaillierung des Mechanismus zur Synchronisation mit Hilfe von Ereignisvariablen ergibt sich aus dem folgenden Implementierungsvorschlag. Zur Realisierung von Ereignisvariablen wird in einer Unit `ereignis` ein Objekttyp `TEreignis` deklariert:

```
TYPE ereignis_typ = (oder, und, nicht);

     PEreignis = ^TEreignis;
     TEreignis =
       OBJECT (TSyncelement)
         PRIVATE
           menge   : WORD;   { Ereignismenge                        }
           gesetzt : WORD;   { bereits eingetretene Ereignisse      }
           typ     : ereignis_typ;
                             { Art der logischen Verknüpfung der
                               Ereignisse                           }
         PUBLIC
           CONSTRUCTOR init (ereignismenge : WORD;
                             wartetyp      : ereignis_typ);
           PROCEDURE waitevent (VAR ereignisse : WORD);
           PROCEDURE sendevent (ereignisse : WORD);
           DESTRUCTOR close; VIRTUAL;
     END;
```

Bei der Einrichtung eines Objekts vom Typ `TEreignis` wird die Menge der Ereignisse, die diese Ereignisvariable definiert, in der Komponente
`menge`
abgelegt. Eine Ereignisvariable kann bis zu 16 Ereignisse zusammenfassen. Die Komponente `typ` gibt an, ob das Eintreten dieser Ereignisse durch logisches UND bzw. ODER bzw. NICHT verbunden werden soll. In der Komponente
`gesetzt`
werden die bisher eingetretenen, aber noch nicht verbrauchten Ereignisse gespeichert. Auf diese Weise gehen keine eingetretenen Ereignisse verloren, auch wenn bisher noch kein Prozeß überhaupt auf sie gewartet hat.

Es können mehrere Prozesse dieselbe Ereignisvariable verwenden. Falls die Ereignisvariable sperrt, werden die TCBs der auf das Eintreten wartenden Prozesse in der FIFO-Warteschlange der Ereignisvariablen (Komponente `list`) festgehalten und die Prozesse in den Zustand WARTEND versetzt. Sobald Ereignisse so eingetreten sind, daß die Ereignisvariable nicht sperrt, werden alle an dieser Ereignisvariablen wartenden Prozesse wieder aktiviert.

Die folgende Tabelle beschreibt die Methoden, die der Objekttyp zusätzlich zu den aus `TSyncelement` geerbten Methoden definiert.

Methode	Bedeutung
`CONSTRUCTOR Tereignis.Init (ereignismenge : WORD; wartetyp : ereignis_typ)`	Die Ereignisvariable wird eingerichtet.
Bedeutung der Parameter: `ereignismenge` Menge der Ereignisse dieser Ereignisvariablen. Jedes Ereignis ist einem Bit zugeordnet. `wartetyp` Art der Verknüpfung der eingetretenen Ereignisse	

`PROCEDURE Tereignis.waitevent (VAR ereignisse : WORD)`	Der aufrufende Prozeß wartet auf das Eintreten der definierten Ereignisse (in der in der `init`-Methode angegebenen logischen Verknüpfung). Falls die Ereignisvariable sperrt, wird der TCB des Prozesses in die FIFO-Warteschlange der Ereignisvariable eingetragen und der Prozeß in den Zustand WARTEND versetzt. Sperrt die Variable nicht oder sind die Ereignisse in der definierten logischen Kombination eingetreten, so wird dem Prozeß im Aktualparameter des Methodenaufrufs mitgeteilt, aufgrund welcher Ereignisse er wieder aktiviert wurde.
Bedeutung des Parameters: `ereignisse` eingetretene Ereignisse, die den Prozeß haben weiterlaufen lassen	
`PROCEDURE Tereignis.sendevent (VAR ereignisse : WORD)`	Das Eintreten von Ereignissen wird bekanntgegeben.
Bedeutung des Parameters: `ereignisse` eingetretene Ereignisse	
`DESTRUCTOR Tereignis.close;`	Die Ereignisvariable wird aus dem System entfernt.

Der IMPLEMENTATION-Teil der Unit `Ereignis` zeigt die Realisierung der Methoden:

```
IMPLEMENTATION

USES Multitsk;

TYPE PtrLayout = RECORD
                   Ofs : WORD;
                   Seg : WORD;
                 END;


CONSTRUCTOR Tereignis.init (ereignismenge : WORD;
                            wartetyp      : ereignis_typ);

BEGIN { Tereignis.init }
  INHERITED init;
  IF wartetyp = nicht
  THEN BEGIN
         menge := NOT ereignismenge;
         typ   := oder;
       END
  ELSE BEGIN
         menge := ereignismenge;
         typ   := wartetyp
       END;
  gesetzt := NOT menge;
END   { Tereignis.init };
```

```
PROCEDURE Tereignis.waitevent (VAR ereignisse : WORD);

VAR Ptr : Pointer;

BEGIN { Tereignis.waitevent }
  MTBlock;
  IF (typ = oder) AND ((menge AND gesetzt) <> 0)
     OR
     (typ = und)  AND (gesetzt = $FFFF)
  THEN BEGIN { die Ereignisse sind bereist eingetreten          }
         ereignisse := menge AND gesetzt;
         gesetzt    := NOT menge;
         MTContinue;
       END
  ELSE BEGIN
         { eigenen TCB in die Warteschlange eintragen:          }
         list^.insert (AktTask);
         { Übergang in den Zustand WARTEND; der Taskwechsel wird
           automatisch durchgeführt:                            }
         MTWait (@Self);
         { die den Wartezustand beendenden eingetretenen
           Ereignisse lesen:                                    }
         Ptr        := AktTask^.GetParameter;
         ereignisse := PtrLayout(Ptr).Ofs;
       END;
END   { Tereignis.waitevent };


PROCEDURE Tereignis.sendevent (ereignisse : WORD);

VAR task : TaskPtr;
    Ptr  : Pointer;

BEGIN { Tereignis.sendevent }
  MTBlock;
  { Nur gültige Ereignisse zulassen und setzen: }
  ereignisse := ereignisse AND menge;
  gesetzt := gesetzt OR ereignisse;

  IF (typ = oder) AND ((menge AND gesetzt) <> 0)
     OR
     (typ = und)  AND (gesetzt = $FFFF)
  THEN BEGIN { alle Ereignisse sind bereist eingetreten         }
         { wartende Prozesse wieder aktivieren und über das
           Eintreten der Ereignisse informieren                 }
         list^.delete(Pointer(task));
         IF task <> NIL
         THEN BEGIN
                IF typ = oder THEN PtrLayout(Ptr).Ofs := ereignisse
                              ELSE PtrLayout(Ptr).Ofs := menge;
                WHILE task <> NIL DO
                  BEGIN
                    task^.PutParameter (Ptr);
                    MTSignal (task);
                    list^.delete(Pointer(task));
                  END;
                { eingetretene Ereignisse zurücksetzen          }
                gesetzt := NOT menge;
              END;
       END;
  MTContinue;
END   { Tereignis.sendevent };
```

```
DESTRUCTOR Tereignis.close;

BEGIN { Tereignis.close }
  INHERITED done (NIL);
END   { Tereignis.close };
```

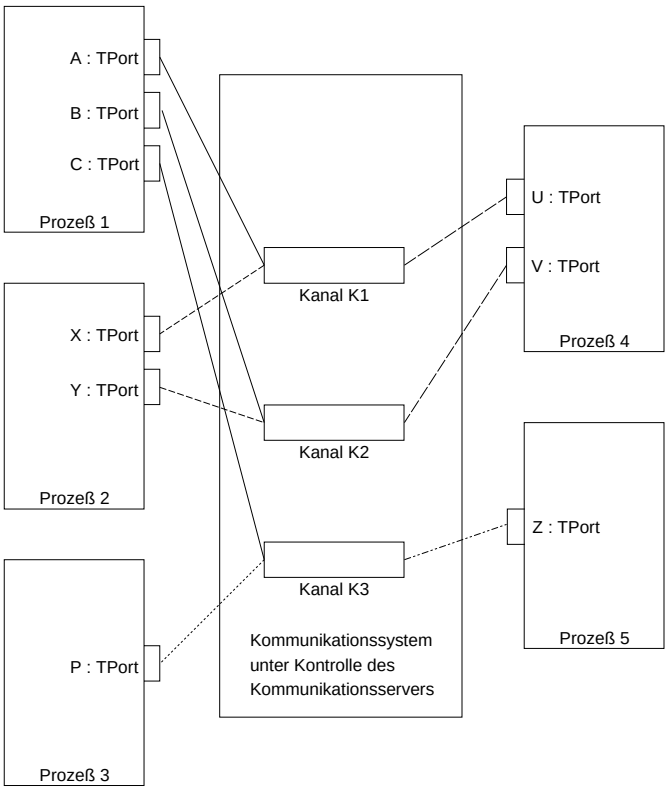
11.5 Nachrichtenaustausch zwischen Prozessen (Interprozeßkommunikation)

Die in Kapitel 11.4 beschriebenen Mechanismen zur Prozeßsynchronisation setzen voraus, daß alle Prozesse (zumindest indirekt über das Betriebssystem) auf einen gemeinsamen Speicherbereich zugreifen können, in dem Synchronisationsmittel wie Semaphoren und Ereignisvariablen verwaltet werden. Besonders in Systemen jedoch, in denen die Prozesse auf mehrere Rechner verteilt sind, haben konkurrierende Prozesse häufig keinen Zugriff auf einen gemeinsamen Datenbereich. **Synchronisationsverfahren in verteilten Systemen** beruhen daher nicht auf der Manipulation von Variablen, auf die die beteiligten Prozesse gemeinsam zugreifen[44]. Die Synchronisation kann vielmehr auf einem allgemeinen **Mechanismus zur Kommunikation zwischen verteilten Prozessen** aufbauen.

Im folgenden wird ein System zum **Nachrichtenaustausch zwischen Prozessen (Interprozeßkommunikation) über logische Kanäle** definiert, wie er in unterschiedlichen Ausprägungen in realen Betriebssystemen vorkommt. Spezielle Mechanismen wie Pipes, Mailboxen, dynamische Intertaskkommunikation, Protokolle wie TCP/IP usw. ordnen sich in dieses Konzept ein. Der Nachrichtenaustausch dient natürlich primär dem Ziel, Daten zwischen Prozessen zu übermitteln. Die Synchronisationsfähigkeit ergibt sich aus dem Protokollverhalten mit impliziten Wartezuständen. Im wesentlichen kombiniert der Nachrichtenaustausch Mechanismen wie Puffermanipulation und ereignisgesteuertes Verhalten (Ereignisse sind hier: "Eine Nachricht ist noch nicht angekommen" bzw. "Eine Nachricht ist bereits da").

Das im folgenden beschriebene Kommunikationsmodell identifiziert eine Kommunikationsverbindung zwischen Anwendunge (Threads) in einem zweistufigen Verfahren: Zwischen Kommunikationspartnern wird zunächst ein **(logischer) Kanal** eingerichtet, der Kommunikationsströme in beide Richtungen zuläßt. An diesen Kanal können sich dann mehrere Prozesse anschließen, indem sie jeweils ein als **Port** bezeichnetes Datenobjekt in ihrem jeweiligen Adreßbereich einrichten. Ein Kanal modelliert die Verbindung von Host-Rechnern, auf denen jeweils mehrere Anwendungen laufen. Aus Sicht dieser Anwendungen bietet der Kanal eine Kommunikationsverbindung auf Rechnerebene. Ein Port ist dann der lokale Anschlußpunkt einer Anwendung an diese Kommunikationsverbindung.

[44] In einem System mit gemeinsam verfügbaren Datenbereichen, z.B. in Multiprocessing-Betriebssystemen können die folgenden Mechanismen natürlich auch verwendet werden.



Prozeß	ist verbunden über den Port	mit Prozessen
1	A B C	2, 4 2, 4 3, 5
2	X Y	1, 4 1, 4
3	P	1, 5
4	U V	1, 2 1, 2
5	Z	1, 3

Abbildung 11.5-1: Kommunikationskonzept

Abbildung 11.5-1 zeigt das Kommunikationskonzept: fünf Prozesse kommunizieren miteinander über drei Kanäle und haben dazu entsprechende Ports eingerichtet, die die angegebene Kommunikationsstruktur definieren. Alle Prozesse könnten auf demselben oder auf verschiedenen Rechnern laufen. Beispielsweise können jetzt Nachrichten, die Prozeß 1 an Port B sendet, von Prozeß 4 über Port V empfangen werden, nicht aber über Port U. Entsprechend können Prozeß 1 oder Prozeß 3 Nachrichten, die Prozeß 5 an Port Z absetzt, über Port C bzw. Port P entgegennehmen; in diesem Fall kann der sendende Prozeß 5 nicht

kontrollieren, wer von beiden möglichen Empfängern die Nachrichten erhält. Um gezielt einen einzigen Empfänger von Nachrichten anzusprechen, muß ein eigener Kanal für diese Kommunikationsverbindung eingerichtet werden.

Insgesamt werden **verbindungsorientierte Kommunikationsströme** nachgebildet. Hierbei können mehrere Sender gleichberechtigt einen Kommunikationsstrom bedienen bzw. mehrere Empfänger von ihm profitieren. Außerdem sind die Rollen als Sender bzw. Empfänger für die beteiligten Prozesse nicht festgelegt, d.h. eine Prozeß kann sowohl Nachrichten an eine Kommunikationsverbindung absetzen als auch von ihr empfangen.

11.5.1 Modellierung der Interprozeßkommunikation

In diesem Kapitel wird der Mechanismus beschrieben, der vom vorliegenden Modell zur Interprozeßkommunikation bereitgestellt wird.

Systemweit wird ein Kommunikationsmechanismus in einem verteilten System über Prozeß- und Rechnerknoten hinweg von einem dedizierten Dienst, dem **Kommunikationsserver**, gesteuert, der alle durch Kanäle definierten Kommunikationsverbindungen kontrolliert und mindestens die Identifikationen kennt, über die ein jeweiliger Prozeß eine Kommunikationsverbindung anspricht. Es wird hier nicht der Begriff Netzwerkserver verwendet, um anzudeuten, daß es sich bei der **Steuerung der Kommunikationsverbindungen** um eine **Tätigkeit auf einer logischen Ebene** handelt im Gegensatz zu der Steuerung und Verwaltung einer Netzwerk-Hardwarearchitektur.

Zunächst ist nicht festgelegt, nach welchen syntaktischen Regeln Prozesse an unterschiedlichen Stellen im verteilten System Kommunikationsverbindungen bezeichnen. Jeder Prozeß, der Nachrichten mit anderen Prozessen über einen logischen Kanal austauschen möchte, muß sich dafür beim Kommunikationsserver anmelden und den Kanal benennen. Es kann das Prinzip verwendet werden, daß der Kommunikationsserver einem an der Kommunikation beteiligten Prozeß bei dessen Anmeldung vorschreibt, wie er eine Kommunikationsverbindung von nun an anzusprechen hat. In diesem Kapitel soll eine **Kanalidentifikation durch eine numerische Kennung** erfolgen, die zur Vereinfachung den Datentyp

```
TYPE Tname = WORD;
```

hat. Die Kanalidentifikation wird von einem Prozeß nur beim Anschluß an den Kanal (Initialisierung der Kommunikationsverbindung) verwendet. Bei allen folgenden Datenübertragung werden Ports verwendet.

Die zwischen zwei Prozessen im verteilten System auszutauschenden Nachrichten sind semantisch vom gleichen Datentyp, z.B. eine Folge von `INTEGER`-Daten. Die interne Darstellung der auszutauschenden Daten in den an der Kommunikation beteiligten Rechnerknoten kann jedoch sehr unterschiedlich sein. Beispielsweise können sich in verschiedenen Rechnersystemen die Reihenfolge von höherwertigen und niedrigwertigen Bytes bei `INTEGER`-Zahlen unterscheiden (vgl. Kapitel 4.1). Derartige technische

Details kann ein Prozeß über entfernte Prozesse nicht kennen. Als grundlegendes Datenform könnte eine Bitfolge vereinbart werden. Ein Prozeß verwendet in seinen Programmen eigene Datentypen und überläßt eine eventuell erforderliche Transformation der internen Datendarstellungen zwischen unterschiedlichen Prozessen Diensten des Betriebssystems bzw. dem Kommunikationssystem. In der hier beschriebenen exemplarischen Realisierung wird daher eine weitere Vereinfachung vorgenommen: eine Nachricht hat einen einheitlichen Datentyp

```
TYPE Tmessage = STRING;
```

Aus Prozeßsicht ist die Nachrichtenübertragung zudem zuverlässig und fehlerfrei, d.h. ein Kanal überträgt Nachrichten unverfälscht. Bei fehlerhaften Übertragungen aufgrund technischer Gegebenheiten sorgt das zugrundeliegende Kommunikationssystem für eventuell erforderliche Sendewiederholungen bzw. Fehlerkorrekturen. **Alle technischen Details der Datenübertragung sind für die beteiligten Prozesse transparent**.

Bevor ein Prozeß mit einem anderen Prozeß Nachrichten austauschen kann, muß er sich an einen Kommunikationskanal anschließen. Er definiert dazu in seinem Adreßbereich einen Port, der mit genau einem Kommunikationskanal verbunden wird. Der Port ist ein Datenobjekt, dessen Methoden Daten an einen Kanal übertragen bzw. Daten aus einem Kanal lesen und dabei eine eventuelle Synchronisation zwischen Prozeß und Datenstrom bewirken. Der Port ist die einzige Verbindungsmöglichkeit eines Prozesses zur Außenwelt und entkoppelt die internen Prozeßläufen vom Kommunikationsnetz. Ein Prozeß kann mehrere Ports einrichten und sich damit an mehrere Kommunikationskanäle anschließen. Dadurch kann ein Prozeß mit einem anderen Prozeß unterschiedliche Nachrichtenströme abwickeln bzw. Nachrichtenströme mit unterschiedlichen Prozessen unterhalten. Unterschiedliche Prozesse können sich über jeweils einen eigenen Port mit demselben Kommunikationskanal verbinden. Ein Prozeß kennt dabei nur seine eigenen Ports und hat insbesondere keine direkte Zugriffsmöglichkeit auf den Kanal und überhaupt keine direkte Verbindung zu Ports anderer Prozesse.

Die Adressen aller Ports eines Prozesses stehen in einer Liste, die über die TCB-Komponente

```
Ports
```

adressiert wird. Bei Entfernung eines Prozesses aus der Multitasking-Kontrolle werden zuvor auch alle Ports des Prozesses freigegeben und Verbindungen zum jeweiligen Kanal gelöst (in der Methode `TD.done` des TCBs).

Daten vom Typ `Tmessage` werden an der Port gesendet bzw. aus dem Port empfangen. Falls ein Empfangswunsch von Daten über einen Port nicht befriedigt werden kann, weil gerade keine Daten am Port anstehen, gerät der Prozeß am Port in den Zustand WARTEND. Der Port ist also ein Synchronisationselement im jeweiligen Prozeß. Andererseits ist bei einem Sendewunsch der Port immer bereit, Daten vom Typ `Tmessage` entgegenzunehmen und in den angeschlossenen Kommunikationskanal zu stellen, so daß der sendende Prozeß nicht unterbrochen wird.

Im folgenden werden Prozesse wieder von der Unit `Multitsk` gesteuert und daher in ein gemeinsames Rahmenprogramm eingebettet. Der Eindruck einer vollständigen Verteilung der Prozesse auf mehrere Systeme wird dadurch etwas verwischt. Im Unterschied zu den bisher beschriebenen Synchronisationsmitteln, bei denen die Initialisierung aus Prozeßsicht global in dem gemeinsamen Rahmenprogramm erfolgte, richtet ein Prozeß einen Port lokal in dem von ihm durchlaufenen Programm ein und schließt sich damit an einen Kommunikationskanal an. Im Unterschied zu den definierten Ports liegen jedoch alle Kommunikationskanäle in der Kontrolle des Kommunikationsservers außerhalb des Zugriffs der Prozesse.

Zur **Definition des Nachrichtenaustauschs** zwischen Prozessen wird eine Unit `IPK` definiert, die Objekttypdeklarationen enthält, die jeder Prozeß benötigt, der Nachrichten mit anderen Prozessen austauschen möchte. Die Unit `IPK` richtet in ihrem Initialisierungsteil mit der Anweisung

```
{ Kommunikationsserver: }
KanalListe := New (PIPKListe, init); { Kanalliste einrichten      }
```

ein Datenobjekt ein, das den Kommunikationsserver simuliert und im wesentlichen aus einer Liste von definierten Kommunikationskanälen mit entsprechenden Methoden besteht. Der Kommunikationsserver ist also implizit in der Unit `IPK` enthalten, ohne daß ein spezifischer Prozeß hierfür eingerichtet werden muß. Alle Datenobjekte des Kommunikationsservers sind im `IMPLEMENTATION`-Teil der Unit `IPK` verborgen.

Ein Kommunikationskanal nimmt eine Nachrichten, die ein Prozeß über seinen entsprechenden Port an einen Kanal sendet, entgegen und puffert sie nach dem FIFO-Prinzip solange, bis sie von einem anderen Prozeß abgeholt wird. Der Datenübertragungsvorgang an den Port beansprucht im sendenden Prozeß eine Zeit, die proportional zur Datenmenge (Nachrichtenlänge) ist. Ein die Nachricht abholender Prozeß verwendet dazu seinen eigenen Port, der mit demselben Kommunikationskanal verbunden ist. Zu beachten ist, daß die Ankunftsreihenfolge von Nachrichten am Kanal von der (nichtdeterministischen) Reihenfolge abhängt, in der Sendeanforderungen an den Kanal abgewickelt werden. Durch den Empfang einer Nachricht wird diese aus dem Kanal entfernt[45]. Die reine Datenübertragung von einem Kanal über den empfangenden Port zu einem Prozeß nimmt eine Zeit in Anspruch, die ebenfalls proportional zur Datenmenge ist. In der Unit `IPK` werden diese Übertragungszeiten durch die im `IMPLEMENTATION`-Teil deklarierte Prozedur

```
PROCEDURE simulated_message_transfer (lng : WORD);
{ simuliert die Übertragungsdauer einer Nachricht zwischen Prozeß
  und Kanal                                                       }

BEGIN { simulated_message_transfer }
  Delay (10 * lng);
END   { simulated_message_transfer };
```

[45] Das bedeutet, daß das Modell in der vorliegenden Form keine Rundspruchmeldungen (Nachrichten im burst-mode) unterstützt, die gleichzeitig an mehrere Empfänger gehen.

simuliert.

Die möglichen zeitlichen Verzögerungen in den Abläufen der Prozesse zeigt Abbildung 11.5.1-1. Dabei sind die Situationen zu unterscheiden, daß beim Empfangswunsch eine Nachricht bereits im Kanal ansteht bzw. ein Sender eine Nachricht erst später absetzt.

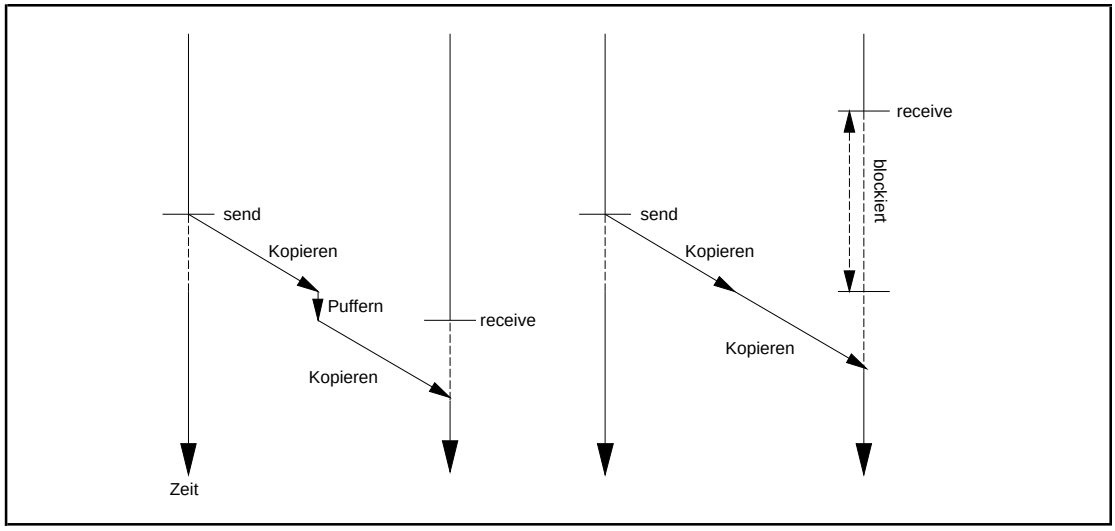


Abbildung 11.5.1-1: Zeitliche Verzögerungen beim asynchronen Nachrichtenaustausch

Die Objekttypdeklaration eines Ports (aus dem `INTERFACE`-Teil der Unit `IPK`) lautet:

```
TYPE Pport = ^Tport;
     Tport = OBJECT (TEreignis)
               PRIVATE
                 name : Tname; { Identifikation des verbundenen
                                 Kommunikationskanals              }
                 adr  : Pmessage;
                              { simulierter Empfangspuffer         }
               PUBLIC
                 CONSTRUCTOR connect (kanalID : Tname);
                 PROCEDURE send (data : Tmessage);
                 PROCEDURE receive (VAR data : Tmessage);
                 PROCEDURE final_handling (taskid : Pointer);
                                                          VIRTUAL;
                  DESTRUCTOR disconnect; VIRTUAL;
                  DESTRUCTOR done (taskid : Pointer); VIRTUAL;
                END;
```

Der Port ist also ein abgeleitetes Synchronisationselement, das als zusätzliche Komponenten die Identifikation des verbundenen Kommunikationskanals und einen (simulierten) Empfangspuffer hat, in den der Kanal beim Datenempfang die Daten einstellt.

Die einzelnen Methoden haben folgende Bedeutung:

Methode	Bedeutung
`CONSTRUCTOR Tport.connect (kanalID : Tname);`	Der Port wird an einen bereits im Kommunikationsserver existierenden Kanal angeschlossen, bzw. der Kommunikationsserver wird zur Einrichten eines neuen Kanals aufgefordert.
Bedeutung des Parameters: `kanalID` Identifikation des Kommunikationskanals	
`PROCEDURE send (data : Tmessage);`	Es werden Daten an den Port (und damit implizit an den angeschlossenen Kanal) gesendet. Die zeitliche Dauer der Operation ist proportional zur Datenmenge (Datenlänge). Der Prozeß wird nicht blockiert.
Bedeutung des Parameters: `data` zu sendende Nachricht	
`PROCEDURE receive (VAR data : Tmessage);`	Daten werden aus dem Port empfangen. Falls zur Zeit keine Daten im Kanal anstehen, wird der Prozeß bis zur Ankunft von Daten in den Zustand WARTEND versetzt. Anschließend wird der weitere Prozeßablauf um eine Zeitspanne verzögert, die proportional zur empfangenen Datenmenge ist.
Bedeutung des Parameters: `data` empfangene Nachricht	
`PROCEDURE final_handling (taskid : Pointer); VIRTUAL;`	Abschlußbehandlung bei Entfernung des Prozesses durch `MTKillTask`: Dabei wird die Verbindung zum Kanal gelöst. Die Methode darf vom anwendenden Prozeß nicht selbst aufgerufen werden. Sie wird verwendet, um dem vererbenden Objekttyp `TSyncelement` eine eigene virtuelle Methode mit Bezeichner `final_handling` bereitzustellen (vgl. Kapitel 11.4)
Bedeutung des Parameters: `taskid` Zeiger auf den TCB des Prozesses, der den Port enthält	
`DESTRUCTOR disconnect; VIRTUAL;`	Der Port wird entfernt. Dies ist die von einem Anwenderprozeß verwendete Methode zum Schließen eines Ports.
`DESTRUCTOR done (taskid : Pointer); VIRTUAL;`	Der Port wird entfernt. Ein Prozeß verwendet zum Abbau eines Ports die Methode `disconnect`. Der Destruktor `done` wird verwendet, um dem vererbenden Objekttyp `TSyncelement` einen eigenen virtuellen Destruktor mit Bezeichner `done` bereitzustellen (vgl. Kapitel 11.4)

Die Funktionsweisen der Methoden des Objekttyps `Tport` haben eine Schnittstelle zum Mechanismus, mit dem der Kommunikationsserver die Kanäle verwaltet: Ein definierter Kanal wird vom Kommunikationsserver in einem Datenobjekt vom Typ `Tkanal`

beschrieben. Alle Kanäle sind in der bereits angegebenen Liste `KanalListe` zusammengefaßt, die im Initialisierungsteil der Unit `IPK` eingerichtet wird und in ihrem `IMPLEMENTATION`-Teil deklariert wird:

```
TYPE Pkanal = ^Tkanal;
     Tkanal = OBJECT
                PRIVATE
                  kanalID   : Tname;   { Kanalidentifizierung       }
                  usercount : INTEGER; { Anzahl angeschlossener
                                         Prozesse                   }
                  messages  : PFIFO;   { am Kanal anstehende
                                         Nachrichten                }
                  ports     : PFIFO;   { Ports der auf Nachrichten
                                         wartenden Prozesse         }
                PUBLIC
                  CONSTRUCTOR init (ID : Tname);
                  DESTRUCTOR done;
              END;


     PIPKListe = ^TIPKListe;
     TIPKListe = OBJECT (TListe)
                   FUNCTION find (kanalID : WORD) : Pkanal;
                                { sucht den zum Kanal mit
                                  Identifikation kanalID gehörenden
                                  Eintrag. Der Rückgabewert ist NIL,
                                  falls es keinen entsprechenden
                                  Eintrag gibt.                     }

                 END;


VAR KanalListe : PIPKListe;  { verwaltete Kanäle; das Objekt mit
                               seinen Methoden repräsentiert den
                               Kommunikationsserver                 }
```

Neben der Identifikation eines Kanals (Komponente `Tkanal.kanalID`) wird die Anzahl angeschlossener Ports (Komponente `Tkanal.usercount`) vermerkt. Die durch `Tkanal.messages` adressierte FIFO-Liste puffert alle im Kanal stehenden und noch nicht abgeholten Nachrichten. In der durch `Tkanal.ports` adressierten FIFO-Liste werden die Port-Adressen derjenigen Prozesse vermerkt, die auf eine Nachricht warten und gerade in ihrem entsprechenden Port blockiert sind.

Die Methoden eines Datenobjekts `Kanal` sind sehr einfach und bedürfen keiner weiteren Erläuterung:

```
CONSTRUCTOR Tkanal.init (ID : Tname);

BEGIN { Tkanal.init }
  kanalID   := ID;
  usercount := 1;
  New (messages, init);
  New (ports, init);
END   { Tkanal.init };


DESTRUCTOR Tkanal.done;
```

```
BEGIN { Tkanal.done }
  Dispose (messages, done);
  Dispose (ports, done);
END   { Tkanal.done };
```

Die Methoden des Datentyps `Tport` sind im folgenden aufgeführt. Zu beachten ist, daß bei Einrichtung eines neuen Ports geprüft wird, ob es bereits einen Kanal mit der angegebenen Identifikation gibt. Falls dies nicht der Fall ist, wird ein derartiger Kanal im Kommunikationsserver eingerichtet. Konzeptionell müßte an dieser Stelle ein Auftrag an den Kommunikationsserver zur Einrichtung eines neuen Kanals erteilt werden. Bei Entfernung eines Ports (Methoden `Tport.disconnect` bzw. `Tport.done`) wird über die Methode `Tport.final_handling` geprüft, ob noch weitere Ports am Kanal mit der Kanalidentifikation des zu entfernenden Ports angeschlossen sind. Falls nicht, wird der Kanal aus dem Kommunikationsserver entfernt. Auch hier müßte konzeptionell ein entsprechender Auftrag an den Kommunikationsserver erteilt werden. Zur Vereinfachung der Mechanismen sind diese Auftragsabwicklungen in die Methoden des Objekttyps `Tport` eingebaut. Weitere Erklärungen sind den eingefügten Kommentaren zu entnehmen.

```
CONSTRUCTOR Tport.connect (kanalID : Tname);

VAR neuerKanal : Pkanal;

BEGIN { Tport.connect }
  INHERITED init (for_message, und);

  name := kanalID;
  adr  := NIL;

  MTBlock;
  { Kommunikationsport im TCB einhängen: }
  AktTask^.Ports^.Insert (@Self);
  { evtl. neuen Kanal einrichten: }
  neuerKanal := KanalListe^.find (kanalID);
  IF neuerKanal = NIL
  THEN BEGIN
         New (neuerKanal, init (kanalID));
         KanalListe^.insert (neuerKanal);
       END
  ELSE Inc (neuerKanal^.usercount);
  MTContinue;
END   { Tport.connect };


PROCEDURE Tport.send (data : Tmessage);

VAR kanal    : Pkanal;
    mess_ptr : Pmessage;
    recport  : Pport;

BEGIN { Tport.send }
  { Nachricht an den Port und damit in den angeschlossenen Kanal
    übertragen: }
  kanal := KanalListe^.find (name);
  simulated_message_transfer (Length (data));
  New (mess_ptr);
  mess_ptr^ := data;
```

```
  MTBlock;
  kanal^.ports^.delete (Pointer(recport));
  IF recport <> NIL
  THEN BEGIN { Nachrichtenadresse übergeben und wartenden Prozeß
               anstoßen }
         recport^.adr := mess_ptr;
         MTContinue;
         recport^.sendevent (for_message);
       END
  ELSE BEGIN { Nachricht puffern }
         kanal^.messages^.insert (mess_ptr);
         MTContinue;
       END;
END   { Tport.send };


PROCEDURE Tport.receive (VAR data : Tmessage);

VAR kanal    : Pkanal;
    dummy    : WORD;
    mess_ptr : Pmessage;

BEGIN { Tport.receive }
  { eine Nachricht aus dem Port empfangen bzw. auf eine Nachricht
    warten: }
  kanal := KanalListe^.find (name);
  MTBlock;
  kanal^.messages^.delete (Pointer(mess_ptr));
  IF mess_ptr = NIL
  THEN { keine Nachricht vorhanden: warten! }
       BEGIN
         kanal^.ports^.insert (@Self);
         MTContinue;
         Self.waitevent (dummy);
       END
  ELSE BEGIN
         adr := mess_ptr;
         MTContinue;
       END;
  { Daten übernehmen: }
  data := adr^;
  Dispose (adr);
  simulated_message_transfer (Length (data));
END   { Tport.receive };


PROCEDURE Tport.final_handling (taskid : Pointer);

VAR kanal    : Pkanal;

BEGIN { Tport.final_handling }
  MTBlock;
  kanal := KanalListe^.find (name);
  kanal^.ports^.delete_entry (@Self);
  Dec (kanal^.usercount);
  IF kanal^.usercount <= 0
  THEN BEGIN
         KanalListe^.delete (erstes, kanal);
         Dispose (kanal, done);
       END;
  MTContinue;
  TaskPtr(taskid)^.Ports^.delete_entry (@Self);
  INHERITED done (taskid);
END   { Tport.final_handling };
```

```
DESTRUCTOR Tport.disconnect;

VAR kanal : Pkanal;

BEGIN { Tport.disconnect }
  final_handling (AktTask);
END   { Tport.disconnect };


DESTRUCTOR Tport.done (taskid : Pointer);

BEGIN { Tport.done }
  final_handling (taskid);
END   { Tport.done };
```

11.5.2 Ein Beispiel zum Nachrichtenaustausch

Eine einfache Anwendung der Interprozeßkommunikation zeigt Abbildung 11.5.2-1. Ein Produzentprozeß, dessen TCB-Adresse in der Variablen
`VAR produzent_task : TaskPtr;`
steht, erzeugt Nachrichten, die er an einen angeschlossenen Port mit Adresse
`VAR sendport : Pport;`
sendet. Zwei Konsumentprozesse mit TCB-Adressen in den Variablen
```
VAR konsument_task_1 : TaskPtr;
    konsument_task_2 : TaskPtr;
```
empfangen die Nachrichten in einem jeweiligen über
`VAR receiveport : Pport;`
adressierten Port. Empfangs- und Sendeports sind lokale Datenobjekte der einzelnen Prozesse. Beide Konsumentprozesse laufen durch denselben Code. Nach der Produktion von jeweils 9 Nachrichten erfragt der Produzentprozeß eine Fortsetzungsmitteilung des Ablaufs, um weitere Nachrichten zu senden. Falls der Ablauf nicht fortgesetzt werden soll, werden beide Konsumentprozesse abgebrochen, auch wenn sie noch nicht alle Nachrichten empfangen haben. Welche gesendeten Nachrichten bei den jeweiligen Konsumentprozessen ankommen, ist nicht geregelt.

```
PROGRAM TestIPK;

USES Crt,
     Multitsk, MultiWin, IPK;

CONST produzent_prio = 100;
      konsument_prio = 100;

VAR produzent_task   : TaskPtr;
    konsument_task_1 : TaskPtr;
    konsument_task_2 : TaskPtr;
```

```
{$F+}
PROCEDURE produzent (zeiger : Pointer);

  VAR idx       : INTEGER;
      jdx       : INTEGER;
      WinHandle : INTEGER;
      txt       : STRING;
      message   : Tmessage;
      sendport  : Pport;
      antwort   : CHAR;

  CONST testmessage = 'Nachricht Nr. ';
        kanalID     = 4096;

  BEGIN { produzent }
    WinHandle := TaskWinOpen( 30, 0, 79, 11, einfacher_Rahmen,
                              ' Produzent' );
    sendport  := New (Pport, connect (kanalID));

    idx := 0;
    jdx := 0;

    WHILE idx <= 8 DO
      BEGIN
        Str (10*jdx + idx, txt);
        Writeln (' ---> Nachricht Nr. ' + txt + ' senden');

        message := testmessage + txt;
        sendport^.send (message);

        Inc (idx);
        IF idx = 9 THEN BEGIN
                          Inc (jdx);
                          TaskWinSetCursor;
                          Writeln ('Weiter? (J/N)');
                          Readln (antwort);
                          IF UpCase (antwort) = 'J'
                          THEN idx := 0
                          ELSE BEGIN
                                 MTKillTask (konsument_task_1);
                                 MTKillTask (konsument_task_2);
                               END;
                        END;
      END;

    Dispose (sendport, disconnect);
    TaskWinClose (TRUE);

  END   { produzent };
{$F-}
```

```
{$F+}
PROCEDURE konsument (zeiger : Pointer);

  VAR txt          : STRING;
      WinHandle    : INTEGER;
      receiveport  : Pport;
      message      : Tmessage;

  CONST kanalID = 4096;

  BEGIN { konsument }
    IF INTEGER(zeiger) = 1
    THEN WinHandle := TaskWinOpen( 1, 12, 28, 24,
                                   doppelter_Rahmen,
                                   ' Konsument 1' )
    ELSE WinHandle := TaskWinOpen( 30, 12, 79, 24,
                                   doppelter_Rahmen,
                                   ' Konsument 2' );

    receiveport := New (Pport, connect (kanalID));

    WHILE TRUE DO
      BEGIN
        receiveport^.receive (message);
        Writeln (message);
        Delay (500);
      END;

    Dispose (receiveport, disconnect);
    TaskWinClose (TRUE);

  END   { konsument };
{$F-}


BEGIN { TestIPK }
  TaskWinClrScr;

  produzent_task   := MTCreateTask (produzent, produzent_prio,
                                    5000, NIL);
  konsument_task_1 := MTCreateTask (konsument, konsument_prio,
                                    5000, Ptr(0, 1));
  konsument_task_2 := MTCreateTask (konsument, konsument_prio,
                                    5000, Ptr(0, 2));

  MTStart (10);

END   { TestIPK }.
```

Abbildung 11.5.2-1: Ein Beispiel zum Nachrichtenaustausch

12 Weiterführende Konzepte

Kapitel 11 stellt mit dem implementierten Prozeßmodell einige wichtige Funktionen eines Betriebssystemkerns vor. In einem realen Betriebssystemkern findet man natürlich noch weitere wesentliche Funktionsbereiche wie das Adreßraummanagement, Zeitüberwachungskomponenten, die Geräteverwaltung (Treiber) usw. (Abbildung 11-1). Legt man das Konzept eines Client/Server-Betriebssystems (Abbildung 11-3) zugrunde, so können ganze Funktionsbereiche, z.B. das Dateimanagement, in eigene (Anwender-) Prozesse verlagert werden. Ein wichtiges Hilfsmittel, das der Betriebssystemunterstützung bedarf, ist hierbei, auch in Hinblick auf Client-Server-Architekturen, der Prozedurfernaufruf, dessen Prinzip im folgenden überblicksmäßig beschrieben wird; weitere Details sind z.B. in [TAN] zu finden. Des weiteren wird eine spezielle Möglichkeit vorgestellt, Synchronisationshilfsmittel direkt in die verwendete Programmiersprache einzubauen, ohne diese syntaktisch und semantisch erweitern zu müssen. Es handelt sich hierbei um das Monitorkonzept, das man in einigen Programmiersprachen als Sprachkonzept direkt findet und das in der Realisierung auf Betriebssystemdienste der Synchronisation abgebildet wird.

12.1 Client-Server-Architektur

In der **Client-Server-Architektur** (vgl. [COM]) sind Rechner über ein Kommunikationsnetz lose gekoppelt, d.h. es ist kein gemeinsamer Speicher vorhanden. Eine Benutzeranwendung wird aufgeteilt in zwei funktionale Teile:

- Ein benutzernaher Teil (z.B. Bildschirmpräsentation) läuft auf dem Endsystem des Benutzers, dem **Client**-Rechner, als **Client-Prozeß** (anstoßender Prozeß). Der Client-Prozeß fordert Dienste von einem auf einem entfernten System laufenden Server-Prozeß, z.B. die Ergebnisse von Datenbankabfragen und -änderungen. Verschiedene Client-Prozesse, die denselben Server-Prozeß benutzen, können die Ergebnisse in unterschiedlicher Weise benutzerindividuell aufbereiten.

- Ein gemeinsam von allen Benutzern genutzter Dienst läuft auf dem **Server**-Rechner als **Server-Prozeß** (reagierender Prozeß, z.B. Anwendungsdienste). Der Zugriff ist netztransparent, d.h. der Benutzer benötigt keine Kenntnisse darüber, wo der Server-Prozeß im Netz lokalisiert ist. Der Server-Prozeß wartet passiv auf die Verbindungsaufnahme durch einen (entfernten) Client und führt dann den dedizierten Dienst aus.

Der Server-Prozeß ist i.a. ein nichtterminierender Prozeß, der auf den Anstoß durch einen Client-Prozeß wartet und meist mehrere Client-Prozesse bedient. Die Dienste des Serverprozesses werden in Form von Prozeduren zur Verfügung gestellt, die per **Prozedurfernaufruf (Remote Procedure Call, RPC)**[46] (Abbildung 12.1-1) aufgerufen werden.

[46] Vgl. auch Schill, A.: Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick, Teil 1, Informatik-Spektrum, 15:1992, S.79-87.

Ein RPC unterscheidet sich im Aufruf aus Anwendersicht nicht von einem "normalen" Prozeduraufruf, jedoch im Ablauf, da das **Ziel des RPC nicht im eigenen Adreßraum** lokalisiert ist. Spezielle Probleme können dadurch entstehen, daß während des RPC-Ablaufs komplexe Datenkommunikationsvorgänge stattfinden, in denen Protokolle zu beachten sind, die im wesentlichen dafür sorgen sollen, daß die Kommunikationsvorgänge für den Aufrufer sicher und transparent ablaufen. Dabei kann es zu Zeitverzögerungen und Datenverlusten kommen. Im folgenden Kapitel 12.1.1 werden der Ablauf eines RPCs beschrieben und einige Probleme des RPCs diskutiert.

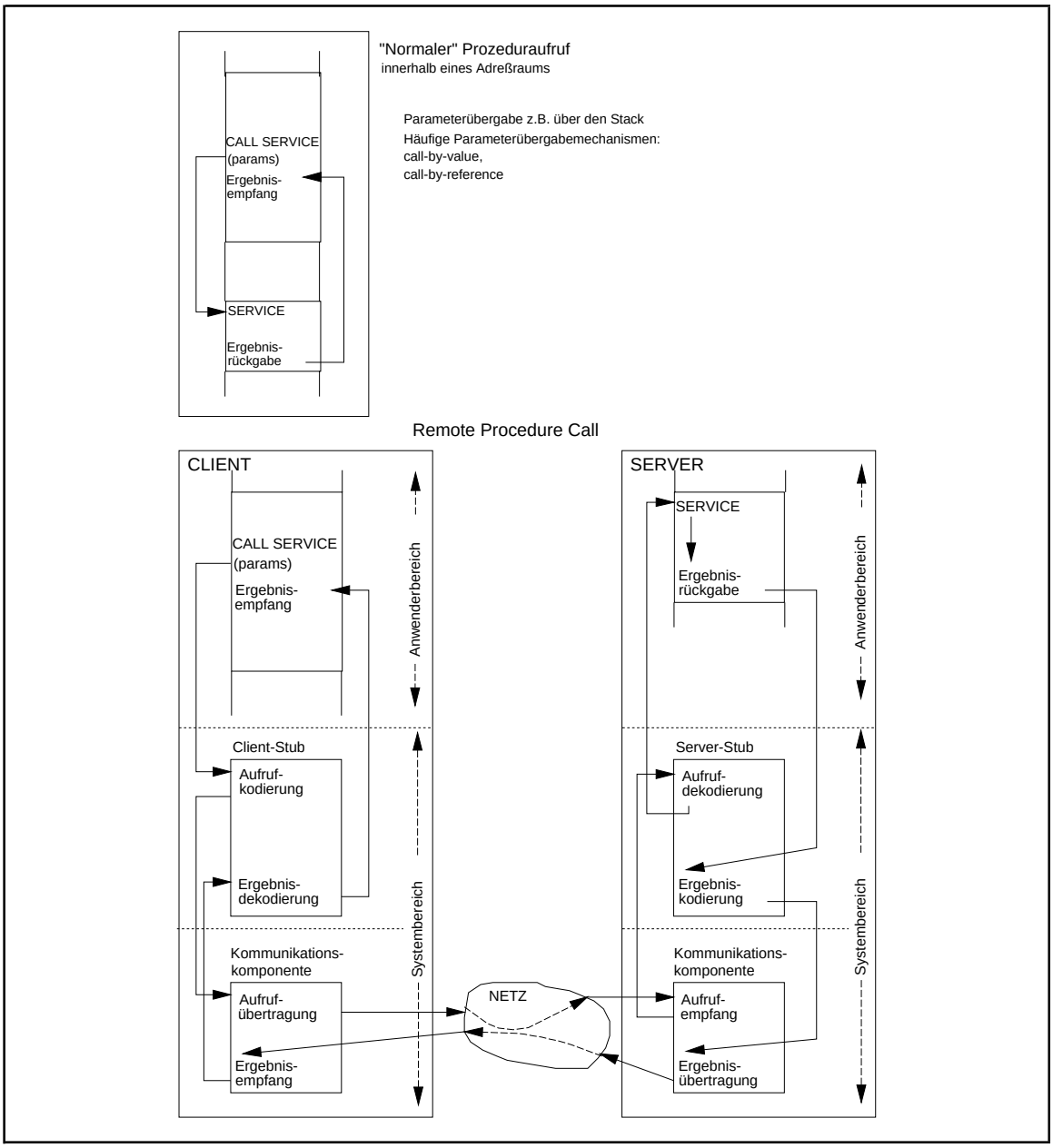


Abbildung 12.1-1: Remote Procedure Call

12.1.1 Remote Procedure Call

Ein Anwendungsprogramm ruft eine Prozedur auf, deren Implementierung in einem nichtlokalen Server vorliegt. Diese Tatsache ist **für das Anwendungsprogramm transparent**. Der Prozeduraufruf wird in den Aufruf einer lokalen Systemprozedur innerhalb des sogenannten **Client-Stubs** transformiert. Dort werden im wesentlichen folgende Aktivitäten abgewickelt:

- **Identifizierung des Servers**, der den gewünschten Dienst anbietet: Diese Information steht im Client-Stub entweder direkt zur Verfügung (logischer Servername, Serveradresse innerhalb des Netzes) und wurde **statisch** bei Übersetzung des Stub-Programms an den entsprechenden Aufruf gebunden. Oder sie wird erst mit Hilfe eines **Directory-Dienstes** oder durch einen **globalen Broadcast-Aufruf** aus der Beschreibung der Prozedurschnittstelle zum Aufrufzeitpunkt ermittelt (**dynamisches Binden**, der Broadcast-Aufruf versucht das Ziel des RPC zu ermitteln, indem sich ein angesprochener Server-Prozeß "meldet"). Diese zweite Möglichkeit ist natürlich bezüglich Änderungen wesentlich flexibler, erhöht aber das Kommunikationsaufkommen auf dem zugrundeliegenden Netz.

- **Transformation der Parameter** in ein vereinbartes Übertragungsformat und **Übergabe des Aufrufs** an die Übertragungskomponente: Eventuell müssen interne Datenformate der Aktualparameter umkodiert oder die Reihenfolge der Aktualparameter umgestellt und in für die Datenübertragung geeignete Datenpakete transformiert werden.

- **Entgegennahme** (von der Kommunikationskomponente) und **Dekodierung der Ergebnisse** des Prozeduraufrufs, sowie **Rückgabe der Ergebnisse** im gewünschten Format an den Aufrufer. Dabei müssen eine Reihe von **möglichen Fehlersituationen** wie Nachrichtenverlust, Ausfall des Servers, Absturz des Anwendungsprozesses, Kontrolle von Aufrufduplikaten, Timeout-Behandlung, verwaiste Aufrufe abgefangen werden (siehe Kapitel 12.1.2).

Auf der Serverseite steht ein entsprechender **Server-Stub** bereit, der den RPC von der Kommunikationskomponente des Servers entgegennimmt, transformiert, an den entsprechenden Service weiterleitet, die Ergebnisse des Prozeduraufrufs kodiert und mit Hilfe der Kommunikationskomponente an den Client zurückschickt.

Die **Kommunikationskomponenten** im Client und im Server sind für das Routing durch das Netz sowie für Quittungen und Übertragungswiederholung bei Übertragungsfehlern der aus dem RPC generierten Nachrichten zuständig.

Während der Ausführung des entfernten Aufrufs ist das Anwendungsprogramm blokkiert.

Bei der Ausführung eines RPCs können eine Reihe von **Fehlersituationen** auftreten, die hauptsächlich dadurch entstehen, daß der Server-Rechner, der Client-Rechner oder das Übertragungssystem ausfallen, während die Bearbeitung eines RPCs noch nicht beendet ist. Geht man davon aus, daß die Kommunikationskomponente dafür sorgt, daß Nachrichten zuverlässig durch das Netz transportiert werden, treten nur die Probleme Ausfall des Server-Rechners und Ausfall des Client-Rechners auf:

- Bei einem **Ausfall des Server-Rechners** kann der Client-Stub auf verschiedene Arten reagieren:
  -- Er wartet auf eine Antwort, die niemals ankommt. Diese Situation entspricht der Ausführung eines "normalen" Prozeduraufrufs aus Aufrufersicht, wenn die Ausführung der Prozedur zu keinem Rücksprung zum Aufrufer führt (z.B. unendliche Schleife, Programmabsturz).
  -- Er erwartet die Antwort innerhalb einer definierten Zeitspanne (Timeout); kommt die Antwort nicht, wird der Client über diese Situation mit einem entsprechenden Returncode benachrichtigt. Diese Variante entspricht der Aufruf von zeitüberwachten Betriebssystemdiensten.
  -- Er erwartet die Antwort innerhalb einer definierten Zeitspanne (Timeout); kommt die Antwort nicht, wird die RPC-Anforderung an den Server-Rechner erneut übertragen. Der Client-Stub übernimmt also die Verantwortung für Maßnahmen zur Behandlung dieser Fehlersituation.

- Bei einem **Ausfall des Client-Rechners** läuft der RPC im Server-Rechner ab, ohne daß es einen zugehörigen Client-Prozeß gibt. Ein arbeitender Server, der keinen Client mehr besitzt, wird als **Waise (orphan)** bezeichnet. Spezielle Vorkehrungen können zum Umgang mit Waisen getroffen werden, z.B.
  -- **Ausrottung (extermination)**: Bei Wiederanlauf des Client-Rechners prüft er, ob er vor dem Absturz RPCs in Auftrag gegeben hatte und veranlaßt den entsprechenden Server zum Abbruch der RPC-Ausführung. Damit ein Client-Stub feststellen kann, daß ein RPC gestartet wird, muß er diesen vor seiner Ausführung (beispielsweise auf der Festplatte) protokollieren.
  -- **Kontrolle durch eine Verfallszeit (expiration)**: Der Client setzt dem Server ein definertes Zeitintervall, während dessen er den RPC vollständig bearbeiten muß. Kommt es innerhalb dieses Zeitintervalls nicht zum Bearbeitungsschluß, muß der Server den Client um ein weiteres Zeitintervall bitten. Falls der Client ausgefallen ist oder erneut gestartet wurde, kann diese Situation vom Server erkannt werden, und er stellt die weitere Bearbeitung des RPCs ein. Der Client braucht kein Protokoll zu führen.
  -- **Wiedergeburt (reincarnation)**: Es kommt u.U. in komplexen Netzwerken vor, daß nicht alle Antworten von Waisen vernichtet werden. Daher wird die Zeit in aufeinanderfolgende numerierte **Epochen** eingeteilt. Alle Nachrichten zwischen Client- und Server-Rechner tragen die Nummer der Epoche, in der sie abgeschickt wurden. Gelingt es einem Client-Rechner nach Wiederanlauf nicht, alle Antworten von Waisen zu vernichten, signalisiert er allen angeschlossenen Rechnern den Beginn einer neuen Epoche, worauf diese alle ihre Serverprozesse beenden. Jetzt auftretende Antworten von Waisen tragen die Nummer einer

vergangenen Epoche und können verworfen werden. In einer weniger drakonischen Variante (**gentle reincarnation**) wird von einem Server bei Beginn einer neuen Epoche versucht, den Auftraggeber eines RPCs (Client) zu finden. Nur wenn dieser nicht festgestellt werden kann, wird der Server unterbrochen.

Für RPC-Implementierungen sind verschiedene **Fehlerklassen** (Gütekriterien) üblich, die hier tabellarisch nach aufsteigender Qualitätsgarantie beschrieben werden:

Fehlerklasse	Beschreibung
Maybe	Der RPC wird höchstens einmal durchgeführt. Der Client erhält keine Nachricht über die erfolglose Ausführung des RPCs.
At-Least-Once	Ein RPC wird solange wiederholt, bis er mindestens einmal erfolgreich ausgeführt wurde. Bei korrekter Ausführung im Server-Rechner und anschließendem Verlust des Ergebnisses bedeutet dies, daß ein RPC mehrfach ausgeführt wird und dadurch eventuell unerwünschte Nebeneffekten hervorruft.
Only-Once-Type-1 (At-Most-Once)	Auch bei Nachrichtenverlust wird ein RPC nur einmal ausgeführt. Aufrufduplikate werden erkannt und unterdrückt. Bei einem Rechnerausfall wird nur garantiert, daß der Aufruf höchstens einmal erfolgt; er kann aber auch fehlschlagen.
Only-Once-Type-2 (Exactly-Once)	Es wird garantiert, daß ein RPC auch bei Ausfall und Wiederanlauf des Client- oder Server-Rechners genau einmal ausgeführt wird.

Es gibt eine Reihe von Situationen, die in normalen Prozeduraufrufen nicht vorkommen, aber beim RPC zu speziellen Problemen führen können. So wird beispielsweise ein RPC-Aufruf **aktuelle Parameter** enthalten:

- Bei einem **call-by-value-Parameter** wird der Wert des aktuellen Parameters als Teil der Nachricht zum Server geschickt, so daß keine prinzipiellen Probleme zu erwarten sind.

- Läßt der RPC-Aufruf **call-by-value-result-Parameter** zu, so muß der Ergebniswert, der vom Server zum Client zurückgeschickt wurde, vom Client-Stub in den aktuellen Parameter kopiert werden. Da Client-Stub und Client-Prozeß im selben Adreßraum operieren, sind auch hierbei keine prinzipiellen Probleme zu erwarten sind.

- Bei einem aktuellen **call-by-reference-Parameter** muß sich jeder Zugriff auf den Parameter in der gerufenen (fernen) Prozedur auf den Adreßraum des Clients beziehen. Eventuelle Wertänderungen des Parameters müssen im (aus der Sicht des Servers seinerseits fernen) Adreßraum des Clients sofort ausgeführt werden. Jeder Zugriff in der gerufenen fernen Prozedur im Server kann dazu beispielsweise in einen RPC zurück an den Client umgesetzt werden. Die Realisierung dieses Ablaufprinzips kann in den jeweiligen Stubs eingebettet sein. Eine andere Lösung, die in den meisten Fällen ausreicht (insbesondere dann, wenn die physikalische Größe des aktuellen

Parmeters bekannt ist), besteht darin, zunächst den aktuellen Wert des call-by-reference-Parameters an den Server in den Server-Stub zu übertragen. Alle Operationen des Servers beziehen sich nun auf diese Kopie im Server-Stub. Am Ende des Prozeduraufrufs wird der Wert an den Client zurückgegeben. Insgesamt wird also der call-by-reference-Übergabemodus durch einen call-by-copy/restore-Übergabemodus ersetzt, auch wenn diese funktional nicht äquivalent sind (z.B. bei Prozedurabbruch).

- Ähnliche Probleme wie bei call-by-reference-Parametern gibt es bei aktuellen **Parametern vom Typ `Pointer`** im Client-Prozeß.

- In einigen Programmiersprachen (z.B. PASCAL, C) ist es möglich, aktuelle **Parameter vom Typ `PROCEDURE`** zu übergeben. U.U. wäre es dazu erforderlich, den Code der Prozedur an den Server (in Objekt- oder Sourcecodeform) zu übergeben, so daß derartige Parameter in der Regel nicht erlaubt werden.

Andere Probleme können entstehen, wenn der Client-Stub bei einem festgestellten Ausfall des Server-Rechners die RPC-Anforderung erneut stellt, da nicht alle Aufrufe ohne weiteres wiederholt werden können, ohne daß es Seiteneffekte gibt. So kann das Lesen einer Datei oder die Initialisierung einr Variablen u.U. ohne Seiteneffekte beliebig oft wiederholbar sein (**idempotente Aufrufe**), andere Aufrufe wie das Anhängen von Datensätzen an eine Datei oder die Veränderungen von Sätzen in Datenbanksystemen nicht.

12.1.2 Simulation einer Client-Server-Architektur

In diesem Kapitel wird exemplarisch der Aufbau einer Client-Server-Architektur diskutiert, die die in Kapitel 11 bereitgestellten Hilfsmittel verwendet. Da das Modell keinen RPC zur Verfügung stellt, verwendet die Simulation die IPK-Hilfsmittel (Kapitel 11.5).

In der vorliegenden Client-Server-Anwendung stellt ein Server-Prozeß Dienstleistungen bereit, die von vielen Client-Prozessen angefordert werden können. Der Server ist in der Lage, verschiedene Typen von Dienstleistungen zu erbringen, so daß ein Client-Prozeß jeweils die angeforderte Dienstleistung noch genauer spezifizieren muß. Die Kommunikation zwischen den Client-Prozessen und dem Server-Prozeß erfolgt nach dem in Kapitel 11.5 behandelten Modell der Interprozeßkommunikation: Um die Dienstleistung anzufordern, sendet ein Client-Prozeß eine Anforderungsnachricht über einen „Request-Kanal“ an den Server-Prozeß. Alle Client-Prozesse verwenden denselben Request-Kanal, an den sie sich jeweils zu Beginn ihres Ablaufs über einen Port angeschlossen haben. In die Nachricht für die Dienstleistungsanforderung an den Server-Prozeß hat der anfordernde Client-Prozeß die eindeutige Identifikation eines „Antwort-Kanals“ eingesetzt, über den er die Antwort vom Server erwartet. Jeder Client-Prozeß hat sich zu Beginn über einen Port an diesen eigenen Antwort-Kanal angeschlossen. Außerdem wird die vom Server erwartete Dienstleistung in der Anforderungsnachricht genauer spezifiziert. Abbildung 12.1.2-1 zeigt eine exemplarische Kommunikationsstruktur mit drei Client-Prozessen.

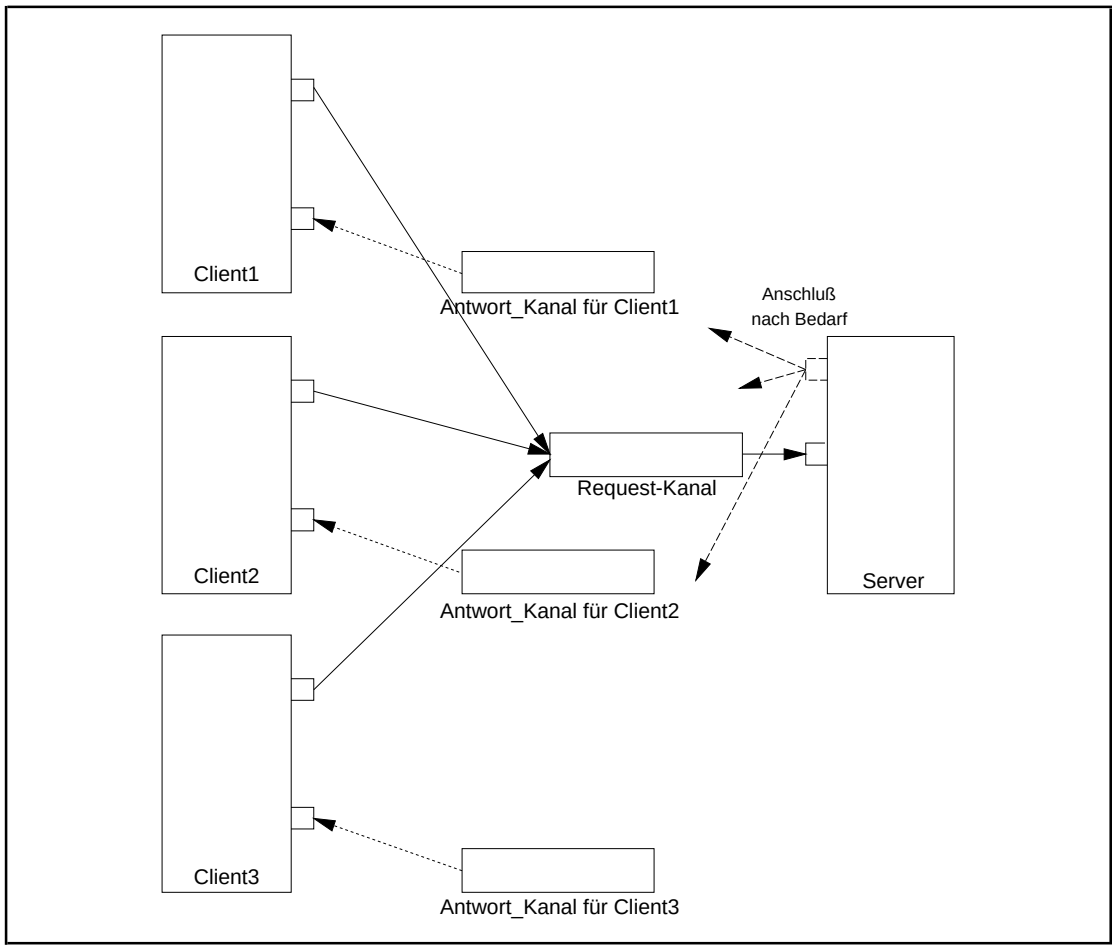


Abbildung 12.1.2-1: Client-Server-Anwendung (Beispiel)

Der Datentyp einer Anfrage, die ein Client-Prozeß an den Server-Prozeß über den Request-Kanal sendet, lautet:

```
TYPE TDienst = (...);   { Aufzählung der verschiedenen vom Server
                          angebotenen Dienste                    }

     Pmessage = ^Tmessage;
     Tmessage = RECORD
                  Antwort_Kanal_ID           = Tname;
                                   { Identifizierung
                                     des Antwortkanals            }
                  Dienst_spezifische_Angaben = TDienst;
                                   { genauere Spezifizierung
                                     des erwarteten Dienstes     }
                END;
```

Der Request-Kanal hat die systemweite Identifizierung

```
CONST Request_Kanal_ID = ...;
```

Im Code, mit dem ein Client-Prozeß eine Dienstleistung anfordert, wird eine Funktion `Antwort_kanal` aufgerufen, die aus der (eindeutigen) Prozeßidentifikation des Client-Prozesses eine systemweit eindeutige Identifikation eines Antwort-Kanals ermittelt. Der Code lautet dann ausschnittsweise:

```
{$F+}
PROCEDURE Client (zeiger : Pointer);

   FUNCTION Antwort_kanal : Tname;
     BEGIN { Antwort_kanal }
       ...
     END   { Antwort_kanal };

VAR Dienstanfrage    : Tmessage;
    Dienstergebnis   : Tmessage;
    request_port     : Pport;
    answer_port      : Pport;
    Antwort_kanal_ID : Tname;

BEGIN { Client }
  Antwort_kanal_ID := Antwort_kanal;
  request_port := New (Pport, connect (Request_Kanal_ID));
  answer_port  := New (Pport, connect (Antwort_Kanal_ID));
  ...
  { Anforderung einer Dienstleistung: }
  Dienstanfrage.Antwort_Kanal_ID           := Antwort_Kanal_ID;
  Dienstanfrage.Dienst_spezifische_Angaben := ...;
  request_port^.send (Dienstanfrage);
  answer_port^.receive (Dienstergebnis);
  ...
  Dispose (answer_port, disconnect);
  Dispose (request_port, disconnect);
END   { Client };
{$F-}
```

In einem ersten Realisierungsvorschlag für den Code des Server-Prozesses läuft dieser durch eine Prozedur `Server_Konzept1`. Hier wartet er auf eine Anfrage über den Request-Kanal. Sobald eine Anfrage eintrifft, wird sie bearbeitet und das Ergebnis der Dienstleistung dem anfragenden Client-Prozeß über den in der Anfrage angegebenen Kanal zurückgesendet. Der Code des Server-Prozesses lautet (hier sind die durch zwei Striche gekennzeichneten Teile als Pseudocode anzusehen):

```
{$F+}
PROCEDURE Server_Konzept1 (ptr : Pointer);

VAR request_port : Pport;
    request      : Tmessage;
    answer_port  : Pport;
    answer       : Tmessage;

BEGIN { Server_Konzept1 }
  request_port := New (Pport, connect (Request_Kanal_ID));

  WHILE TRUE DO
   BEGIN
     request_port^.receive (request);
     -- Bearbeitung der Anfrage gemäß den Angaben in
        request.Dienst_spezifische_Angaben;
     -- Ablage des Ergebnisses in
        answer.Dienst_spezifische_Angaben;
     answer_port:= New (Pport, connect (request.Antwort_Kanal_ID));
     answer_port.send (answer);
     Dispose (answer_port, disconnect);
   END;

  Dispose (request_port, disconnect);
END   { Server_Konzept1 };
{$F-}
```

Die vorstehende Lösung hat den Vorteil der Einfachheit, indem nur die beteiligten Prozesse vom System verwaltet werden. Jedoch müssen Clients, die während der Bearbeitung einer Dienstanfrage durch den Server ihrerseits Dienste nachfragen, warten, bis der Server diese Dienstanfrage beendet hat. Dauert dieser Vorgang lange oder kommen viele Dienstanfragen beim Server an, entstehen für die anfragenden Clients lange Wartezeiten. Es ist daher sinnvoll, den Server lediglich auf Dienstanfragen warten zu lassen und die Bereitstellung des Dienstergebnisses durch jeweils einen neuen Thread erledigen zu lassen. Der folgende Realisierungsvorschlag setzt dieses Konzept in einer Prozedur `Server_Konzept2` um:

Hierbei wartet der Server auf eine Anfrage über den Request-Kanal. Sobald eine Anfrage eintrifft, richtet der Serverprozeß einen neuen Thread ein, übergibt diesem neuen Thread die Anfrage und wartet selbst auf die nächste Anfrage. Der neu eingerichtete Thread läuft durch die Prozedur `process_request`. Darin wird die Anfrage bearbeitet und das Ergebnis der Dienstleistung dem anfragenden Client-Prozeß über den in der Anfrage angegebenen Kanal zurückgesendet. Dann beendet sich der Thread. Der zeitliche Overhead, der für einen eine Dienstleistung anfragenden Client entsteht, ist lediglich die Zeit, die der Server benötigt, um Threads einzurichten, die vorherige Dienstanfragen bearbeiten.

Der Code des Server-Prozesses lauten:

```
{$F+}
PROCEDURE process_request (req : Pointer);

VAR request      : Tmessage;
    answer_port  : Pport;
    answer       : Tmessage;

BEGIN { process_request }
  request := Pmessage(req)^;
  -- Bearbeitung der Anfrage gemäß den Angaben in
     request.Dienst_spezifische_Angaben;
  -- Ablage des Ergebnisses in
     answer.Dienst_spezifische_Angaben;
  answer_port:= New (Pport, connect (request.Antwort_Kanal_ID));
  answer_port.send (answer);
  Dispose (answer_port, disconnect);
  Dispose (Pmessage(req));
END   { process_request };
{$F-}
```

```
{$F+}
PROCEDURE Server_Konzept2 (ptr : Pointer);

VAR request_port : Pport;
    req_ptr      : Pmessage;

BEGIN { Server_Konzept2 }
  request_port := New (Pport, connect (Request_Kanal_ID));

  WHILE TRUE DO
   BEGIN
     New (req_ptr);
     request_port^.receive (req_ptr^);
     MTCreateTask (process_request, AktTask^.Prio - 1, 2000,
                   req_ptr);
   END;

   Dispose (request_port, disconnect);
END   { Server_Konzept2 };
{$F-}
```

12.2 Das Monitorkonzept

Gerade in der systemnahen Programmierung liegt es nahe, höhere Programmiersprachen einzusetzen, die **Synchronisationskonzepte als Sprachkonstrukte** enthalten. Derartige Programmiersprachen sind definiert, z.B. Concurrent Pascal, Modula (Monitorkonzept) und Ada (Rendezvouskonzept). Da die bisher behandelte Programmiersprache Pascal (so wie auch C oder C++) entsprechende Sprachelemente nicht enthält, wird zunächst ein allgemeines Konzept, das **Monitorkonzept**, als "fiktive" Spracherweiterung von PASCAL beschrieben[47]. Anschließend wird gezeigt, wie diese mit Hilfe der in Kapitel 11 behandelten Synchronisationselemente simuliert werden kann.

Ein **Monitor** ähnelt einer "normalen" PASCAL-Unit. Er enthält als Besonderheit das identifizierende Schlüsselwort `MONITOR` (anstelle von `UNIT`) und Prozeduren vom (fiktiven) Typ **`ENTRY PROCEDURE`**, deren Deklaration nach außen über den `INTERFACE`-Teil anderen Programmen und Units bekannt gegeben werden und von dort aufgerufen werden können. Zusätzlich gibt es einen Objekttyp **`CONDITION`** mit den Methoden **`WAIT`**, **`SIGNAL`** und **`QUEUE`**:

```
TYPE CONDITION  = OBJECT
                    PROCEDURE WAIT;
                    PROCEDURE SIGNAL;
                    FUNCTION QUEUE : BOOLEAN;
                  END;
```

Die Methoden `WAIT`, `SIGNAL` und `QUEUE` werden unten erläutert. Auf die Angabe von Konstruktor und Destruktor wird zunächst verzichtet. Sie werden in der das Kapitel abschließenden Simulation beschriebenen. Ist `cv` ein Datenobjekt vom Typ `CONDITION`, so sagt man, ein Prozeß P **wartet auf die Bedingung** `cv`, wenn dessen TCB-Adresse in der zu `cv` gehörenden Warteschlange eingehängt ist und er sich im Zustand WARTEND befindet.

---

[47] Vgl. auch [H/H] und das Themenheft ACM Computing Surveys, 27:1, 1995.

Im folgenden werden die Bezeichner* `MONITOR`, `CONDITION` *(einschließlich der zugehörigen Methoden) und* `ENTRY PROCEDURE` *wie Schlüsselwörter der Sprache PASCAL eingesetzt.

Abbildung 12.2-1 zeigt den Aufbau eines Monitors.

Der Code des **Initialisierungsteils** des Monitors wird ausgeführt, wenn der Monitor wie eine Unit von einem Programm bzw. einer Unit angebunden wird. Er dient dazu, die internen Datenobjekte des Monitors mit Anfangswerten zu versehen. Der Initialisierungsteil kann auch leer sein.

Wird eine `ENTRY PROCEDURE` von außen aufgerufen, so gilt der Monitor als **belegt**. Ein derartiger Aufruf blockiert den Aufruf aller `ENTRY PROCEDURE`s für alle anderen Prozesse solange, bis der Monitor wieder freigegeben wird. Eine **Freigabe** des Monitors erfolgt, wenn der belegende Prozeß aus der `ENTRY PROCEDURE` zurückspringt oder aufgrund einer `WAIT`-Operation in den Zustand WARTEND gerät.

Der Methodenaufruf

```
cv.WAIT
```

durch einen Prozeß Q bewirkt, daß dessen TCB-Adresse in die zu `cv` gehörende Warteschlange eingehängt wird und Q in den Zustand WARTEND kommt und den Monitor dadurch freigibt. Sobald ein anderer Prozeß P die Operation

```
cv.SIGNAL
```

absetzt, wird ein Prozeß Q, der vorher `cv.WAIT` ausgeführt hat und wartet, aus der zu `cv` gehörenden Schlange ausgehängt und fortgesetzt, wobei P solange unterbrochen wird, bis Q entweder den Monitor verläßt oder selbst eine weitere `WAIT`-Operation im Monitor ausführt. Dann wird P unmittelbar fortgesetzt (Q "ersetzt" also den signalisierenden Prozeß P im Monitor). Gibt es keinen Prozeß, der auf die Bedingung `cv` wartet, so hat `cv.SIGNAL` keine Wirkung. Die Operation

```
cv.QUEUE
```

liefert den Wert `TRUE`, falls Prozesse auf die Bedingung `cv` warten; andernfalls liefert die Operation den Wert `FALSE`.

```
MONITOR name;        { Identifizierung des Monitors          }
{ ------------------------------------------------------- }
INTERFACE

ENTRY PROCEDURE proc_1 (parameterlist_1);

 ...

ENTRY PROCEDURE proc_m (parameterlist_m);
{ ------------------------------------------------------- }
IMPLEMENTATION

VAR local_data_definition;
                  { Deklaration lokaler Variablen mit
                    "normalen" Datentypen                 }

VAR cond_var_1, ..., cond_var_n : CONDITION;
                  { Deklaration lokaler Variablen mit
                    Datentyp CONDITION                    }

{ Deklaration "normaler" für den Monitor lokaler Prozeduren,
  die von außen nicht aufrufbar sind                      }

PROCEDURE local_proc_1 (parameterlist_l_1);
  VAR data_definition_local_proc_1
                        { Deklaration lokaler Variablen für
                          local_proc_1 mit "normalen"
                          Datentypen                      }
  BEGIN { local_proc_1 }
   ...
  END   { local_proc_1 };

...

PROCEDURE local_proc_k (parameterlist_l_k);
  VAR data_definition_local_proc_k;
                        { Deklaration lokaler Variablen für
                          local_proc_k mit "normalen"
                          Datentypen                      }
  BEGIN { local_proc_k }
   ...
  END   { local_proc_k };

{ Implementierungsdetails der ENTRY PROCEDUREs }

ENTRY PROCEDURE proc_1 (parameterlist_1);
  VAR local_data_definition_proc_1;
                        { Deklaration lokaler Variablen für
                          proc_1 mit "normalen" Datentypen  }
  BEGIN { proc_1 }
   ...
  END   { proc_1 };

...

ENTRY PROCEDURE proc_m (parameterlist_m);
  VAR local_data_definition_proc_m;
                        { Deklaration lokaler Variablen für
                          proc_m mit "normalen" Datentypen  }
  BEGIN { proc_m }
   ...
  END   { proc_m };
{ ------------------------------------------------------- }
BEGIN { Initialisierungsteil des Monitors }
  ...
END   { MONITOR name }.
```

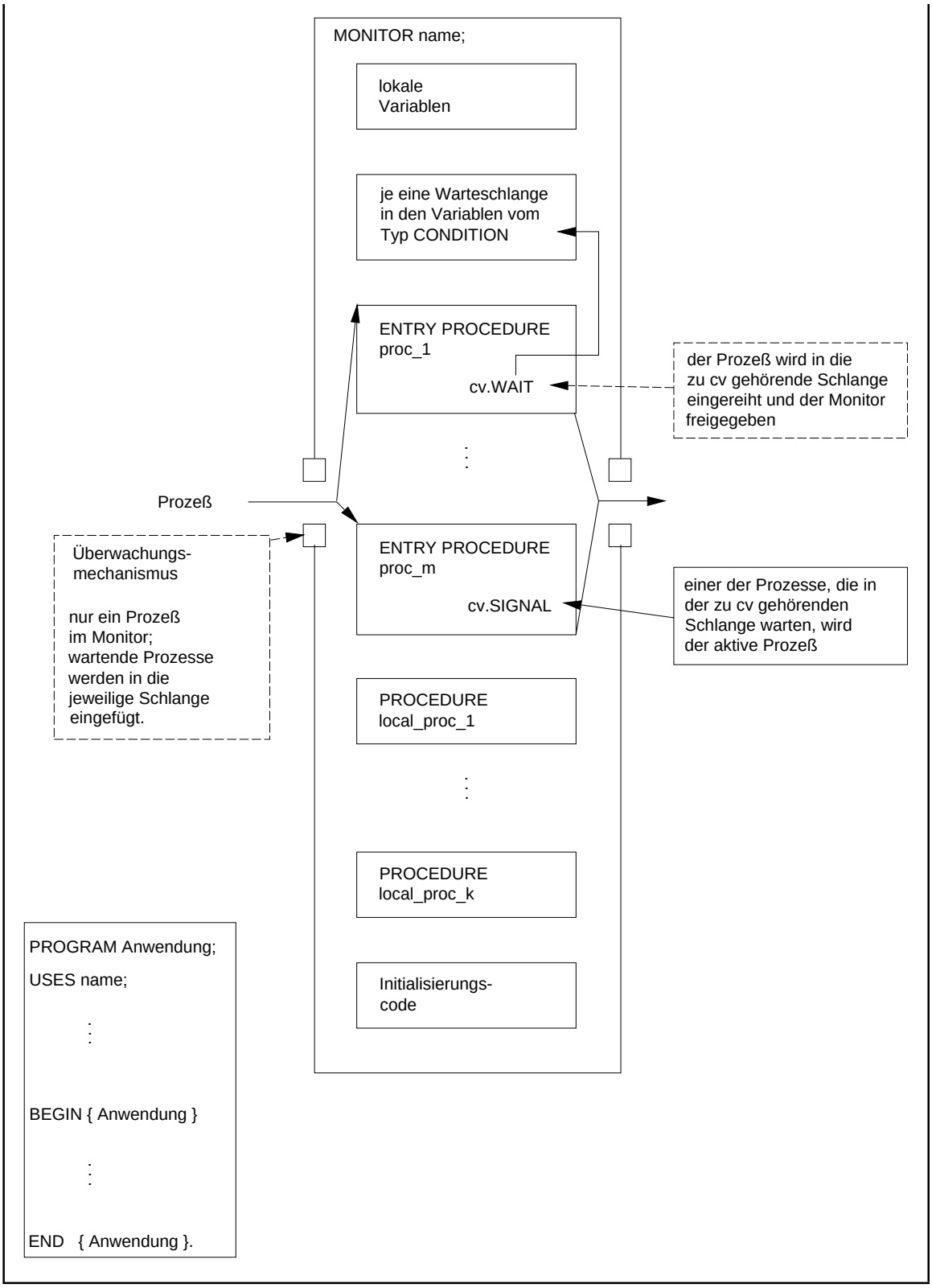


Abbildung 12.2-1: Monitor

Ein Prozeß kann einen so definierten Monitor mit Bezeichner `name` in seinen Programmcode mittels

```
USES name;
```

einbinden. Zu beachten ist, daß zum einen in einem Monitor keine globalen Variablen, die außerhalb des Monitors deklariert sind und auf die durch eine eventuelle `USES`-Anweisung innerhalb des Monitors zugegriffen wird, verwendet werden, um Inkosistenzen durch unkoordinierten Zugriff zu vermeiden. Zum anderen sind die lokalen Variablen innerhalb eines Monitors statisch und werden zwischen allen Prozessen geteilt, die den Monitor betreten. Ihre Werte bleiben erhalten, wenn ein Prozeß den Monitor freigibt.

Zusätzlich ergibt sich ein Problem, wenn geschachtelte Monitoraufrufe zugelassen werden, d.h. wenn innerhalb der Ausführung einer `ENTRY PROCEDURE` die `ENTRY PROCEDURE` eines weiteren Monitors aufgerufen wird. Bei einer Freigabe des inneren Monitors aufgrund eines `SIGNAL`-Aufrufs müssen dann auch die um den Monitor herum geschachtelten Monitore freigegeben werden, um potentielle Deadlocks zu verhindern; wird der signalisierende Prozeß im Monitor später fortgesetzt, müssen die Sperren in der Aufrufhierarchie wieder gesetzt werden.

Exemplarisch wird gezeigt, wie sich das Leser/Schreiber-Problem mit Vorrangregel (i) (siehe Kapitel 11.4.1 und [M/O]) mit Hilfe eines Monitors realisieren läßt. Es wird wieder das Prozeßmodell aus Kapitel 11 vorausgesetzt und die Realisierung in den dort beschriebenen Rahmen eingebettet. Aus Anwendersicht ist die Verwendung eines Monitors sehr einfach, da er alle Synchronisationsdetails in seinen impliziten Mechanismen verbirgt.

Der Datenbereich, auf dem alle beteiligten Prozesse zugreifen, wird als ein Objekt mit Objekttyp `Tdatenbereich` (im `INTERFACE-Teil` einer Unit `Datenbereich`) vereinbart, dessen Methoden hier jedoch nur benannt und nicht weiter ausgeführt werden:

```
UNIT Datenbereich;

INTERFACE

TYPE Pdatenbereich = ^Tdatenbereich;
     Tdatenbereich = OBJECT
                       PRIVATE
                         daten : ...;  { Datenbereich         }
                       PUBLIC
                         CONSTRUCTOR init;
                         PROCEDURE lesen (...);
                         PROCEDURE schreiben (...);
                         DESTRUCTOR done;
                     END;

IMPLEMENTATION
...
END.
```

Der Monitor `ls_monitor` in Abbildung 12.2-2 kontrolliert den Zugriff auf den Datenbereich, der selbst im Anwendungsprogramm mit Bezeichner `leser_schreiber_i_mit_monitor` vereinbart ist. Dort wird auch die `FUNCTION check_abbruch` deklariert, die den Wert `TRUE` liefert, wenn der weitere Ablauf nicht fortgesetzt werden soll.

```
MONITOR ls_monitor;

{ ------------------------------------------------------------- }
INTERFACE

ENTRY PROCEDURE startread;

ENTRY PROCEDURE endread;

ENTRY PROCEDURE startwrite;

ENTRY PROCEDURE endwrite;

PROCEDURE close_monitor;

{ ------------------------------------------------------------- }
IMPLEMENTATION

VAR readcount  : INTEGER;
    busy       : BOOLEAN;

    ok_to_read : CONDITION;
    ok_to_write: CONDITION;


ENTRY PROCEDURE startread;

BEGIN { startread }
  IF busy THEN ok_to_read.WAIT;
  readcount := readcount + 1;
  IF ok_to_read.QUEUE THEN ok_to_read.SIGNAL
END   { startread };
ENTRY PROCEDURE endread;

BEGIN { endread }
  readcount := readcount - 1;
  IF readcount = 0 THEN ok_to_write.SIGNAL
END   { endread };

ENTRY PROCEDURE startwrite;

BEGIN { startwrite }
  IF (readcount <> 0) OR busy THEN ok_to_write.WAIT;
  busy := TRUE
END   { startwrite };

ENTRY PROCEDURE endwrite;

BEGIN { endwrite }
  busy := FALSE;
  IF ok_to_read.QUEUE THEN ok_to_read.SIGNAL
                      ELSE ok_to_write.SIGNAL
END   { endwrite };

PROCEDURE close_monitor;

BEGIN { close_monitor }
  ok_to_read.done;
  ok_to_write.done;
END   { close_monitor };
```

```
{ ------------------------------------------------------------- }
{ Initialisierung }
BEGIN { ls_monitor }
  ok_to_read.init;
  ok_to_write.init;

  readcount := 0;
  busy      := FALSE;
END   { ls_monitor }.



PROGRAM leser_schreiber_i_mit_monitor;

USES Datenbereich, ls_monitor;

CONST anz_leser     = ...;  { Anzahl der Leserprozesse       }
      anz_schreiber = ...;  { Anzahl der Schreiberprozesse   }

VAR idx    : INTEGER;
    puffer : Pdatenbereich;


FUNCTION check_abbruch : BOOLEAN;

BEGIN { check_abbruch }
  ...
END   { check_abbruch };
{$F+}
PROCEDURE leser (zeiger : Pointer);

PROCEDURE verarbeiten (...); { ... der gelesenen Daten }

BEGIN { verarbeiten }
  ...
END   { verarbeiten };

BEGIN { leser }
  REPEAT
    BEGIN
      startread;
      puffer^.lesen(...);
      endread;
      verarbeiten (...);  { Verarbeitung: unkritischer Abschnitt }
    END
  UNTIL check_abbruch
END   { leser };
{$F-}
```

```
{$F+}
PROCEDURE schreiber (zeiger : Pointer);

PROCEDURE erzeugen (...);    { von Daten }

BEGIN { erzeugen }
  ...
END   { erzeugen };

BEGIN { schreiber }
  REPEAT
    BEGIN
      erzeugen (...);     { Verarbeitung: unkritischer Abschnitt }
      startwrite;
      puffer^.schreiben(...);
      endwrite;
    END
  UNTIL check_abbruch
END   { schreiber };
{$F-}
{ ------------------------------------------------------------- }

BEGIN { leser_schreiber_i_mit_monitor }

  New (puffer, init);

  { Einrichten aller Leser-Prozesse, jeweils mit PROCEDURE leser }
  FOR idx := 1 TO anz_leser DO
    MTCreateTask (leser, ...);
  { Einrichten aller Schreiber-Prozesse, jeweils mit PROCEDURE
    schreiber                                                   }
  FOR idx := 1 TO anz_schreiber DO
    MTCreateTask (schreiber, ...);

  MTStart (...);

  Dispose (puffer, done);
  close_monitor;
END   { leser_schreiber_i_mit_monitor }.
```

Abbildung 12.2-2: Schreiber/Leser-Problem

Das Monitorkonzept ermöglicht eine einfache Handhabung der Synchronisation, stellt aber bezüglich der Synchronisationsfunktionalität keine Erweiterung dar. Mit Hilfe eines Monitors lassen sich Semaphore leicht funktional nachbilden, und ein Monitor ist mit Semaphoren und normalen Prozeduren und Variablen simulierbar. Im folgenden wird diese zweite Aussage in einem Designvorschlag für das in Kapitel 11 beschriebene Prozeßmodell in der dort verwendeten Terminologie erläutert (vgl. auch [H/H]). Die Simulation wird an dem vereinfachten "Monitor-Gerüst" in Abbildung 12.2-3 gezeigt.

```
MONITOR name;

INTERFACE

ENTRY PROCEDURE proc_1 (parameterlist_1);

 ...

ENTRY PROCEDURE proc_m (parameterlist_m);

IMPLEMENTATION

...

VAR cv : CONDITION;

...

ENTRY PROCEDURE proc_1 (parameterlist_1);
  ...
  BEGIN { proc_1 }
    ...
    cv.WAIT;
    ...
  END   { proc_1 };

...

ENTRY PROCEDURE proc_m (parameterlist_m);
  ...
  BEGIN { proc_m }
    ...
    cv.SIGNAL;
    ...
  END   { proc_m };

...

BEGIN { Initialisierungsteil }
  ...
END   { MONITOR name }.
```

Abbildung 12.2-3: Simulation eines Monitors durch Semaphore

Die Schlüsselwörter `MONITOR` und `ENTRY PROCEDURE` müssen durch Pascal-Schlüsselwörter und geeigneten -Code ersetzt und die Implementation der Methoden für den Objekttyp `CONDITION` angegeben werden, und zwar unter Beachtung der Charakteristika eines Monitors:

- Die Exklusivität der `ENTRY PROCEDURE`s muß gewährleistet werden

- Durch Ausführung einer Operation `cv.WAIT` muß der Monitor sofort freigegeben werden.

- Die Operation `cv.SIGNAL` unterbricht den ausführenden Prozeß und setzt einen Prozeß fort, der zuvor durch `cv.WAIT` in den Wartezustand geraten ist (falls es einen derartigen Prozeß überhaupt gibt). Eine eigene Prozeßfortsetzung erfolgt, sobald dieser Fortsetzungsprozeß den Monitor freigibt. In diesem Fall hat die eigene Prozeßfortsetzung

Priorität über alle Prozesse, die eventuell den Monitor betreten wollen, und prinzipiell auch Priorität über alle Prozesse im System, die inzwischen in den Zustand BEREIT gekommen sind und höhere Priorität besitzen[48].

Zur weiteren Vereinfachung der Simulation werden die Annahmen getroffen, daß Prozeßprioritäten nicht berücksichtigt zu werden brauchen und daß keine geschachtelten Monitoraufrufe vorkommen. Bei geschachtelten Monitoraufrufen müssen die von einem Prozeß betretenen und noch nicht wieder verlassenen Monitore in einem Aufrufstack protokolliert werden, damit bei Freigabe auch alle äußeren Monitore zischenzeitlich freigegeben und später wieder gesperrt werden können.

Das Schlüsselwort `MONITOR` wird durch das Schlüsselwort `UNIT` ersetzt, d.h. der Monitor wird zu einer normalen Pascal-Unit.

Jeder Monitor erhält eine mit 0 initialisierte lokale Variable

```
CONST dringend : INTEGER = 0;
```

die dann inkrementiert wird, wenn ein Prozeß einen anderen Prozeß aufgrund einer `SIGNAL`-Operation angestößt, dabei selbst unterbrochen wird und fortgesetzt werden muß, sobald der angestoßene Prozeß den Monitor freigibt. Dabei die Variable wieder dekrementiert. Außerdem erhält der Monitor zwei durch

```
VAR sem_dringend : PSemaphor;
    mutual       : PSemaphor;
```

adressierte Semaphore. Das erste Semaphor regelt die vorrangige Fortsetzung eines aufgrund einer `SIGNAL`-Operation im Monitor unterbrochenen Prozesses, das zweite Semaphor gewährleistet die Exklusivität der `ENTRY PROCEDURE`s. Beide Semaphore werden im Initialisierungsteil des Monitors initialisiert.

Die Deklaration jeder `ENTRY PROCEDURE` wird durch eine normale `PROCEDURE`-Deklaration ersetzt. Außerdem werden in den zugehörigen Prozedurrumpf am Anfang und Ende Anweisungen eingefügt, um die Exklusivität zu sichern.

Die Simulation des Objekttyps `CONDITION` wird durch eine entsprechende Deklaration `TCondition` in einer eigenen Unit mit Bezeichner `Condition` präzisiert, die in den Monitor eingebunden wird. Jedes Datenobjekt vom Typ `TCondition` enthält in der Komponente `sem` ein eigenes Semaphor, in das Prozesse eingereiht werden, die wegen einer `WAIT`-Operation am Datenobjekt warten. Ein Zähler in der Komponente `count` zählt die Anzahl dieser Prozesse (dieser Zähler ist notwendig, da keine Semaphormethode definiert ist, die feststellt, wieviele Prozesse vor einem Semphor warten). Die weiteren Komponenten eines Datenobjekts vom Typ `TCondition` werden im Konstruktor mit den Adressen der

[48] In [M/O] werden Designalternativen in Bezug auf dieses Prioritätsproblem diskutiert.

oben beschriebenen Variablen `dringend` und der beiden zusätzlichen Semaphore initialisiert, die zu dem Monitor gehören, der das Datenobjekt enthält. Die Unit `Condition` lautet:

```
UNIT Condition;

INTERFACE

USES Semaphor;

TYPE PINTEGER = ^INTEGER;

     PCondition = ^TCondition;
     TCondition =
       OBJECT
         PRIVATE
            sem          : TSemaphor; { wegen WAIT wartende Prozesse}
            count        : INTEGER;   { Anzahl wartender Prozesse   }
             { Adressen der Synchronisationshilfsmittel
               des umfassenden Monitors:                            }
            dringend     : PINTEGER;
            sem_dringend : PSemaphor;
            mutual       : PSemaphor;
          PUBLIC
            CONSTRUCTOR init (a_d : PINTEGER;
                              a_s : PSemaphor;
                              a_m : PSemaphor);
            PROCEDURE WAIT;
            PROCEDURE SIGNAL;
            FUNCTION QUEUE : BOOLEAN;
            DESTRUCTOR done; VIRTUAL;
        END;


IMPLEMENTATION

USES Multitsk;

CONSTRUCTOR TCondition.init (a_d : PINTEGER;
                             a_s : PSemaphor;
                             a_m : PSemaphor);

BEGIN { TCondition.init }
  MTBlock;
  dringend     := a_d;
  sem_dringend := a_s;
  mutual       := a_m;
  sem.init (0);
  count := 0;
  MTContinue;
END   { TCondition.init };

PROCEDURE TCondition.WAIT;

BEGIN { TCondition.WAIT }
  MTBlock;
  IF dringend^ > 0
  THEN { ein Prozeß, der einen anderen Prozeß durch eine WAIT-
         Operation angestoßen hat und dadurch selbst unterbrochen
         wurde, muß fortgesetzt werden                              }
       sem_dringend^.V
  ELSE { den Monitor für einen eventuell vor dem wartenden Prozeß
         freigegeben                                                }
       mutual^.V;
```

```
  Inc (count);
  MTContinue;
  sem.PnoControl;
END   { TCondition.WAIT };


PROCEDURE TCondition.SIGNAL;

BEGIN { TCondition.SIGNAL }
  MTBlock;
  IF count > 0
  THEN BEGIN
         Inc (dringend^);
         Dec (count);
         sem.V;
         MTContinue;
         sem_dringend^.PnoControl;
         MTBlock;
         Dec (dringend^);
       END;
  MTContinue;
END   { TCondition.SIGNAL };


FUNCTION TCondition.QUEUE : BOOLEAN;

BEGIN { TCondition.QUEUE }
  MTBlock;
  IF count > 0 THEN QUEUE := TRUE
               ELSE QUEUE := FALSE;
  MTContinue;
END   { TCondition.QUEUE };


DESTRUCTOR TCondition.done;

BEGIN { TCondition.done }
  sem.close;
END   { TCondition.done };


END.
```

Hiermit wird der Monitor in Abbildung 12.2-3 in die in Abbildung 12.2-4 gezeigte Unit umgewandelt.

```
{ MONITOR } UNIT name;

INTERFACE

{ ENTRY } PROCEDURE proc_1 (parameterlist_1);

 ...

{ ENTRY } PROCEDURE proc_m (parameterlist_m);

IMPLEMENTATION

USES Condition, Semaphor;

CONST dringend   : INTEGER = 0;

VAR sem_dringend : PSemaphor;
    mutual       : PSemaphor;
...

VAR cv : TCondition;

...

{ ENTRY } PROCEDURE proc_1 (parameterlist_1);
  ...
  BEGIN { proc_1 }
    mutual^.PnoControl;
    ...
    cv.WAIT;
    ...
    IF dringend > 0 THEN sem_dringend^.V
                    ELSE mutual^.V;
  END   { proc_1 };

...

{ ENTRY } PROCEDURE proc_m (parameterlist_m);
  ...
  BEGIN { proc_m }
    mutual^.PnoControl;
    ...
    cv.SIGNAL;
    ...
    IF dringend > 0 THEN sem_dringend^.V
                    ELSE mutual^.V;
  END   { proc_m };

...

BEGIN { Initialisierungsteil }
  sem_dringend := New (PSemaphor, init (0));
  mutual       := New (PSemaphor, init (1));
  cv.init (@dringend, sem_dringend, mutual);
  ...
END   { MONITOR/UNIT name }.
```

Abbildung 12.2-4: Simulation eines Monitors durch Semaphore (Fortsetzung)

13 Anhang

13.1 Exkurs über Zahlensysteme

13.1.1 Natürliche Zahlen

Die übliche Darstellung einer natürlichen Zahl verwendet die **Dezimaldarstellung**, d.h. eine *k*-stellige natürliche Zahl *m* wird als eine Folge von **Dezimalziffern** notiert:

$$m = [a_{k-1}a_{k-2}\ldots a_1a_0]_{10}$$

$$= a_{k-1}\cdot 10^{k-1} + a_{k-2}\cdot 10^{k-2} + \ldots + a_1\cdot 10 + a_0$$

$$= \sum_{i=0}^{k-1} a_i \cdot 10^i \text{ mit } a_i \in \{0,1,2,3,4,5,6,7,8,9\} \text{ für } i = 0,\ldots,k-1.$$

Beispiel:
$$m = 2139$$
$$= [2139]_{10}$$
$$= 2\cdot 1000 + 1\cdot 100 + 3\cdot 10 + 9$$
$$= 2\cdot 10^3 + 1\cdot 10^2 + 3\cdot 10^1 + 9\cdot 10^0.$$

Der **Basiswert** des Dezimalsystems ist der Wert 10; die **Ziffern** im Dezimalsystem sind 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Das folgende **Rechenschema liefert nacheinander die Ziffernfolge** $a_0, a_1, \ldots, a_{k-2}, a_{k-1}$ einer natürlichen Zahl *m* im Dezimalsystem:

Schritt 1:	Setze x = *m*. Weiter bei Schritt 2.
Schritt 2:	Dividiere x mit Restbildung durch den Basiswert 10. Das Ergebnis heiße y, der Rest ist die nächste Ziffer in der Ziffernfolge von *m*. Weiter bei Schritt 3.
Schritt 3:	Falls y ≠ 0 ist, ersetze den Wert von x durch den Wert von y und wiederhole Schritt 2 mit diesem neuen Wert von x. Falls y = 0 ist, beende das Rechenverfahren.

Beispiel: $m = 2139$

x	: 10 =	y	Rest (Ziffer)
2139	: 10 =	213	Rest $9 = a_0$
213	: 10 =	21	Rest $3 = a_1$
21	: 10 =	2	Rest $1 = a_2$
2	: 10 =	0	Rest $2 = a_3$

Offensichtlich liegen die Reste jeweils zwischen 0 und 9, so daß man im Dezimalsystem mit 10 Ziffern auskommt.

Durch die Wahl eines von 10 verschiedenen Basiswerts erhält man die Darstellung einer natürlichen Zahl *m* in anderen Zahlensystemen. In der Informatik häufig verwendete Basiswerte sind 2, 8 und 16. In diesen Fällen spricht man vom Binärsystem (Basiswert 2), Oktalsystem (Basiswert 8) bzw. Sedezimalsystem (Basiswert 16). Im folgenden werden nur das Binär- und das Sedezimalsystem behandelt.

Im **Binärsystem** ist der **Basiswert** 2, so daß eine natürliche Zahl *m* als eine Folge von **Binärziffern** 0 bzw. 1 notiert wird:

$$m = [b_{n-1}b_{n-2}\ldots b_1 b_0]_2$$

$$= b_{n-1}\cdot 2^{n-1} + b_{n-2}\cdot 2^{n-2} + \ldots + b_1 \cdot 2 + b_0$$

$$= \sum_{i=0}^{n-1} b_i \cdot 2^i \text{ mit } b_i \in \{0,1\} \text{ für } i = 0,\ldots,n-1.$$

Beispiel: $m = 2139$

$$= [2139]_{10}$$

$$= [100001011011]_2$$

$$= 1\cdot 2^{11} + 0\cdot 2^{10} + 0\cdot 2^9 + 0\cdot 2^8 + 0\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0.$$

Das obige Rechenschema liefert nacheinander die Ziffernfolge $b_0, b_1, \ldots, b_{n-2}, b_{n-1}$ einer natürlichen Zahl *m* im Binärsystem, wenn man in Schritt 2 den Basiswert 2 nimmt.

Beispiel: $m = 2139$

x	: 2 =	y	Rest	(Ziffer)
2139	: 2 =	1069	Rest	$1 = b_0$
1069	: 2 =	534	Rest	$1 = b_1$
534	: 2 =	267	Rest	$0 = b_2$
267	: 2 =	133	Rest	$1 = b_3$
133	: 2 =	66	Rest	$1 = b_4$
66	: 2 =	33	Rest	$0 = b_5$
33	: 2 =	16	Rest	$1 = b_6$
16	: 2 =	8	Rest	$0 = b_7$
8	: 2 =	4	Rest	$0 = b_8$
4	: 2 =	2	Rest	$0 = b_9$
2	: 2 =	1	Rest	$0 = b_{10}$
1	: 2 =	0	Rest	$1 = b_{11}$

Offensichtlich sind die Reste jeweils 0 oder 1, so daß man im Binärsystem mit 2 Ziffern auskommt.

Im **Sedezimalsystem** ist der **Basiswert** 16. Analog zu den bisherigen Ausführungen benötigt man zur Zahlendarstellung im Sedezimalsystem genausoviele verschiedene Sedezimalziffern, wie die Basis angibt, also hier 16. Zu den üblichen Ziffern 0 bis 9 werden sechs neue Ziffern hinzugenommen, die man mit A, B, C, D, E und F bezeichnet. Im Sedezimalsystem wird eine natürliche Zahl m als eine Folge von **Sedezimalziffern** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F notiert:

$$m = [s_{l-1}s_{l-2}\ldots s_1 s_0]_{16}$$

$$= s_{l-1}\cdot 16^{l-1} + s_{l-2}\cdot 16^{l-2} + \ldots + s_1\cdot 16 + s_0$$

$$= \sum_{i=0}^{l-1} s_i \cdot 16^i \text{ mit } s_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\} \text{ für } i = 0,\ldots,l-1.$$

Die Sedezimalziffer A ist entspricht also dem Dezimalwert 10, die Sedezimalziffer B dem Dezimalwert 11, die Sedezimalziffer C dem Dezimalwert 12, die Sedezimalziffer D dem Dezimalwert 13, die Sedezimalziffer E dem Dezimalwert 14 und die Sedezimalziffer F dem Dezimalwert 15.

Beispiel: $m = 2139$

$= [2139]_{10}$

$= [100001011011]_2$

$= [85B]_{16}$

$= 8 \cdot 256 + 5 \cdot 16 + 11$

$= 8 \cdot 16^2 + 5 \cdot 16^1 + 11 \cdot 16^0.$

Die Sedezimalziffernfolge $s_0, s_1, \ldots, s_{l-2}, s_{l-1}$ einer natürlichen Zahl m im Sedezimalsystem erhält man aus obigem Rechenschema, wenn man in Schritt 2 den Basiswert 16 nimmt:

Beispiel: $m = 2139$

x	: 16 =	y	Rest (Ziffer)
2139	: 16 =	133	Rest $11_{10} = \mathrm{B} = s_0$
133	: 16 =	8	Rest $5 = s_1$
8	: 16 =	0	Rest $8 = s_2$

Eine natürliche Zahl m, die im Dezimalsystem k Stellen hat (wobei führende Nullen weggelassen werden), benötigt ca. $n = 3{,}32 \cdot k$ Stellen im Binärsystem und ca. $l = 0{,}83 \cdot k = \frac{1}{4} \cdot n$ Stellen im Sedezimalsystem.

Die **Umwandlung einer natürlichen Zahl von Dezimal- in Binär- bzw. Sedezimaldarstellung** kann mit Hilfe des angegebenen Rechenschemas durchgeführt werden. Aus der Binär- bzw. Sedezimaldarstellung kommt man mit Hilfe der Formeln

$m = [b_{n-1} b_{n-2} \ldots b_1 b_0]_2$

$= b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \ldots + b_1 \cdot 2 + b_0$

$= \sum_{i=0}^{n-1} b_i \cdot 2^i$

bzw.

$m = [s_{l-1} s_{l-2} \ldots s_1 s_0]_{16}$

$= s_{l-1} \cdot 16^{l-1} + s_{l-2} \cdot 16^{l-2} + \ldots + s_1 \cdot 16 + s_0$

$= \sum_{i=0}^{l-1} s_i \cdot 16^i.$

Die **Umwandlung einer natürlichen Zahl von Binär- in Sedezimaldarstellung** erfolgt dadurch, daß man von rechts jeweils 4 Binärstellen zu einer Sedezimalstelle zusammenfaßt; eventuell müssen dann links führende Nullen ergänzt werden.

Beispiel: $m = 2139$

$= [2139]_{10}$

$= [1000 \quad 0101 \quad 1011]_2$

$= [\ 8 \quad 5 \quad B\ ]_{16}$

Die **Umwandlung einer natürlichen Zahl von Sedezimal- in Binärdarstellung** erfolgt dadurch, daß man jede Sedezimalziffer als Ziffernfolge der 4 Binärziffern mit dem entsprechenden Wert schreibt.

13.1.2 Binär- und Sedezimaldarstellung ganzer Zahlen

Zunächst wird eine Stellenzahl n festgelegt, mit der Zahlen in Binärdarstellung notiert werden. Übliche Größen in Rechenanlagen sind $n = 16$ und $n = 32$. Jede Zahl m hat dann die Form

$m = [b_{n-1}b_{n-2}\ldots b_1b_0]_2$.

Für die absolute Größe des Zahlenwerts von m sind nur die rechten n-1 Stellen $b_{n-2}, \ldots b_1b_0$ ausschlaggebend; die am weitesten links stehende Stelle b_{n-1} bestimmt, ob m als positive oder als negative Zahl interpretiert wird: Für $b_{n-1} = 0$ ist m positiv, für $b_{n-1} = 1$ ist m negativ. Auf diese Weise wird die Binärziffernfolge

$0b_{n-2}b_{n-3}\ldots b_1b_0$ mit $b_i \in \{0,1\}$ für $i = 1,\ldots,n-2$

als

$$\sum_{i=0}^{n-2} b_i 2^i$$

interpretiert. Entsprechend repräsentiert die Binärziffernfolge

$1b_{n-2}b_{n-3}\ldots b_1b_0$

den Wert

$$\sum_{i=0}^{n-2} b_i 2^i - 2^{n-1}$$

und somit eine negative Zahl, da

$$\sum_{i=0}^{n-2} b_i 2^i \le 2^{n-1} - 1$$

ist.

Auch hier werden wieder üblicherweise vier aufeinanderfolgende Binärziffern zu einer Sedezimalziffer zusammengefaßt. Bei $n = 32$ vorgesehenen Binärstellen wird damit ein (asymmetrischer) Zahlenbereich von $-2\,147\,483\,648_{10} = -2^{31}$ bis $+2\,147\,483\,647_{10} = +2^{31}-1$ ermöglicht:

Binärdarstellung	Sedezimaldarstellung	Dezimalwert
0000 0000 ... 0000 0000	00 00 00 00	0
0000 0000 ... 0000 0001	00 00 00 01	1
0000 0000 ... 0000 0010	00 00 00 02	2
0000 0000 ... 0000 0011	00 00 00 03	3
...	...	...
0111 1111 ... 1111 1111	7F FF FF FF	+ 2 147 483 647
1000 0000 ... 0000 0000	80 00 00 00	- 2 147 483 648
1000 0000 ... 0000 0001	80 00 00 01	- 2 147 483 647
1000 0000 ... 0000 0010	80 00 00 02	- 2 147 483 646
...	...	...
1111 1111 ... 1111 1110	FF FF FF FE	- 2
1111 1111 ... 1111 1111	FF FF FF FF	- 1

Der **Übergang von einer Zahl m zu der betragsmäßig gleichen Zahl mit entgegengesetztem Vorzeichen -m** erfolgt durch **Komplementbildung** (**2er-Komplement**):

1. Man ersetzt in der Binärdarstellung von *m* jede Binärziffer 0 durch die Binärziffer 1 bzw. jede Binärziffer 1 durch die Binärziffer 0.

2. Man addiert eine 1 auf die niedrigstwertige Stelle. Ein eventueller Übertrag wird auf die benachbarte linke Stelle addiert. Die Rechenregeln lauten also binärstellenweise
   0 + 0 = 0
   0 + 1 = 1 + 0 = 1
   1 + 1 = 0 und Übertrag auf die links benachbarte Stelle.

Das entsprechende Verfahren kann man auch mit der Sedezimaldarstellung einer Zahl *m* durchführen:

1. Man ersetzt in der Sedezimaldarstellung von *m* jede Sedezimalziffer s_i durch die Sedezimalziffer $\overline{s_i}$, die s_i auf den Wert $15_{10} = F_{16}$ ergänzt, d.h. für die $s_i + \overline{s_i} = F$ gilt.

2. Man addiert eine 1 auf die niedrigstwertige Stelle. Ein eventueller Übertrag wird auf die benachbarte linke Stelle addiert.

Beispiele:

2139_{10} = 0000 0000 ... 0000 1000 0101 1011_2 = 00 00 08 $5B_{16}$
-2139_{10} = 1111 1111 ... 1111 0111 1010 0101_2 = FF FF F7 $A5_{16}$

1_{10} = 0000 0000 ... 0000 0000 0000 0001_2 = 00 00 00 01_{16}
-1_{10} = 1111 1111 ... 1111 1111 1111 $111\mathbf{1}_2$ = FF FF FF $F\mathbf{F}_{16}$

128_{10} = 0000 0000 ... 0000 0000 1000 0000_2 = 00 00 00 80_{16}
-128_{10} = 1111 1111 ... 1111 1111 **1000** **0000**$_2$ = FF FF FF 80_{16}

13.1.3 Rechnen im Binär- und Sedezimalsystem

Die **Addition im Binärsystem** erfolgt binärstellenweise von rechts nach links unter Beachtung der arithmetischen Regeln 0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1 = 0 und Übertrag auf die links benachbarte Stelle. Die **Subtraktion** einer Zahl *m* erfolgt durch Addition der betragsmäßig gleichen Zahl *-m* mit umgekehrtem Vorzeichen. Entsprechendes gilt für das Sedezimalsystem.

Beispiele:

$2139_{10} + 128_{10}$ =		0000	0000	...	0000	1000	0101	1011_2		00	00	08	$5B_{16}$
	+	0000	0000	...	0000	0000	1000	0000_2	+	00	00	00	80_{16}
	=	0000	0000	...	0000	1000	1101	1011_2	=	00	00	08	DB_{16}
$2139_{10} - 128_{10}$ =		0000	0000	...	0000	1000	0101	1011_2		00	00	08	$5B_{16}$
	+	1111	1111	...	1111	1111	1000	0000_2	+	FF	FF	FF	80_{16}
	=	0000	0000	...	0000	1000	1101	1011_2	=	00	00	07	DB_{16}
$128_{10} - 2139_{10}$ =		0000	0000	...	0000	0000	1000	0000_2		00	00	00	80_{16}
	+	1111	1111	...	1111	0111	1010	0101_2	+	FF	FF	F7	$A5_{16}$
	=	1111	1111	...	1111	1000	0010	0101_2	=	FF	FF	F8	25_{16}

Ergebnis: -2011_{10} = FF FF F8 25_{16} = - 00 00 07 DB_{16}

Die **Multiplikation im Binärsystem** erfolgt im wesentlichen durch sukzessive Addition des Multiplikanten gemäß dem "Binärziffernmuster" des Multiplikators mit getrennter Vorzeichenrechnung. Entsprechend erfolgt die (ganzzahlige) **Division**[49] **im Binärsystem** durch sukzessive Subtraktion.

[49] Division ohne Rest. Beispielsweise ist $2139_{10} \div 129_{10} = 16_{10}$ Rest 75_{10}.

Beispiel:

$131_{10} * 11_{10} = \ldots\ 0000\ 1000\ 0011_2 * 1011_2$

```
                    000010000011
                     000000000000
                      000010000011
                       000010000011
```

$1441_{10} = \ldots 000010110100001_2$

13.1.4 Zahlenbereichsüberlauf

Da für die Darstellung der Zahlen nur jeweils eine maximale Stellenzahl *n* vorgesehen ist, kann es vorkommen, daß das Ergebnis einer arithmetischen Operation mehr als *n* Stellen benötigt, und zwar möglicherweise bei der Addition zweier positiver oder zweier negativer Zahlen. In diesem Fall spricht man von einem **Zahlenbereichüberlauf,** insbesondere von einem **Festpunktüberlauf**. Ein Festpunktüberlauf kann nicht bei der Addition einer positiven zu einer negativen Zahl entstehen.

Beispiel: *n* = 32 Binärstellen

+ 2 130 706 432$_{10}$	0111 1111 0000 ... 0000$_2$	7F 00 00 00$_{16}$
+ 16 777 216$_{10}$	0000 0001 0000 ... 0000$_2$	01 00 00 00$_{16}$
= 2 147 483 648$_{10}$	1000 0000 0000 ... 0000$_2$	80 00 00 00$_{16}$
	= -2 147 483 648$_{10}$	

- 2 147 483 645$_{10}$	1000 0000 0000 ... 0011$_2$	80 00 00 03$_{16}$
- 5$_{10}$	1111 1111 1111 ... 1011$_2$	FF FF FF FB$_{16}$
= - 2 147 483 650$_{10}$	0111 1111 1111 ... 1110$_2$	7F FF FF FE$_{16}$
	= +2 147 483 646$_{10}$	

Zur leichteren Feststellung eines Festpunktüberlaufs wird der Binärdarstellung einer Zahl eine zusätzliche Binärstelle ("Schutzstelle") vorangestellt, die den gleichen Wert wie die Vorzeichenstelle hat. Eine gültige positive Zahl beginnt so in der Schutz- und Vorzeichenstelle mit der Ziffernkombination 00_2, eine gültige negative Zahl mit 11_2. Bei der Addition wird diese zusätzliche Stelle in die Rechnung mit einbezogen. Ein möglicherweise auftretender Übertrag in die nicht mehr vorhandene Stelle links der Schutzstelle wird ignoriert. Einen Festpunktüberlauf erkennt man daran, daß das Ergebnis der Addition in der Schutz- und Vorzeichenstelle die Binärziffernkombination 01_2 oder 10_2 enthält.

13.1.5 Darstellung reeller Zahlen

Eine reelle Zahl wird durch eine rationale Zahl mit endlichem gebrochenem Anteil angenähert. Die entstehende Zahlendarstellung heißt **Gleitpunktdarstellung**.

Eine Zahl x mit ganzzahligem und gebrochenem Anteil läßt sich in der Form

$x = \pm|x| = \pm m B^e$

schreiben. Dabei bezeichnet ± das **Vorzeichen**, m die **Mantisse**, B die **Basis** der Zahlendarstellung und e den **Exponenten** der Zahl.

Die Wahl der Basis und die Art der im folgenden gewählten Normalisierung hängt vom Rechnertyp ab. Es sei hier $B = 16$.

Die Mantisse m ist eine Ziffernfolge von Sedezimalziffern:
$m = b_{-1}b_{-2}\ldots b_{-k}$ mit $b_{-1} \neq 0$.

Die Ziffernfolge der Mantisse wird als

$$\sum_{i=1}^{k} b_{-i} 16^{-i}$$

interpretiert. Durch die Normalisierung $b_{-1} \neq 0$ ist der Exponent eindeutig bestimmt; vereinbarungsgemäß hat der Wert 0 die Mantisse $m = 0$ und einen fest definierten Exponenten e_0. Die Darstellung soll durch einige Beispiel erläutert werden. Dabei werden Zahlen mit ganzzahligem und gebrochenem Anteil von ihrer Dezimaldarstellung in Sedezimaldarstellung umgewandelt und anschließend normiert. Diese Umwandlung erfolgt getrennt für den ganzzahligen Anteil durch sukzessive Division durch 16 und für den gebrochenen Anteil durch sukzessive Multiplikation und Notieren der "Überläufe". Der letzte Teil, nämlich die Umwandlung eines gebrochenen Anteils einer Zahl von Dezimal- in Sedezimaldarstellung, wird zunächst erläutert:

Der umzuwandelnde Dezimalwert a mit $0 \leq a < 1$ wird mit 16 multipliziert (genauer durch 16^{-1} dividiert). Der ganzzahlige Anteil des Multiplikationsergebnisses, der als Überlauf der Multiplikation bezeichnet werden soll, stellt die erste Sedezimalziffer b_{-1} von a nach dem "Sedezimalkomma" dar. Es gilt $0 \leq b_{-1} < 16$. b_{-1} wird vom Multiplikationsergebnis subtrahiert. Der verbleibende Rest m_1, für den $0 \leq m_1 < 1$ gilt, wird wieder mit 16 multipliziert. Der entstehende Überlauf, d.h. der ganzzahlige Anteil des Multiplikationsergebnisses, bildet die zweite Sedezimalziffer b_{-2} von a nach dem Sedezimalkomma. Wieder wird der Überlauf, d.h. b_{-2}, vom Multiplikationsergebnis subtrahiert, wodurch der Rest m_2 mit $0 \leq m_2 < 1$ entsteht. Das Verfahren wird solange fortgesetzt, wie Sedezimalziffern für die Darstellung nach dem Sedezimalkomma benötigt werden.

Die folgenden Beispiele zeigen das Verfahren, wobei für die Mantisse nach der Normierung $b_{-1} \neq 0$ jeweils $k = 6$ Sedezimalziffern verwendet werden.

$46{,}415_{10}$ hat den ganzzahligen Anteil $46_{10} = 2E_{16}$ und den gebrochenen Anteil $0{,}415_{10}$. Der gebrochene Anteil wird wie folgt umgewandelt (alle Ziffern sind Dezimalziffern):

0,415 · 16 = 6,64, d.h. Überlauf 6
0,64 · 16 = 10,24, d.h. Überlauf 10
0,24 · 16 = 3,84, d.h. Überlauf 3
0,84 · 16 = 13,44, d.h. Überlauf 13
0,44 · 16 = 7,04, d.h. Überlauf 7
0,04 · 16 = 0,64, d.h. Überlauf 0, ab hier periodisch, denn

0,64 · 16 = 10,24, d.h. Überlauf 10.

Also hat $46{,}415_{10} = 2E{,}6A3D70A3D70..._{16}$ in normierter Darstellung die Mantisse 2E 6A $3D_{16}$ und den Exponenten 2.

$-0{,}03125_{10} = -8 \cdot 16^{-2} = -0{,}008_{16} = -0{,}8_{16} \cdot 16^{-1}$ hat in normierter Darstellung die Mantisse $80\ 00\ 00_{16}$ und den Exponenten -1.

$-32{,}57_{10} = -(2 \cdot 16^{1} + 0 \cdot 16^{0} + 9 \cdot 16^{-1} + 1 \cdot 16^{-2} + 14 \cdot 16^{-3} + 11 \cdot 16^{-4} + ...) = -20{,}91EB..._{16} = -0{,}2091EB..._{16} \cdot 16^{2}$ hat in normierter Darstellung die Mantisse $20\ 91\ EB_{16}$ und den Exponenten 2.

$213{,}04_{10} = 13 \cdot 16^{1} + 5 \cdot 16^{0} + 0 \cdot 16^{-1} + 10 \cdot 16^{-2} + 13 \cdot 16^{-4} + ... = D5{,}0A3D..._{16} = 0{,}D50A3D \cdot 16^{2}$ hat in normierter Darstellung die Mantisse D5 0A $3D_{16}$ und den Exponenten 2.

Häufig wird anstelle des Exponenten einer Zahl deren **Charakteristik** *c* notiert, die durch

$$c = e + b$$

definiert ist, wobei *b* als **Bias** bezeichnet wird (vgl. Kapitel 4.1.1). Durch diese Umrechnung wird ein Exponentenbereich mit positiven und negativen Werten auf einen nichtnegativen Wertebereich abgebildet, der als vorzeichenlose Festpunktzahl behandelt werden kann (Abbildung 13.1.5-1).

Für die oben angeführten Beispiele lauten die einzelnen Teile der Gleitpunktdarstellung (für den Exponenten werden 7 Binärstellen genommen, er deckt also den Zahlenbereich von $1000000_2 = -64_{10}$ bis $0111111_2 = 63^{10}$ ab, der Bias lautet $b = 64_{10}$, Mantisse mit 24 Binärziffern):

$46{,}415_{10} = 0{,}2E6A3D_{16} \cdot 16^{2}$;
Vorzeichen = +;
Exponent = 2; Charakteristik = 1000010_2;
Mantisse = 2E 6A $3D_{16} = 0010\ 1110\ 0110\ 1010\ 0011\ 1101_2$

$-0{,}03125_{10} = -0{,}8_{16} \cdot 16^{-1}$;
Vorzeichen = -;
Exponent = -1; Charakteristik = 0111111_2;
Mantisse = 80 00 00_{16} = 1000 0000 0000 0000 0000 0000_2

$-32{,}57_{10} = -0{,}2091EB..._{16} \cdot 16^2$;
Vorzeichen = -;
Exponent = 2; Charakteristik = 1000010_2;
Mantisse = 20 91 EB_{16} = 0010 0000 1001 0001 1110 1011_2

$213{,}04_{10} = 0{,}D50A3D \cdot 16^2$;
Vorzeichen = +;
Exponent = 2; Charakteristik = 1000010_2;
Mantisse D5 0A $3D_{16}$ = 1101 0101 0000 1010 0011 1101_2.

Beispiel: $b = 64_{10}$			
Exponent		Charakteristik	
dezimal	binär	binär	sedezimal
-64	1000000	0000000	00
-63	1000001	0000001	01
...	...	...	...
- 1	1111111	0111111	3F
0	0000000	1000000	40
1	0000001	1000001	41
2	0000010	1000010	42
...	...	...	...
62	0111110	1111110	7E
63	0111111	1111111	7F

Abbildung 13.1.5-1: Exponent und Charakteristik

Die Arithmetik mit Gleitkommazahlen erfordert komplexe Operationen wie die Angleichung der Exponenten und die Normalisierung berechneter Ergebnisse. Typische Fehlersituationen sind hier Exponentenüberlauf (der Ergebnisexponent einer arithmetischen Operation paßt nicht in die vorgesehene Stellenzahl) oder die Tatsache, daß sich die Mantisse mit Wert 0 ergibt.

13.2 Codes zur internen Zeichendarstellung

Üblicherweise werden zur internen Darstellung von Zeichen der ASCII-Code bzw. der EBCDI-Code verwendet. Beide Zeichensätze kodieren Ziffern, Buchstaben, Sonderzeichen und Steuerzeichen, die z.B. in der Textverarbeitung und dem Datentransfer vorkommen. Beide Codes definieren sowohl internationale als auch nationale Versionen.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	~	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	QS	-	=	M	]	m	{
E	SO	RS	.	>	N	^	n	-
F	SI	US	/	?	O	_	o	DEL

Bedeutung der Steuerzeichen:

NUL	(Füllzeichen, Null)	DCx, x=1..4	Gerätesteuerung (Device Control)
SOH	Anfang des Kopfes (Start of Heading)		
STX	Anfang des Textes (Start of Text)	NAK	Negative Rückmeldung (Negative Acknowledge)
ETX	Ende des Textes (End of Text)		
EOT	Ende der Übertragung (End of Transmission)	SYN	Synchronisation (Synchronous Idle)
		ETB	Ende des Datenübertragungsblocks (End of Transmission Block)
ENQ	Stationsaufforderung (Enquiry)		
ACK	Positive Rückmeldung (Acknowledge)	CAN	Ungültig (Cancel)
BEL	Klingel (Bell)	EM	Ende der Aufzeichnung (End of Medium)
BS	Rückwärtsschritt (Backspace)	SUB	Substitution (Substitute Character)
HT	Horizontal-Tabulator (Horizontal Tabulation	ESC	Umschaltung (Escape)
		FS	Hauptgruppen-Trennung (File Separator)
LF	Zeilenvorschub (Line Feed)	GS	Gruppen-Trennung (Group Separator)
VT	Vertikal-Tabulator (Vertical Tabulation)	RS	Untergruppen-Trennung (Record Separator)
FF	Formularvorschub (Form Feed)		
CR	Wagenrücklauf (Carriage Return)	US	Teilgruppen-Trennung (Unit Separator)
SO	Dauerumschaltung (Shift-out)	DEL	Löschen (Delete)
SI	Rückschaltung (Shift-in)		
DLE	Datenübertragungsumschaltung (Data Link Escape)		

Abbildung 13.2-1: ASCII-Code

Der **ASCII-Code** (American standard code for information interchange) ist ein 7-Bit-Code, definiert also 128 Zeichen. Das im Byteformat freie achte Bit kann als Prüfbit oder zur Erweiterung auf einen 8-Bit-Code, z.B. für verschiedene Drucker-Zeichensätze, verwendet werden. Abbildung 13.2-1 zeigt den ASCII-Code in seiner internationalen Ausprägung. Ein zu kodierendes Zeichen steht im Schnittpunkt einer Spalte und einer Zeile. Die linke Sedezimalziffer ist in der Spaltenbeschriftung abzulesen, die rechte Sedezimalziffer in der Zeilenbeschriftung. Beispielsweise wird das Zeichen + (Plus) durch den Bytewert $2B_{16}$ dargestellt. Die durch einen Text von 2 oder 3 Buchstaben beschriebenen "Zeichen" stellen

definierte Steuerzeichen dar, denen kein druckbares Zeichen entspricht. Beispielsweise hat das in der Datenkommunikation häufig verwendete ESC-Zeichen (ESCAPE) die Kodierung $1C_{16}$ und das Leerzeichen (SPACE) die Kodierung 20_{16}.

Der **EBCDI-Code** (extended binary coded decimal interchange code) benutzt die 8 Bits eines Bytes, wobei einige Bitkombinationen im Code nicht definiert sind. Ein Zeichen besteht dabei aus einer linken Sedezimalziffer, dem **Zonenteil**, und einer als **Zifferntei**l bezeichneten rechten Sedezimalziffer. Abbildung 13.2-2 tabelliert den EBCDI-Code in der internationalen und der deutschen Version. Ein zu kodierendes Zeichen steht wieder im Schnittpunkt einer Spalte und einer Zeile. Die linke Sedezimalziffer ist in der Spaltenbeschriftung abzulesen, die rechte Sedezimalziffer in der Zeilenbeschriftung. Einige Bytewerte sind in der internationalen bzw. deutschen Version mit doppelten Bedeutungen belegt (z.B. bedeutet der Bytewert $7C_{16}$ entweder das Zeichen @ oder das Zeichen §); welche Bedeutung im Einzelfall genommen wird, ist maschinenabhängig.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL				SP	&	-									0
1							/		a	j			A	J		1
2									b	k	s		B	K	S	2
3									c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	EOB	UC					f	o	w		F	O	W	6
7	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
8									h	q	y		H	Q	Y	8
9									i	r	z		I	R	Z	9
A			SM		¢	! ÷	^	:								
B					.	$¤	,	# £				[Ä				{ ä
C					<	*	%	@ §				\ Ö				
D					(	)	_	'				] Ü				} ü
E					+	;	>	=								
F					| ö	¬	?	"								~ ß

Bedeutung der Steuerzeichen:

NUL	(Füllzeichen)	EOB	Blockende
PF	Stanzer aus	PRE	(Bedeutungsänderung der beiden Folgezeichen)
HT	Horizontaltabulator		
LC	Kleinbuchstaben	PN	Stanzer ein
DEL	Löschen	RS	Leser Stop
RES	Sonderfolgenende	UC	Großbuchstaben
NL	Zeilenvorschub mit Wagenrücklauf	EOT	Ende der Übertragung
BS	Rückwärtsschritt	SM	Betriebsartenänderung
IL	Leerlauf	SP	Leerzeichen
LF	Zeilenvorschub		

Abbildung 13.2-2: EBCDI-Code

Während der EBCDI- und der ASCII-Code für die Darstellung eines Zeichens ein Byte, also 8 Bits verwenden, definiert der in neueren Betriebssystemen (z.B. WINDOWS-NT) intern eingesetzte **16-Bit-Unicode** einen Zeichensatz mit 16 Bits pro Zeichen, so daß $2^{16} = 65.536$ verschiedene Zeichen möglich und damit Zeichensätze asiatischer, arabischer und europäischer Sprachen repräsentierbar sind. Nicht alle definierten Zeichen sind eigenständige Schriftzeichen, sondern Bausteine, aus denen sich zusammen mit einem Buchstabe ein gültiges Zeichen ergibt.

Die Norm ISO/IEC 10646 definiert einen 32-Bit-Code, den **UCS-4-Octett-Code** (universal character set). Mit ihm sind $2^{32} = 4.294.967.296$ verschiedene Zeichen möglich, so daß die Schriftzeichen aller lebenden und toten Sprachen mit diesem Code darstellbar sind.

13.3 Graphen

Ein **gerichteter Graph** $G = (V, E)$ besteht aus der Menge $V = \{v_1, \ldots, v_n\}$ der **Knoten** (vertices) und der Menge $E = \{e_1, \ldots, e_k\} \subseteq V \times V$ der **Kanten** (edges).

Für eine Kante $e = (v_i, v_j)$ läuft die Kante e von v_i nach v_j. v_i heißt **Anfangsknoten** von e, v_j **Endknoten** von e. Zu einem Knoten $v \in V$ heißt $pred(v) = \{v' \mid (v', v) \in E\}$ die Menge aller **Vorgänger** von v; $succ(v) = \{v' \mid (v, v') \in E\}$ die Menge aller **Nachfolger** von v.

Ein **Kantengewichtung** für einen Graphen $G = (V, E)$ ist eine Abbildung $w: E \rightarrow \mathbf{R}$.

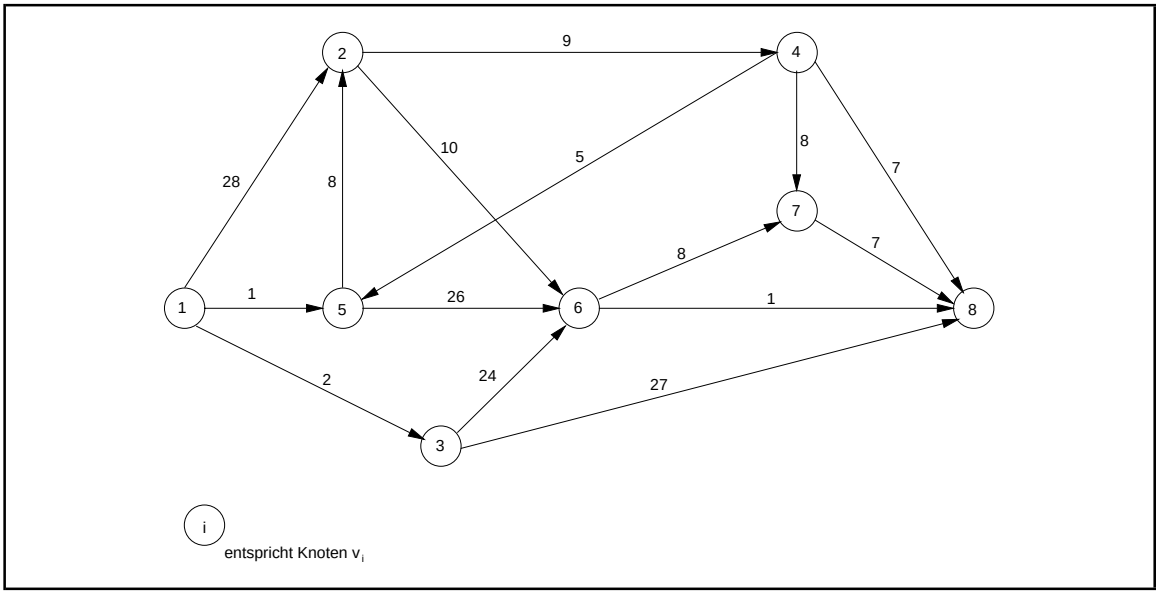


Abbildung 13.3-1: Graph

Abbildung 13.3-1 zeigt einen gerichteten Graphen $G = (V, E)$ mit
$V = \{v_1, \ldots, v_8\}$ und
$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_5), (v_2, v_4), (v_2, v_6), (v_3, v_6), (v_3, v_8), (v_4, v_5), (v_4, v_7), (v_4, v_8), (v_5, v_2), (v_5, v_6), (v_6, v_7), (v_6, v_8), (v_7, v_8)\}$.
das Gewicht $w(e) = w((v_i, v_j))$ einer Kante ist direkt an die Kante geschrieben.

Ein gewichteter Graph kann durch seine **Adjazenzmatrix** $A_{n \times n} = [a_{i,j}]_{n \times n}$ beschrieben werden:

$$a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}, 1 \le i \le n, 1 \le j \le n.$$

Im Beispiel:

$$A = \begin{bmatrix} \infty & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 9 & \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & \infty & 5 & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & \infty & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 8 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}.$$

Bei einem ungewichteten Graphen sind die Einträge der Adjazenzmatrix durch

$$a_{i,j} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}, 1 \le i \le n, 1 \le j \le n$$

definiert.

Andere Darstellungsformen eines ungewichteten Graphen sind seine Knoten/Kantenliste und seine Nachfolgerliste:

Die **Knoten/Kantenliste** besteht aus der Auflistung aller Knoten des Graphen gefolgt von der nach Anfangsknotennummern geordneten Liste aller Kanten. Die **Nachfolgerliste** enthält für jeden Knoten die Menge seiner Nachfolger. Im Beispiel lautet die Knoten/Kantenliste
$[[v_1, \ldots, v_8],$
$[(v_1, v_2), (v_1, v_3), (v_1, v_5), (v_2, v_4), (v_2, v_6), (v_3, v_6), (v_3, v_8), (v_4, v_5), (v_4, v_7), (v_4, v_8), (v_5, v_2), (v_5, v_6), (v_6, v_7), (v_6, v_8), (v_7, v_8)]]$
und die Nachfolgerliste
$[(v_2, v_3, v_5), (v_4, v_6), (v_6, v_8), (v_5, v_7, v_8), (v_2, v_6), (v_7, v_8), (v_8), ()]$.
Je nach Anwendung sit eine Darstellungsform besser geeignet als eine andere.

Ein **Pfad** von v_i nach v_j **der Länge** m in einem Graphen G ist eine Folge von Kanten $((a_0, a_1), (a_1, a_2), \ldots, (a_{m-1}, a_m))$ mit $a_0 = v_i$, $a_m = v_j$ und $(a_{i-1}, a_i) \in E$ für $i = 1, ..., m$; der Pfad **verbindet** v_i und v_j.

Ein Pfad $((a_0, a_1), (a_1, a_2), \ldots, (a_{m-1}, a_m))$ heißt **Zyklus**, falls $a_0 = a_m$ ist.

Ein **Semipfad** von v_i nach v_j ist eine Folge von Paaren $(\{a_0, a_1\}, \{a_1, a_2\}, \ldots, \{a_{m-1}, a_m\})$ mit den Eigenschaften: $v_i \in \{a_0, a_1\}$, $v_j \in \{a_{m-1}, a_m\}$ und $(a_{i-1}, a_i) \in E \vee (a_i, a_{i-1}) \in E$ für $i = 1$, ..., m; der Semipfad **verbindet** v_i und v_j.

Ein Graph heißt **zusammenhängend**, wenn es für je zwei Knoten $v_i \in V$ und $v_j \in V$ einen sie verbindenden Semipfad gibt.

Ein **Baum** ist ein zyklenfreier, zusammenhängender Graph $G = (V, E)$ mit $|V| \geq 1$ und $|E| = |V| - 1$ oder $V = E = \emptyset$ (**leerer Baum**).

Die **Wurzel eines Baums** ist der einzige Knoten, für den die Menge der Vorgänger leer ist (der keinen Vorgänger hat). Jeder andere Knoten hat genau einen Vorgänger. Ein Knoten, dessen Nachfolgermenge leer ist, heißt **Blatt**, ansonsten **innerer Knoten des Baums**.

Ein Baum der **Ordnung** $d \geq 1$ wird rekursiv wie folgt definiert:

1. Der leere Baum, der keinen Knoten enthält, ist ein Baum der Ordnung d.

2. Es seien $G_1, \ldots, G_d$ Bäume der Ordnung d. Für jedes $G_i \in \{G_1, \ldots, G_d\}$, das nicht der leere Baum ist, sei v_i die Wurzel. Man erhält einen weiteren Baum der Ordnung d, indem man einen neuen Knoten w und neue Kanten (w, v_i) einrichtet.

   In einem Baum der Ordnung d hat also jeder Knoten *höchstens d* Nachfolger.

Die **Höhe** $h(G)$ **eines Baums** $G = (V, E)$ mit $n \geq 0$ Knoten wird rekursiv definiert durch:

1. Ist G der leere Baum, d.h. $n = 0$, so hat G die Höhe $h(G) = 0$; der Baum G, der nur aus der Wurzel besteht (d.h. $n = 1$), hat die Höhe $h(G) = 1$.

2. Ist w die Wurzel des Baums G mit $n > 1$ Knoten und ist $succ(w) = \{v_1, \ldots, v_d\}$ die Nachfolgermenge von w, so ist jedes v_i, $i = 1$, ..., d, die Wurzel eines Baums $G_i = (V_i, E_i)$ mit weniger als n Knoten. Dann gilt $h(G) = \max\{h(G_1), \ldots, h(G_d)\} + 1$.

Ein **Binärbaum** ist ein Baum der Ordnung 2, bei dem also *die Nachfolgermenge für jeden Knoten aus höchstens 2 Elementen* besteht. Beispiele zeigt Abbildung 13.3-2.

Ein **AVL-Baum** (benannt nach G.M. Adelson-Velskii und E.M.Landis) ist ein Binärbaum, für den gilt: Bei jedem Knoten $v \in V$ unterscheiden sich die Höhen des linken und des rechten Teilbaums, die von v ausgehen, höchstens um 1.

Für jeden Knoten eines Baums gibt es genau einen Pfad von der Wurzel zu diesem Knoten. Die Länge dieses Pfads (er enthält keine Zyklen!), d.h. die Anzahl der Kanten auf diesem Pfad, ist der **Rang des Knotens**. Der Rang eines Knotens läßt sich ebenfalls rekursiv definieren:

1. Die Wurzel hat den Rang 0.

2. Ist *v* ein Knoten im Baum mit Rang *r*-1 und *w* ein Nachfolger von *v*, so hat *w* den Rang *r*.

Die Höhe eines Baums ist also gleich dem maximalen Rang eines Blattes + 1. Für einen Knoten *v* mit Rang *r* werden auf dem Pfad von der Wurzel bis zu *v* genau *r*+1 Knoten (einschließlich der Wurzel) besucht.

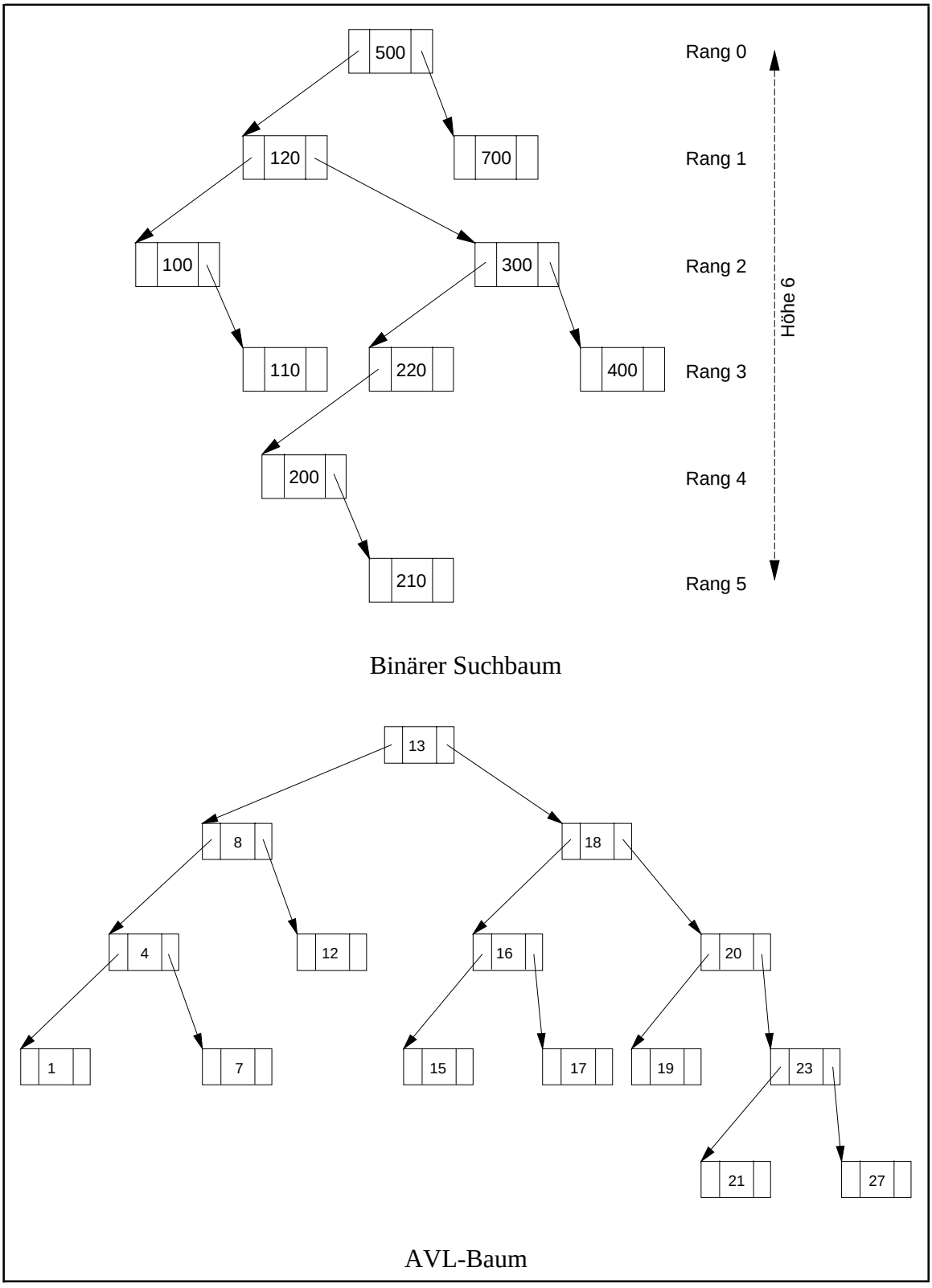


Abbildung 13.3-2: Bäume (Beispiele)

Alle Knoten gleichen Ranges bilden ein **Niveau des Baums**. Das Niveau 0 eines Binärbaums enthält nur die Wurzel, also genau einen Knoten. Das Niveau 1 eines Binärbaums enthält mindestens 1 und höchstens 2 Knoten. Das Niveau j eines Binärbaums enthält höchstens doppelt soviele Elemente wie das Niveau j-1. Daher gilt (siehe z.B. [GKP]):

Das Niveau $j \geq 0$ eines Binärbaums enthält mindestens einen und höchstens 2^j Knoten. Die Anzahl der Knoten vom Niveau 0 *bis zum Niveau j* (einschließlich) beträgt mindestens $j+1$ Knoten und höchstens $\sum_{i=0}^{j} 2^i = 2^{j+1} - 1$ Knoten.

Ein Binärbaum hat maximale Höhe, wenn jedes Niveau genau einen Knoten enthält. Er hat minimale Höhe, wenn jedes Niveau eine maximale Anzahl von Knoten enthält. Also gilt für die Höhe $h(G)$ eines Binärbaums mit n Knoten:
$\lceil \log_2(n+1) \rceil \leq h(G) \leq n$.

Bei einem AVL-Baum gilt sogar:
$\lceil \log_2(n+1) \rceil \leq h(G) \leq 1{,}4404 \cdot \log_2(n+2)$,
d.h. er "degeneriert" nicht zu einer linearen Liste.

Die Anzahl (strukturell) verschiedener Binärbäume mit n Knoten beträgt[50]
$$B(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4n}{\sqrt{n^5}}\right).$$

Die **interne Pfadlänge** $I(G)$ eines Baums G wird definiert als die Summe der Knoten, die besucht werden müssen, um alle Knoten nacheinander jeweils von der Wurzel aus zu erreichen.

Die mittlere Anzahl von Knoten, die von der Wurzel aus bis zur Erreichung eines beliebigen Knotens eines Binärbaums mit n Knoten (gemittelt über alle n Knoten) besucht werden muß, d.h. **der mittlere "Abstand" eines Knotens von der Wurzel**, ist $\sqrt{\pi n} + O(1)$. Im günstigsten Fall[51] ist der größte Abstand eines Knotens von der Wurzel in einem Binärbaum mit n Knoten $O(\log n)$, im ungünstigsten Fall ist dieser Abstand $O(n)$.

[50] Hier wird für die Approximation der Fakultäten die Stirlingsche Formel $n! \sim \sqrt{2\pi n}(n/e)^n$ verwendet.

[51] Der günstigste Fall liegt vor, wenn alle Niveaus voll besetzt sind; der ungünstigste Fall ist ein Binärbaum in Form einer linearen Liste.

Stichwortverzeichnis