RTOS–UH

Prof. Dr.–Ing. W. Gerth

Last update 21/06/2006

Vorwort

Mehr als zwnazig erfolgreiche Jahre der industriellen Anwendung von **RTOS–UH** liegen hinter uns. Die Zahl der registrierten RTOS-UH Systeme, die bisher das Licht der Welt erblickt haben, überschreitet längst die 40.000-er Grenze. Der außerordentliche Erfolg der 68k-Prozessoren in der Automatisierungstechnik hat dazu sicher seinen Teil beigetragen. Inzwischen ersetzen die schnelleren PowerPC-Prozessoren bei immer mehr Einsätzen die 68k-Familie. Oft werde ich gefragt, ob denn die Unterstützung der bewährten 68k-Familie auch in Zukunft mit vollem Engagement bei der Systempflege weitergehen wird. Wer das vorliegende Handbuch genauer liest, kennt die Antwort: Weil es für die 68k- und die PowerPC-Familie nur einen gemeinsamen Quellcode aller Systemkomponenten für die Transferassemblierung gibt, ist es gar nicht möglich, den einen oder den anderen Prozessortyp bei der Systempflege zu favorisieren.
Gegenüber der Vorgängerversion dieses Handbuches hat es kleinere Erweiterungen und Korrekturen gegeben. Nicht geändert haben wir unsere primären Zielsetzungen:

- **Kompaktheit:**

  Während allgemein der Speicherbedarf selbst für einfachste Programmfunktionen in den letzten Jahren stark angewachsen ist, passen Anwendungen mit **RTOS–UH** immer noch in die kleineren EPROMs. Unser besonderes Augenmerk gilt stets den embedded Anwendungen.

- **Skalierbarkeit:**

  Die Anwendungen reichen vom kleinsten Controller, bei dem das komplette System ohne externe Speicher direkt auf dem Chip abgelegt ist (System on a Chip) bis zum komplexen VME-System mit hunderttausenden Zeilen von Quellkode. Auch Projekte mit vielen hundert Tasks auf einem Prozessor sind keine Seltenheit.

- **Nachvollziehbare Arbeitsweise:**

  Durch die PEARL-Orientierung des Betriebssystemes ergab sich zwangsläufig eine klare und sehr präzise beschreibbare Architektur – optimal für die Automatisierungstechnik.

- **Echtzeiteigenschaften:**

  Der problematischste Bereich aller Echtzeit-Betriebssysteme – Input und Output, Vernetzung – hat in **RTOS–UH** längst seine Schrecken verloren. Bei wichtigen Strukturmerkmalen – etwa der Verlagerung des I/O auf prioritätsgerechte Dämonprozesse – werden keine Kompromisse gemacht. Wir versuchen stets, auch die unvermeidlichen Auswirkungen von Vernetzung und Multiwindowing auf die Echtzeiteigenschaften so gering wie irgend möglich zu halten. Im Rahmen der wissenschaftlichen Weiterentwicklung haben wir ein objektives Meßverfahren für die Dienstgüte eines Echtzeitbetriebssystemes entwickelt und publiziert – damit konnten wir die Systemstruktur nach wissenschaftlichen Kriterien weiter optimieren.

- **Qualitätssicherung:**

  Wir dokumentieren und verifizieren jede Änderung im System mit größter Sorgfalt. Bevor eine neue Version freigegeben wird, hat sie eine lange Testphase mit hochqualifizierten Testern zu überstehen. Wir verwenden von der Qualitätssicherungsnorm ISO 9000 die für Software gültige Handlungsanweisung ISO 9001-2 als interne Leitlinie. Viel aussagekräftiger ist jedoch eine erfolgreich abgelegte offizielle Betriebsbewährtheitsprüfung nach DIN VDE 801/A1. Für diese Zertifizierung kommen nur Systeme in Frage, die mit einer einzigen Version – das kann niemals die neueste sein – viele Millionen fehlerfreie Betriebsstunden in verschiedenen Einsatzfällen nachweisen können. In diesem Fall waren es 972 industriell eingesetzte CPUs, die über 9 Millionen Stunden ausgewertet wurden.

- **Anwenderkontakt:**

  Durch ständigen Kontakt mit den Anwendern eliminieren wir alle bekannten Fehler umgehend. Auch Anregungen zur Verbesserung greifen wir gerne auf, soweit dies mit vernünftigem Aufwand möglich ist. Das System liegt bei vielen modischen Nutzeroptionen gegenüber der PC- oder Workstation-Welt manchmal etwas zurück. Ich denke aber, daß eine strukturell saubere Einbindung neuer Dinge für den Anwender am Ende nützlicher ist als eine Hauruck-Lösung.

Die Urform dieses Handbuches wurde vor einigen Jahren von der Fa. IEP Hannover durch Umschreiben der alten Vorlage nach TEX initiiert. Aus dieser kleinen Dokumentation wurde inzwischen ein Werk mit über 700 Seiten.

Durch die Herstellung mit TEX gibt es das Handbuch auch als Postscript- und als PDF-File. Bitte beachten Sie dazu den Wegweiser auf unserer Internetseite.

Wichtig:

Trotz aller Sorgfalt ist auch unser System höchstwahrscheinlich nicht fehlerfrei. Gleiches ist auch für dieses Handbuch anzunehmen. Eine Haftung für Schäden, die durch den Gebrauch von **RTOS–UH** oder durch Fehler in diesem Dokument entstehen, wird ausdrücklich ausgeschlossen.

Bitte maximieren Sie die Zuverlässigkeit Ihrer Software und inspizieren Sie zur rechten Zeit die Fehlerbulletins, die Sie über unsere Homepage erreichen können!

www.rtos-uh.de.

www.rtos.irt.uni-hannover.de.

Diese Seite widme ich den vielen hochqualifizierten Helfern, die bei der Entstehung des Systemes mitgewirkt haben. Sie haben schwierige Teilprobleme mit Ingenieursfleiß gelöst und meist auch Code beigesteuert. Nur so konnte das System seine heutige Reife erreichen. Mein Dank gilt insbesondere:

Dr.-Ing. Dipl.-Ing. Dipl.-Ök. A. Albert	seinerzeit IRT Uni Hannover
Dipl.-Ing. R. Arlt	heute Fa. esd Hannover
Dipl.-Ing. H. Bartels	heute Fa. ATR
Dipl.-Ing. U. Bartels	seinerzeit Hannover
Dr.-Ing. S. Bunzel	seinerzeit Hannover
Dipl.-Ing. A. Domeyer	seinerzeit Hannover
Dipl.-Ing. A. Hadler	heute Fa. IEP Hannover
Prof. Dr.-Ing. R. Hausdörfer	heute FH Lippe
Dipl.-Ing. M. Huck	heute Fa. esd Hannover
Dr.-Ing. H. Husmann	seinerzeit Hannover
Dipl.-Ing. I. Jovers	seinerzeit Hannover
Dipl.-Ing. K. Koerth	heute Fa. IEP Hannover
Dipl.-Ing. B. Kroll	heute Fa. IEP Hannover
Prof. Dr.-Ing. K.-H. Niemann	seinerzeit Hannover
Dr.-Ing. T. Lilge	IRT Uni Hannover
Dr.-Ing. T. Probol	seinerzeit Hannover
Dr.-Ing. B. Wolter	seinerzeit Hannover

Für Ihre Anwendungen wünsche ich Ihnen viel Erfolg!

Hannover, im Juni 2006 Prof. Dr.-Ing. W. Gerth

Dies ist die Handbuchversion 5.4 vom 21/06/2006 Sie umfaßt 718 Seiten.

Inhaltsverzeichnis

Tabellenverzeichnis

(Leere Seite vor neuem Kapitel)

Kapitel 1: Die innere Architektur

1.1 Was muß der Systemanwender wissen?

Kaum ein Hersteller macht verwertbare Angaben über das innere Funktionsmodell seines Betriebssystemes. Das bedeutet, daß eklatante Schwächen des Systemkonzeptes manchmal sehr lange unerkannt bleiben. Eine wichtige Frage ist ja stets die nach einer sicheren und schnellen Reaktion des Rechnersystemes auf Alarme und Ausnahmesituationen. Zu dem Thema „Reaktionszeit“ hat sich leider in letzter Zeit eine unseriöse Unart eingebürgert: sehr oft wird z. B. als Reaktionszeit einfach nur jene Zeit angegeben, die vom Eintreffen des äußeren Ereignissignales bis zum Beginn der Interruptroutine (Supervisorprozeß s. u.) verstreicht. Diese Zeit ist bei **RTOS–UH** strukturbedingt absolut hervorragend, doch verwenden wir sie niemals als Qualtitätsmaß. Sie besagt nämlich für sich genommen überhaupt nicht, daß man sichere Echtzeitsoftware mit dem System erzeugen kann. Man erlebt gerade bei dieser Frage oft, daß es selbst sog. „Fachberatern“ an jeglichem Verständnis für die innere Arbeitsweise der von ihnen vertriebenen Systeme fehlt.

Mit diesem Kapitel wollen wir unsere Anwender in die Lage versetzen, bei der Auswahl eines Betriebssystemes die richtigen Fragen zu stellen und treffende Qualitätsargumente für die Verwendung unseres Systemes zu finden. Schließlich haben wir mit den intellektuellen Möglichkeiten einer Universität viele Probleme erkannt und gelöst, die von anderen Systemen nicht beherrscht werden. Bei fast jeder Echtzeitanwendung bleibt ein unbekanntes Restrisiko, weil im Probebetrieb stets nur irgendein kleiner Ausschnitt der im späteren Betrieb denkbaren zeitlichen Konstellationen wirklich getestet werden kann. Minimieren Sie dieses Risiko – wenn möglich zu Null – und studieren Sie das Multitaskingkonzept unseres Systemes.

1.2 Programme, Prozesse und Kontext

Der Begriff **Programm** ist sicher jedermann geläufig. Man versteht darunter eine Handlungsanweisung an den Rechner, bestimmte Abläufe selbständig auszuführen. Wir unterteilen Programme in „Systemprogramme“ und „Anwenderprogramme“ . Erstere werden meist vom Hersteller des Betriebssystemes stammen und sind Teil des Betriebssystemes. Anwenderprogramme enthalten dagegen die eigentliche spezielle Problemlösung, z. B. Regelalgorithmen, Anlagenüberwachung oder die Bedienerunterstützung. Anwenderprogramme benutzen die (zumeist universellen) Systemprogramme, während es eine umgekehrte Abhängigkeit nicht gibt. Dieser eingeführte Programmbegriff ist jedoch leider nicht ausreichend zum Verständnis eines modernen Multitasking-Echtzeit-Betriebssystemes.

Eine zentrale Bedeutung in modernen Echtzeitmultitaskingsystemen hat der Begriff **Prozeß**. Darunter verstehen wir den Vorgang, der abläuft, wenn der Prozessor ein Programm bearbeitet. Wenn unser Rechnersystem nur einen Prozessor besitzt, so kann zu jedem Zeitpunkt stets nur genau ein Prozeß wirklich ablaufen. Anders ausgedrückt: Es ist zwar möglich, ja sogar üblich, daß es mehr als einen Prozeß gibt, es haben jedoch nur immer genauso viele eine Ablaufgeschwindigkeit `> 0`, wie es verfügbare Prozessoren im System gibt. Prozesse sind also quasi die Inkarnation von Programmen, sie sind Subjekte, die nach einer Programmvorschrift handeln, aber manchmal vorübergehend bewegungslos verharren.

Wenn der Prozessor einen Prozeß ablaufen läßt, dann muß er sich einige Notizen machen, z. B. welcher Maschinenbefehl als nächster zu bearbeiten ist, sowie einige Zwischenresultate, die er sich in seinen Registern auf dem Chip „merkt“. Wenn der Prozeß eine Weile ruht und der Prozessor einen anderen lebendig werden läßt, so muß die Gesamtheit dieser „Notizen“ aufgehoben werden, damit eine korrekte Fortführung des im Moment ruhenden Prozesses später möglich ist. Die Gesamtheit dieser Hilfsinformationen nennen wir den **Kontext**. Aus wieviel Daten dieser Kontext besteht, hängt vom Prozessor und vom Prozeß ab: beim 68000 genügen für die „Nutzerprozesse“ (s. u.) 80 bytes, beim PowerPC dagegen sind es mindestens 160 Bytes. Bei Benutzung des Gleitkommarechenwerkes des PowerPC kann das Volumen des Kontextes sogar über 400 Bytes erreichen. Bei den sogenannten „Supervisorprozessen“ (s. u.) besteht der Teil des Kontextes, den sie brauchen und ändern, oft nur aus wenigen Bytes.

Wenn der Prozessor einen Prozeß ruhen läßt, um einen anderen (weiter) zu bearbeiten, so rettet er den alten Kontext und lädt den neuen: das nennen wir **Kontextswitch**. Wegen des größeren Datenvolumens ist der Begriff aber eigentlich nur beim Umschalten zwischen den „Nutzerprozessen“ gebräuchlich. Auch der Begriff **Prozeßumschaltung** ist in der Regel nur für den Wechsel von einem Nutzerprozeß zum anderen üblich. In den Diagrammen verwenden wir das Kürzel „csw“ für diese Umschaltung.

Um es noch einmal zu veranschaulichen, hier eine Übertragung der obigen Fachbegriffe in das normale Leben: Auf Papier gedruckte Noten treten an die Stelle der Programme. Das Flötenspiel ist der Prozeß, der diese Handlungsanweisung (Programm) lebendig werden läßt. Die Flöte samt Spieler ist logischerweise das Äquivalent zum Prozessor. Wenn mehrere Stücke gleichzeitig gespielt werden müssen, es aber nur einen Flötenspieler gibt, so muß der Spieler auf seiner Flöte abschnittweise mal dieses und mal jenes Stück weiterspielen. Dazu muß er sich aber mindestens die jeweils nächste Note und die gültige Tonart als „Kontext“ für jeden dieser Prozesse merken und bei Wiederaufnahme des Stückes erneut installieren. (Gottseidank gibt es solcherlei „Katzenmusik“ nur in den Multitaskingbetriebssystemen und nicht bei Konzerten . . .)

In der ...ix-Welt sind die „Prozesse“ normalerweise an einen Nutzerarbeitsplatz gebunden und wollen nichts miteinander zu tun haben (sie streiten sich höchstens um die Resourcen). Erst in neuerer Zeit (POSIX-Norm-Versuch) denkt man dort an eine besondere Form von Prozessen, die „Threads“ genannt werden. In einigen ...ix-Derivaten sind die Threads bereits heute installiert. Threads sollen immerhin gemeinsame Speicherzellen kennen. Ob mit der POSIX-Schnittstellennorm nach all den Jahren der Diskutiererei am Ende irgendetwas für die Automatisierungstechnik brauchbares herauskommt, kann man weiterhin anzweifeln. Das PEARL-Prozeßkonzept ist sicher erheblich umfassender, dabei aber auch flexibler und viel praxisorientierter.

In unserer **RTOS–UH**-Welt kennen wir aus guten Gründen folgende zwei Sorten von Prozessen:

- **Supervisorprozesse**.

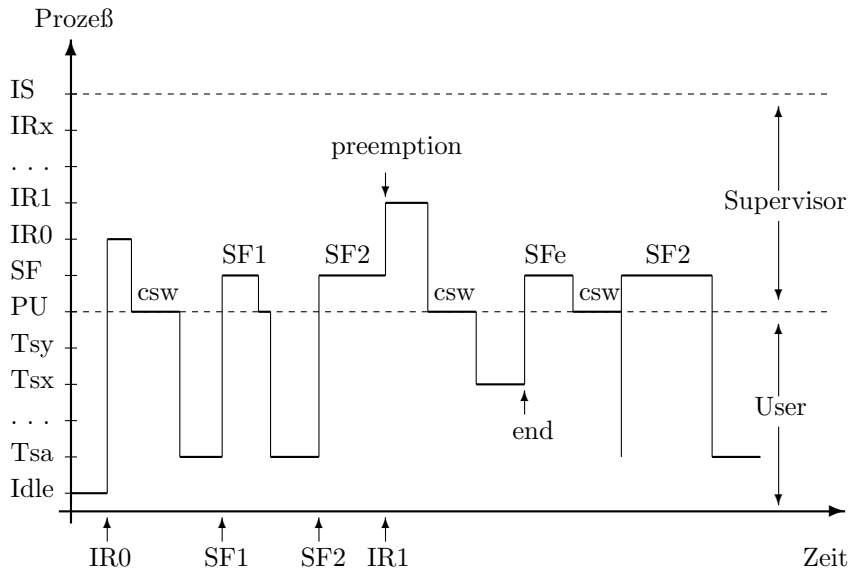
  Diese Prozesse laufen in **RTOS–UH** stets auf Systemebene im Supervisormode des Prozessors. Es würde gegen das Sicherheitskonzept unseres Systemes verstoßen, diese Sorte von Prozessen in die Hand des Nutzers zu geben. Wie unten erläutert, gehören u. a. die Interruptantwortroutinen und die mit Systemtraps angestoßenen Prozesse zu dieser Kategorie. Diese Prozesse dürfen niemals Warteschleifen oder ähnliches ausführen, denn ihre zeitliche Kürze ist entscheidend für die sichere Echtzeitfunktion des Systemes. Logischerweise soll das Datenvolumen ihres Kontextes so klein wie möglich sein.

- **Nutzerprozesse**.

  In der **RTOS–UH**-Welt nennen wir diese Prozesse auch „Tasks“ . Sie entsprechen den PEARL-Tasks und damit in etwa den „Threads“ der POSIX-Philosophie. Allerdings gibt es in **RTOS–UH** auch Prozesse (z. B. die Shell-Prozesse), die eher den normalen Prozessen der ...ix-Welt entsprechen. Das Wort „Nutzer...“ bedeutet nicht zwingend, daß in diese Gruppe nur Prozesse gehören, deren Programm ein Nutzer kodiert hat. Auch die im System immer vorhandene Leerlauftask (`#IDLE`), sowie der Errordämon (`#ERRDM`) und die I/O-Dämonen sind in diesem Sinne Nutzerprozesse.

1.3 Beschreibung des RTOS–UH-Prozeßmodelles

Das Innenleben eines Echtzeitbetriebssystemes mit seinen Stärken und Schwächen begreift der Ingenieur am besten an Hand eines sauber definierten „Prozeßmodelles“ . Dazu studieren wir hier einfach einen hypothetischen Ablauf längs der Zeitachse. Solche Prozeß/Zeit Diagramme werden vom Autor seit sehr langer Zeit in der Prozeßrechentechnikvorlesung benutzt. Sie haben sich längst beim Studium der verschiedensten Phänomene bewährt. Als Ordinate (vertikale Achse) wird mit wachsender Priorität nach oben für jeden Zeitpunkt derjenige Prozeß markiert, dessen Ablaufgeschwindigkeit von Null verschieden ist. Jeder Prozeß ist durch Prozesse, die im Diagramm unter ihm liegen, nicht unterbrechbar, solange er selbst noch arbeiten kann.



Ein beispielhaftes Prozeß/Zeitdiagramm von RTOS–UH.

An der Grenzlinie zwischen Supervisor- und Nutzerprozessen liegt der Prozeßumschalter „PU“. Er ist gleichzeitig der niedrigst priorisierte Supervisorprozeß. Jede Rückkehr des Prozessors aus dem Supervisormode oberhalb der „PU“-Linie in den User-Bereich führt zu einer (evtl. sehr kurzen) Inspektion der Sachlage durch den „PU“: es kommt zum Kontextswitch „csw“ oder nicht. Im letzteren Fall wird dies durch eine nur kurze waagerechte Linie ohne Zusatz „csw“ auf der „PU“-Ebene dargestellt.

Im Diagramm dargestellt ist die durch einen Hardwareinterrupt bewirkte Aktivierung der Task „TSa“, wie sie etwa durch das PEARL-Statement

WHEN IR0 ACTIVATE TSa;

vereinbart sein könnte.

Die Task TSa ruft dann eine Systemfunktion SF1 auf, z. B. ein erfolgreiches **REQUEST** auf eine Semaphorvariable. Im obigen Beispiel wurde angenommen, daß beim Aufruf einer weiteren Systemfunktion SF2 zufällig gerade ein Hardwareinterrupt ausgelöst wird, der zum echten „preemptive Contextswitch“ (CSW) zugunsten der Task TSx führt. SF2 wird also abgebrochen, damit TSx möglichst sofort starten kann. TSx ruft an ihrem dynamischen Ende die Terminate- (end)-Systemfunktion SFe auf. Der PU exekutiert einen Contextswitch, der jedoch nur fiktiv in den Prozeß TSa zurückführt, da sofort die abgebrochene Systemfunktion SF2 wieder in Bearbeitung genommen wird. Ein eventueller interner Kontext von SF2 wird bedingt durch ihre Bauart dabei neu erstellt - mußte also vorher nicht gerettet werden (s. u.).

Trotz der im Gegensatz zu anderen Betriebssystemen klaren Struktur von **RTOS–UH** gibt es auch hier natürlich Leistungsgrenzen.
So gibt es zwangsläufig durch die Maschinenbefehlssequenzen in den Supervisorprozessen oberhalb der „PU“-Linie Zeitabschnitte, in denen eine Umschaltung durch den „PU“ nicht möglich ist – z. B. bei der Systemfunktion „Request Semaphore“ zwischen Testen und Umsetzen der Variablen. Für die korrekte Funktion des Semaphorkonzeptes ist es sogar zwingend notwendig, daß der „PU“ diese beiden Operationen nicht teilen kann. Je länger aber jeweils die ungünstigste Zeitspanne einer solchen Behinderung des „PU“ ist, desto schlechter ist die Echtzeitqualität (Determiniertheit) des Systemes! Es gibt tatsächlich einige sogenannte Echtzeitsysteme, die hier je nach Nutzerprogrammsituation keine obere Grenze zu kennen scheinen und eigentlich in der Echtzeit-DV nicht eingesetzt werden dürften.

Die durch den „PU“ nicht aufspaltbaren Sequenzen sind in **RTOS–UH** im Prinzip vorher statisch auszählbar, hängen also z. B. nicht von der aktuellen Speicherbelegung ab. Auch die Suche nach Platz oder irgendwelchen Objekten im Speicher ist nach jeweils einer Handvoll Maschinenbefehle immer wieder für den „PU“ abbrechbar. Bei der Unterbrechung einer „SF“ durch den „PU“ gilt so eine Art „Throw-away“-Prinzip: Die „SF“ selbst haben keinen eigenen Kontext oder nur solchen, der bei Neubeginn der „SF“ von alleine wiederkehrt; was die „SF“ bis zum Abbruch geleistet hat, wird einfach bei Wiederaufnahme der verdrängten Task wiederholt. Dadurch entsteht theoretisch natürlich ein Verlust von Prozessorarbeitsleistung. Er ist jedoch in der Praxis kaum nachzuweisen. Lediglich beim Labortestbetrieb mit Signalgenerator und zyklischen Interrupts im Frequenzbereich der (hohen) Systemleistungsgrenze beobachtet man verfahrensbedingte charakteristische Phänomene. Man beachte, daß die „verworfene Arbeit“ ja stets dem minderwichtigen, zu verdrängenden Prozeß aufgehalst wird. Diese Technik wurde in den modernen Versionen von **RTOS–UH** ständig weiter perfektioniert und ist sicher einer der Gründe für die sehr gute Phasentreue, Determiniertheit und schnelle Reaktivität der aktuellen Implementierungen.

Die Abbrechbarkeit von Systemfunktionen ist für den Regelungstechniker zwingend. Multiusersysteme, wie z. B. normales Unix oder gewisse OS-x haben trotz ihrer sonstigen Qualitäten hier ganz gravierende konzeptionelle Mängel, die sie für typische anspruchsvolle Regelaufgaben ungeeignet machen: Irgendeine niederpriore Task veranlaßt eine Terminalausgabe, ruft dazu eine „SF“ auf, und schon ist der Reglerzyklus völlig zerstört, weil der Timerinterrupt erst am Ende der SF zum Taskwechsel führt. Das große Gefahrenpotential durch die Verwendung solcher Systeme wird hier sehr deutlich.

In **RTOS–UH** liegen oberhalb des Prozeßumschalters folgende Supervisorprozesse:

IS	=	Interruptsperre, durch Software ein-/ausgeschaltet
IR	=	Interruptroutinen, durch Prozessorhardw. aktiviert
SF	=	Systemfunktion, durch Software-IR/Trap aktiviert

Bei unserem System wird aus einsichtigen Gründen generell die Strategie verfolgt, möglichst wenig Aktionen von den Supervisorprozessen ausführen zu lassen. Wie an dem Prozeß-/Zeitdiagramm deutlich wird und oben schon gesagt wurde, entziehen sie den Nutzerprozessen Prozessorleistung und gefährden die Determiniertheit von Anwenderprogrammen. Sie sind deswegen quasi Störenfriede im System. Im **RTOS–UH** ist darum die einzige Aufgabe der IR-Prozesse die Veränderung des Laufzustandes von Nutzerprozessen. Haben sie den Laufzustand irgendeines Nutzerprozesses geändert, dann hinterlassen sie eine Notiz in einer zentralen Sammelflag. Gleiches tun auch die „SF“-Prozesse, jedoch finden hier im Gegensatz zu den IR-Prozessen noch andere Aktionen (Speichersuche etc.) statt.

Der **RTOS–UH**-Kern prüft bei jedem „Abstieg“ vom Supervisor- in den Userstatus auf der PU-Ebene die oben erwähnte Sammelflag, in der jede zwischenzeitliche Taskzustandsänderung archiviert wurde. Der PU selbst ist der niedrigst priorisierte Supervisorprozess des Systemes, läuft also im privilegierten Mode des Prozessors mit vollem Instruktionssatz. Da auch die Aufgaben des PU sehr langatmig werden können, ist er ebenfalls so konstruiert, daß ein preemptiver Contextswitch möglich ist.

Wichtiger Hinweis:

Das Interruptsystem des Prozessors ist in **RTOS–UH** bis auf sehr kurze Sequenzen immer offen. So können jederzeit Alarme oder Dateninterrupts das System erreichen. Der Aufruf einer SF durch eine Nutzertask gefährdet also auch diese Art von Echtzeitreaktivität nicht. Grundsätzlich kann auch der Anwender eigene IR-Prozesse zum System hinzufügen, es ist aber unbedingt erforderlich, daß er sich an die Konventionen hält und seine IR-Prozesse so gestaltet, daß sie nach möglichst wenig Maschinenzyklen (z. B. 20) ihren Ausgang (über die PU-Ebene!!) nehmen können.

1.4 Das I/O-System

Die Ein- oder Ausgabe unter Verwendung von Geräten, die langsamer sind als der Prozessor, ist eine Aufgabe des Betriebssystemes. Solche Geräte sind z. B. Drucker, Plattenspeicher, ein Fenster auf dem Monitor oder auch eine ferne Station im Netz.

Eine Besonderheit des **RTOS–UH** liegt hier in der Verwendung besonderer Input- und Output-Tasks. Diese Tasks sind quasi dienstbare Geister des Systemes und werden darum hier auch „I/O–Dämonen“ genannt. Eine Task, die etwas drucken möchte, füllt dazu ein sogenanntes „Communication–Element“ mit Text und Verarbeitungsvorschriften und übergibt es an den Druckerdämonen. Der Druckerdämon kümmert sich fortan autonom um die Ausführung des Auftrages, während die auftraggebende Task schon wieder andere Aktivitäten entfalten kann; wenn sie will, kann sie allerdings auch auf das Ende des Druckvorganges warten, ohne dabei den Prozessor zu belasten.

Fast alle anderen Systeme haben hier nur eine prozedurale Schnittstelle, und die zugehörige Systemfunktion ist naturgemäß sehr zeitaufwendig. Die gewaltigen Nachteile aufwendiger Systemfunktionen wurden ja bereits in diesem Kapitel herausgearbeitet. In **RTOS–UH** aktivieren und versorgen die Systemfunktionen für die Ein- und Ausgabe dagegen nur den zuständigen Dämonen, der mit einer Priorität ganz knapp über seinem momentanen Auftraggeber eine nach Prioritäten geordnete Warteschlange abarbeitet.

Ist ein Dämon gerade mitten in einem Auftrag, während ein neuer wichtigerer Auftrag eingeht, so ändert das System sofort seine Priorität genau passend, um den Rest des alten Auftrages, so schnell es möglich ist, zu Ende zu bringen. Damit ist eine Irritation hochpriorer Tasks durch Ein- Ausgabevorgänge niedrig priorer Prozesse in **RTOS–UH** auf ein kaum noch zu unterbietendes Maß reduziert. Praktisch tritt eine Beeinflussung meist nur noch dann auf, wenn hoch- und niederpriore Tasks Geräte gemeinsam benutzen, weil **RTOS–UH** die Aufträge aus guten Gründen nicht beliebig fein zerstückelt.

So ist erklärlich, weshalb sogar Reglertasks mit Abtastzyklen im Millisekundenbereich zeitkonform arbeiten können, während andere niedriger priorisierte Tasks umfangreiche Datensätze von der Platte lesen oder dorthin schreiben.

(Leere Seite vor neuem Kapitel)

Kapitel 2: Betriebssystem RTOS–UH

2.1 Schnellkurs Teil 1: Erste Schritte

2.1.1 Einschalten

Systeme mit Monitorprogramm erwarten nach Meldung des Monitors die Eingabe eines systemabhängigen „Boot“-Befehles, andere starten **RTOS–UH** aus dem EPROM oder booten automatisch.

Es dauert bis zu 20 Sekunden, bevor überhaupt etwas auf dem Sichtgerät passiert. In dieser Zeit konfiguriert sich das Betriebssystem (automatic linking) an Hand der im EPROM oder Bootbereich zusammengestellten Systemkomponenten. Passiert auch nach längerer Zeit nichts, so muß die Hardware (Baudrate richtig?) untersucht werden. Ansonsten erscheint die Kopfzeile mit Konfigurations– und Lizenzhinweisen, abgeschlossen mit der Meldung **RESET**. Wann immer **RESET** erscheint, handelt es sich um einen sogenannten „Kaltstart“.

Nach dieser Reset-Meldung ist das System im leeren Grundzustand. Es kann so konfiguriert sein, daß es nun sofort beginnt, auf der Platte nach Initialisierungsfiles zu suchen. Diese stehen meistens im Ordner `/HO/AUTO` und heißen `AUTOC`, `AUTOCx` und `AUTOW` sowie `AUTOWx`. Dabei steht `C` für den Kaltstart- und `W` für den Warmstartvorgang. Mit `x` ist die Nummer (1, 2, 3 ..) der jeweils einzurichtenden Nutzerarbeitsplätze gemeint. Mit diesen Initialisierungsfiles in Shellsprache kann bei Bedarf natürlich auch automatisch Anwendungssoftware geladen werden.

„Leerer Grundzustand“ bedeutet aber auch, daß eventuelle Editordateien im Speicher oder dort bisher abgelegte Daten und Programme verloren sind. Auch nach einem „System Abort“ nimmt das System diesen Zustand automatisch ein, wenn vorher durch fehlerhafte Programme wichtige **RTOS–UH**-eigene Daten inkonsistent geworden sind. In solch einem Fall sollte man bei nicht ROM-residenten Systemen besser neu booten.

2.1.2 Erste Aktion

Im Gegensatz zu den meisten anderen Betriebssystemen passiert jedoch bei Anschlag eines Zeichens auf der Tastatur zunächst gar nichts. Hier muß man sich mit der ersten Besonderheit von **RTOS–UH** vertraut machen: Der Nutzer mit seiner Tastatur ist nicht der Herrscher im System, sondern nur ein von Zeit zu Zeit störendes externes Ereignis, auf das **RTOS–UH** prioritätsgerecht reagieren wird. Was aber tut der Prozessor zur Zeit? Der Prozeßumschalter hat festgestellt, daß die einzige lauffähige „Task“ die Leerlaufaktivität `#IDLE` ist und läßt diese arbeiten. `#IDLE` ist immer lauffähig. Um den Prozessor von seiner `#IDLE`–Task zu trennen, erzeugen wir einen Interrupt durch Anschlag von

`Ctrl A` (Bei gedrückter Ctrl-Taste Taste A anschlagen)

Auch mit `Ctrl B` oder `Ctrl C` oder `BREAK/Undo` kann der Interrupt erzeugt werden. Diese Befehle haben aber eine besondere Bedeutung. Mit `Ctrl B` entscheidet man sich für eine Eingabe auf der seriellen Schnittstelle im B*x*-Mode (siehe dazu Seite 389). Mit Hilfe von `BREAK/Undo` kann man Bedienbefehle selbst unterbrechen oder sich in verfranster Situation, z. B. im Editor, mit einem „Notruf“ der Shell wieder befreien. Mit Ausnahme von `Ctrl C` oder `Break/Undo`, dort dauert es etwa 2 Sekunden, erscheint sofort nach Anschlag der Prompt „`*`“ und zeigt Eingabebereitschaft an.

! → Wann immer wir Bedienfunktionen des Rechners benötigen, muß eine der obigen Tasten, i. a. `Ctrl A`, angeschlagen werden.

Mit dem Erscheinen des „`*`“ übergibt der Dispatcher (Prozeßumschalter, PU) gleich wieder an „`#IDLE`“. Dennoch können wir jetzt eine Eingabe machen. Wir sind nämlich mit dem Bedieninterpreter (Shell) des Systemes verbunden.

! → Die Sprache des Interpreters darf nicht verwechselt werden mit einer PEARL–Programmierung, auch wenn es viele ganz ähnliche oder gar gleiche Anweisungen gibt.

Die nun zulässigen Eingaben werden in einem eigenen Kapitel später beschrieben. Wir wählen das Kommando

```
L
```

und schließen es mit Carriagereturn (CR) ab. Dieses Kommando bedeutet „List all tasks“. Möglich wäre dann etwa folgende Ausgabe:

```
00001986 +FFF/1 RUN  TWS=00000D36 PC=00081192 #IDLE
000019D0 -001/1 DORM TWS=00000D92 PC=00000000 #EDFMN
00001A1A -00A/1 SCHD TWS=00000E00 PC=00081DEC #ERROR
00001AAE -007/1 RUN  TWS=00000EB8 PC=00082C84 #USER1
00001AF8 -006/2 DORM TWS=000010CE PC=00000000 #USER2
00001B42 -002/1 DORM TWS=000012E4 PC=00000000 #XCMMD
00001B8C -001/1 DORM TWS=000014FA PC=0008C9A0 #UHFX6
00001BD6 -005/1 RUN  TWS=000016B0 PC=0008F4A8 #ACIA1
00001C20 -005/1 DORM TWS=00001716 PC=00000000 #ACIA2
```

Dabei bezeichnet die erste Adresse den Ort, an dem der sogenannte „Taskkopf“ im verwalteten RAM zu finden ist. In der zweiten Spalte finden wir Priorität/Usernummer des Prozesses. Negative Prioritäten sind für den Hochsprachprogrammierer unerreichbar hoch. Der angezeigte Laufzustand (`RUN`, `DORM`, `SUSP` etc.) ist beim `L`–Kommando genauer beschrieben.

2.1.2.1 Groß oder klein?

! → Im Gegensatz zum PEARL–Compiler akzeptiert der Bedienterpreter alle Kommandos sowohl in Groß– als auch in Kleinschreibung. Aber man beachte dabei, daß Filenamen immer gleich geschrieben werden müssen! Groß definiert — groß referenziert.

Nun können wir z. B. den Befehl `S` eingeben (natürlich erst nach erneutem Anschlag von `Ctrl A`!) und sehen, wie der verwaltete Speicherbereich aufgeteilt ist.

2.1.3 PEARL–Programmentwicklung

Folgende Aufgabe sei angenommen: Ein Rechenprozeß („Task“) soll fortwährend die Zahlen 1.0, 2.0, 3.0, 4.0, etc. sowie deren Kehrwerte und Quadrate ausgeben. Ein zweiter unabhängiger Rechenprozeß soll jede Sekunde in einem File notieren, welcher Wert gerade bearbeitet wird und den ersten Prozeß nach 100 Sekunden anhalten.

2.1.3.1 Festlegung eines „Working Directories“ („WD“)

Um für den Editor, Compiler, Lader usw. einfachere Ansprechwege zu erschließen, geben wir den Befehl „Change Directory“ ein:

`CD /ED` Der Slash bedeutet: „`ED`“ ist eine Datenstation. (Nur zwecks Abwärtskompatibilität akzeptiert das System vorläufig auch noch veraltete syntaktische Konstrukte. Aus früherer Zeit stammen `CD ED.` und `CD ED:`.)

(Nach `Ctrl A`, wie immer vor einer Bedienbefehlseingabe ...). Der Befehl wird vom System bestätigt mit:

```
WD=/ED/-  aktuelles Working Directory=/ED/
XD=/-     kein Execution Directory
```

Die Vereinbarung bleibt bis zum neuen `CD`–Befehl oder „Kaltstart“ erhalten.

2.1.3.2 Festlegung eines Execution Directory

Für unsere kleine Einführungsaufgabe benötigen wir es nicht, aber es sei hier erwähnt: neben dem Working Directory existieren noch ein oder mehrere Execution Directories. Das sind Verzeichnisse, in denen die Bedienkomponente des Systemes, die „Shell“, nach ihr unbekannten Bedienbefehlen sucht. Die Änderung des Executingdirectories erfolgt mittels der Anweisung: `CXD` *devpath* — es erfolgt dann auch die Ausgabe der aktuellen Einstellung.

2.1.3.3 Einloggen in den Editor

Wir geben das Kommando `ED` (nach `Ctrl A`!) ein und finden uns im Nu im Bildschirmeditor des Systemes wieder. Der Filename wurde vom System „defaultiert“. Wir können (s. `ED`–Kommando) aber auch einen frei gewählten Filenamen wählen, etwa `ED TEST`.

Erscheint ein chaotisches Bild, so geben Sie gleich die Zeichen `ESC X` (nacheinander anschlagen, ohne `Ctrl A`) ein, um den Editor sofort wieder zu verlassen. Wir versuchen eine Umparametrierung mit Hilfe des Bedieninterpreters:

`SD /A1/+1 01` oder
`SD /A1/+1 02` oder
`SD /A1/+1 03`

und loggen erneut mit `ED` (bzw. `ED TEST`) in den Editor ein.

Hilft auch das nicht, so muß ggf. das Terminal auf den Televideo oder VT–52 kompatiblen Mode umgestellt werden.

Standardmäßig ist der Editor auf den Televideo–Typ eingestellt. Solange wir im Editor sind, ist übrigens der Zugang zum Bedieninterpreter verwehrt, weil die Zeichen `Ctrl A`, `Ctrl B` und `Ctrl C` vom Editor abgefangen werden. Nur `Break/Undo` ist für den größten Notfall, wenn man sich total verfranst haben sollte, noch aktivierbar. Radikaler ist die Abort–Taste der CPU. Wir löschen die erste Zeile mit `ESC R` (siehe Beschreibung des ED-Befehles auf Seite 136) und geben ein:

```
MODULE PROBE;          /* Bezeichner sind max. 24 Zeichen lang*/;
SYSTEM;           /* Leitet Definition der Datenstationen ein */;
  Disp: A1<->;               /* Datenstation der Ein-Ausgabe */;
  File: /ED/Daten->;           /* File zum Sammeln der Daten */;
PROBLEM;        /* Leitet den Hardware-unabhaengigen Teil ein */;
  SPECIFY Disp DATION INOUT ALPHIC CONTROL(ALL);
  SPC File DATION OUT ALPHIC CONTROL(ALL);
  DCL Ausgabenummer FIXED;   /* Zaehler fuer Anzahl Ausdruecke*/;
  DCL (x,Leer) FLOAT;          /* Aktuelles Argument, Leerzyk*/;
  DCL Erstesmal BIT(1);        /* Fuer Startvorgang */;
/*-------------------------------------------------------------*/;
Machs:TASK PRIO 50;                     /* Ausfuehrende Task */;
   x=1.0;                                /* Startwert setzen */;
   REPEAT;                   /* Unbegrenzte Wiederholschleife */;
      PUT x, x*x, 1.0/x TO Disp BY SKIP,(2)F(20),E(20,7);
      x= x + 1.0;
   END;                                      /* Schleifenende */;
END;                                  /* Ende der Task Machs */;
/*-------------------------------------------------------------*/;
Steuer:TASK PRIO 49;                    /* Ueberwachende Task */;
   IF Erstesmal THEN
      ACTIVATE Machs;
      Ausgabenummer = 1;
      Erstesmal = NOT Erstesmal;
   FIN;                                    /* Ende If-Bereich */;
   PUT Ausgabenummer,x,Leer TO File BY F(8),(2)F(20),SKIP;
   IF Ausgabenummer EQ 100 THEN
      TERMINATE Machs;
      PREVENT Steuer;
   FIN;
   Ausgabenummer=Ausgabenummer+1;
END;                                 /* Ende der Task Steuer */;
/*-------------------------------------------------------------*/;
Rest: TASK PRIO 100;/* Niedrige Prio f.Restkapazitaetsmessung */;
   REPEAT Leer=Leer+1.0; END; END;       /* Keine Wartephasen */;
/*-------------------------------------------------------------*/;
start:TASK PRIO 48;                              /* Start-Task */;
   Erstesmal = '1'B;   Leer = 0.0;
   ALL 1 SEC ACTIVATE Steuer; ACTIVATE Rest;
END;                                  /* Ende der Start-Task */;
/*-------------------------------------------------------------*/;
MODEND; Ende des Modules.
```

Wir verlassen den Editor, indem wir nacheinander (!) die Zeichen `ESC` und `X` anschlagen. Im Gegensatz zum `Ctrl A`, bei dem die Ctrl-Taste nur eine Umschaltung der Tastatur bedeutet, ist `ESC` ein eigenes Zeichen. Eventuell können wir nun noch den File auf die Floppy oder die Platte retten (siehe `COPY`-Befehl auf Seite 115). Dann aktivieren wir die Bedientask (`Ctrl A`, wie üblich) und starten den Compiler:

```
P (CR)          |  /ED/SI kompilieren
P TEST (CR)     |  Editierter Filename war TEST.
```

Bei dieser Parametrierung erzeugt der Compiler das Übersetzungsprotokoll auf unserem Terminal. Genaueres über weitere Parameter (kein Protokoll etc.) finden Sie beim PEARL-Befehl auf Seite 180. Schon nach wenigen Sekunden ist das Programm übersetzt. Falls es Tippfehler gegeben hat — der Compiler zeigt sie uns an — müssen wir erneut mit `ED` bzw. `ED TEST` den Editor aufrufen. Das korrigierte Programm wird dann erneut übersetzt.

Irgendwann zeigt der Compilerlauf keinen Fehler mehr an. Jetzt können wir das ganze Modul in den Speicher laden:

```
LOAD (CR)          | Objektcode-Filename wurde defaultiert
```

Sofort meldet der Lader, daß er fertig ist, und wir geben das nächste Bedienkommando ein:

```
LU (CR)            | (Bedeutung: List User–Tasks.)
```

Nun erhalten wir eine Liste, in der unsere 4 Tasks aufgeführt sind. Um das Programm in Gang zu bringen, müssen wir nur noch die Task `start` aktivieren: (Wie stets vor jeder Bedienung auch hier vorher `Ctrl A` anschlagen ...)

```
start (CR)
```

Einen solchen Bedienbefehl gibt es nicht, wohl aber eine gleichnamige geladene Task. So weiß das Bediensystem, die Shell, in diesem Fall, was zu tun ist: die Task „start“ nimmt ihre Arbeit auf. Jetzt rasen Zahlen über den Bildschirm, aber nach 100 Sekunden ist alles wieder ruhig. Wir schauen uns den File `Daten` mit Hilfe des Editors an:

```
ED Daten      | oder
ED /ED/Daten | falls kein WD vereinbart
```

Es ist zu erkennen, daß die Task `Rest` offensichtlich wesentlich mehr (100 bis 300 mal!) Runden als die doch viel höher prioritierte Task `Machs` drehen konnte. Die Ursache liegt in einem verborgenen Wartezustand von `Machs`: Das Terminal ist zu langsam, und nach einer gewissen Anzahl Zeilen ist der Zwischenpuffer im **RTOS–UH** voll. Dann bremst **RTOS–UH** die Task `Machs` mit der Folge, daß der wartezustandfreie Rechenprozeß `Rest` blendend bedient werden kann. Wenn Sie sich jetzt wundern, weil sie den eben beschriebenen Effekt nicht beobachten konnten, dann haben Sie ein System mit emuliertem Terminal. Bei solchen Systemen kommt `Rest` nicht von der Stelle, weil die CPU voll fürs „Malen“ beansprucht wird!!

Jetzt können Sie mit dem Programm ein paar Experimente machen. Sie können z. B. die Ausgabe aus `Machs` entfernen, um dann den Versuch zu wiederholen. Dazu müssen Sie jedoch zunächst das Modul (Name: `PROBE`) aus dem System entfernen. (Nebenbei: `Rest` geht's inzwischen noch blendender, sie läuft und läuft...) Das Entfernen dieses Modules erledigen wir mit:

`UNLOAD PROBE*` (`*` = „Inklusive aller Tasks“)

Unschön, wenn wir mal den Stern vergessen haben sollten. Dann müssen wir nämlich anschließend jede einzelne Task namentlich mit `UNLOAD` entfernen, z. B. `Unload Machs,Rest,Steuer,start (CR)`.

Wir loggen nun einfach erneut mit `ED` bzw. mit `ED TEST` in unseren Quellfile ein, ändern diesen, übersetzen ihn neu und laden ihn genau wie vorhin. Wichtig ist, daß wir vor dem Laden wirklich das Modul entfernt haben, ggf. mit dem `LU`–Befehl überprüfen, ob alle Tasks verschwunden sind.

Bevor wir nun ein zweites Mal `start` aktivieren, sollten wir den Ausgabefile `Daten` zurücksetzen, z. B. mit

`REWIND Daten`	oder
`rewind Daten`	oder, falls kein Work.-Dir. vereinbart
`REWIND /ED/Daten`	bzw. gleichwertig dazu:
`rewind /ed/Daten`	Statt „`Daten`“ jedoch nicht „`DATEN`“ oder „`daten`“

oder durch Elimination des ganzen Files:

`RM Daten`	oder
`ERASE Daten`	

Sonst würden die neu produzierten Daten hinter die alten gesetzt. Jetzt können wir wieder die Task `start` aktivieren. Damit ist ein kompletter Programmentwicklungszyklus abgeschlossen.

2.1.4 Retten des Programmes auf Platte oder Diskette

Jetzt darf kein Netzausfall passieren, sonst ist das ganze erstellte Programm unrettbar verloren. Aus diesem Grund legen wir nun eine Diskette (hier: Laufwerk 0) ein und formatieren sie neu:

`RTOSFILES /F0`		`MSFILES /F0`
`FORM D /F0/B5DS80`	oder	`FORM D /F0/C5DS80`

Das rechte Anweisungspärchen gilt für den Fall, daß aus irgendwelchen Gründen nicht das RTOS-UH-eigene, sondern das „kompatible“ PC-Format gewählt werden soll. Dabei steht `D` für Double density, `B5` für das kompakteste „B“–Format und `C5` für das weniger effiziente aber kompatible „C“–Format. `DS` bedeutet „Double sided“ und `80` steht hier für 80 Tracks. Die Formatierungsinformation wird beim `FORM` wie ein Filename übergeben.

Wir sind nun ausnahmsweise für eine ganze Weile aus dem System ausgeblendet, weil das Formatieren bei den meisten Systemen eine sehr zeitkritische Angelegenheit ist. Zunächst werden nacheinander beide Seiten beschrieben, dann wird nach defekten Blöcken gesucht — diese werden aus der Verwaltung gestrichen — und, das Ergebnis wird auf dem Terminal ausgegeben. Bei mehr als 9 defekten Blöcken allerdings wird die Bearbeitung der Diskette abgebrochen.

Meist werden Sie eine Platte in Ihrem System haben, die bereits formatiert ist. Dann kann man das kleine Testprogramm statt auf Diskette auch erst einmal dort ablegen. Dazu ist in den folgenden Befehlen „`/F0`“ durch den Plattenbezeichner, z. B. „`/H0`“ zu ersetzen.

Nun können wir mit dem Befehl

`DIR /F0` („/..“ gibt an: Kein File, sondern Gerät)

nachsehen, wieviel Platz wir haben.

Das Retten unseres Programmes erledigen wir mit `COPY`:

```
COPY /ED/SI>/F0/xyz   (Def. Filename für User1)
COPY TEST>/F0/xyz     (File-name war TEST)
```

Später können wir den File jederzeit wieder von der Diskette holen, etwa mit

```
COPY /F0/xyz > /ED/TEST   (kein Work.-Dir.)
COPY /F0/xyz > /A1/       (Ausgabe auf Terminal)
```

2.1.4.1 Files geschlossen?

Zur Einsparung von Platten- und Diskettenoperationen werden, solange es geht, das Inhaltsverzeichnis und der aktuelle Datenfile teilweise im Speicher gehalten. Die Daten sind daher nur dann auf dem Medium gesichert, wenn alle Files vor dem Herausnehmen geschlossen sind. Geben Sie dem System eine Chance, prüfen Sie, ob Sie die Diskette herausnehmen oder den Rechner abschalten dürfen:

`CF /F0/` Change Floppy, erzeugt eine Warnung, falls ein File noch offen ist.
`FILES /F0/` Listet alle offenen Files auf, anschließend wird mit
`RETURN /F0/xyz,/F0/abcd` jeder einzelne File geschlossen oder
`SYNC /F0/; CF /F0/FORGET` eingegeben. Nun erst Diskette entnehmen.

Auch auf einer Festplatte müssen alle Files geschlossen sein, bevor Sie den Rechner ausschalten oder Abort/Reset drücken. Ansonsten kann es passieren, daß langsam das gesamte File–System zerstört wird und alle Daten verlorengehen! Dann sollten Sie retten, was zu retten ist, und neu formatieren. Der Lader, Compiler, Copy etc. hinterlassen normalerweise keine offenen Files. Sie treten darum nur bei Unregelmäßigkeiten und abgebrochenen Programmen auf.

2.1.5 Zeit sparen durch Multitasking

Wenn Sie etwas experimentiert haben, so werden Sie schon bemerkt haben, daß unser System anscheinend tausend Dinge gleichzeitig erledigen kann. Sie können z. B. einen `COPY` auf die Schnittstelle `/A2/` oder `/PP/` (Centronics) in Gang setzen, und während die Ausgabe läuft, ungehindert das nächste Programm entwickeln. Die bei anderen Systemen oft zu findenden „Spooler“ sind auf Grund der kompromißlosen Multitasking-Architektur bei **RTOS–UH** überflüssig. Man muß nur immer darauf achten, daß stets nur eine Operation pro File stattfinden darf, also nicht den gleichen File sowohl im Editor haben als auch gleichzeitig kompilieren oder kopieren!

Dagegen können Sie ruhig während der Kompilation eines Files den Kompilationslauf eines anderen Files in Gang setzen oder diesen anderen editieren und vielleicht zwanzig andere assemblieren...

2.1.6 Das Bediensystem in Kürze

Bisher haben wir immer nur eine Anweisung in jeder Zeile benutzt. Wir können durchaus mehrere Anweisungen in eine Zeile schreiben und dabei festlegen, ob diese „gleichzeitig“ oder nacheinander zu erledigen sind:

```
P TEST -- load -- start
```

bedeutet: Kompiliere, wenn fertig (und 0 Fehler!) lade, wenn mit Laden erfolgreich fertig, aktiviere `start`. Der Sinn der zeitlichen Kettung besteht darin, daß man vielleicht gerne mal irgendwohin möchte und das Programm in Aktion sehen will, wenn man zurück kommt...

Dagegen werden bei dem Befehl

```
P TEST;COPY mist>/A2/;COPY kaese>/PP/
```

gleichzeitig ein Compiler und zwei Kopiervorgänge lauffähig gemacht. Nebenbei: `/A2` ist die Stationsbezeichnung für die zweite serielle Schnittstelle, `/PP` die Stationsbezeichnung für ein evtl. vorhandenes Parallelport (Centronics).

2.1.6.1 Fernsteuerung

Alle Befehle des Bedieninterpreters können auch von PEARL-Programmen aus ausgeführt werden! Sie brauchen dazu nur den Eingabetext mit "`PUT`" in die Systemdatenstation „`/XC`" (Remote-Control) einzuschreiben. Auch wenn es dank der Shellsprache und bestimmter Unterprogramme (wie z.B. `EXEC`) hierzu auch noch andere und elegantere Lösungen gibt, betrachten wir einmal ein PEARL-Beispiel, bei dem der Rechner sich um 16.00 Uhr selbst das Bedienkommando zur Anzeige aller Files auf dem Plattenmedium `/H0` „eingibt":

```
MODULE TEST;
SYSTEM;
  Bedien:/XC;
PROBLEM;
  SPC Bedien DATION OUT ALPHIC;
Plattenschau:TASK;
  PUT 'DIR /H0' TO Bedien BY A,SKIP;
END;
MODEND;
```

Nach dem Übersetzen und Laden geben Sie den Bedienbefehl

```
AT 16.00 ACTIVATE Plattenschau
```

ein und vergessen die Sache zunächst einmal.

Wir hoffen nun doch sehr, daß Sie solche „Zeitbomben“ nicht mutwillig für den nächsten Nutzer im Rechner zurücklassen und einen unerfahrenen Nachfolger mit solchen oder schlimmeren schlafenden Bedienbefehlen erschrecken. Allerdings kann man als Nutzer jederzeit sicherstellen, daß sich keine ungewollten Einplanungen mehr im System befinden. Aus Sicherheitsgründen gibt es nämlich keine Möglichkeit, solche Prozesse vor dem „`L`“ oder „`LU`“–Kommando zu verstecken. Eine sinnvollere Möglichkeit des `/XC` (bzw. der `EXEC`-Routine) besteht in der Kreation von Blockkommandos:

```
...
Neu:TASK;
PUT 'Unload PROBE*;WE--P--Load--start' TO Bedien;
END;
...
```

Nach dem Laden dieser Task braucht man nur noch jeweils den Befehl `Neu` einzugeben (auch während `Machs` läuft!) und findet sich im Window-Editor wieder. Sobald man diesen verläßt, wird kompiliert, geladen und gestartet.

RTOS–UH bietet allerdings auch hierzu noch andere Möglichkeiten, zum Beispiel mit Hilfe des `DEFINE`-Befehles (siehe Seite 126) oder durch ein sogenanntes „Skript“ in der Shellsprache.

2.1.6.2 Weitere Nutzer

An anderen Terminals, die ebenfalls einen Bedieninterfaceanschluß besitzen, ist die Bedienung nicht anders. Allerdings sollte man bei mehr als einem Nutzer sich hierarchisch in das ED–Filesystem einloggen. Sonst riskieren Sie zufällige Namensgleichheit bei den Files.

```
CD /ED/Mueller
```

Mit z. B. `ED TEST` wird nun mit voller Pathlist `/ED/Mueller/TEST` adressiert. Wenn der andere Nutzer nicht auch „Mueller“ heißt und sich unter anderem Namen einführt, ist sein File `TEST` einer im anderen Zweig des Baumes.

Das Konsolenterminal (User 1) hat allerdings vereinzelt doch einen Sonderstatus, z. B. bei Fehlermeldungen in Interruptprozessen.

2.1.6.3 Haben Sie eine Festplatte?

Bei größeren Speichermedien (Festplatte oder Wechselplatte) sollten Sie Ihre Files immer nur hierarchisch organisieren, sonst stehen hinterher vielleicht 200 Files in der Root–Ebene, und Sie finden sich garantiert nicht mehr zurecht. Man lege sich dazu „Ordner“, auch „Unterverzeichnisse“ oder „Subdirectories“ genannt, mit Hilfe des `MKDIR`–Befehles (Beschreibung auf Seite 175) an:

```
mkdir /H0/usr
mkdir /H0/usr/mueller
```

Fortan können Sie etwa nach `/H0/usr/mueller/TEST` schreiben oder von dort lesen. Greifen Sie häufiger darauf zu, so kann natürlich mit

```
CD /H0/usr/mueller
```

der Zugriff vereinfacht werden. Leider müssen Sie jetzt aber beim Editieren die ED–Files über die volle pathlist ansprechen. Wenn wir also den File `/H0/usr/mueller/TEST` in Bearbeitung haben, könnte eine Kommandozeile wie folgt uns weiterhelfen:

```
copy TEST > /ED/TEST -- ed /ED/TEST -- copy /ED/TEST > TEST
```

Bei diesem Befehl wird der File von der Festplatte geholt, editiert und anschließend wieder zurückgeschrieben.

2.1.6.4 Ein- und Ausgabe von Daten ganz allgemein.

Im folgenden Teil 2 des Schnellkurses finden Sie noch einige Tabellen zu den Schnittstellen. Die Schnittstellen selbst sind einzeln im Kapitel 6 ab Seite 389 beschrieben. Es ist sehr wahrscheinlich, daß Ihr System noch weitere Ein- oder Ausgabeschnittstellen besitzt, zu denen Sie zusätzliche Beschreibungen zur Ergänzung des Handbuches erhalten haben, z. B. für den Windowmanager.

2.1.7 Empfehlung für das weitere Anlernen

Studieren Sie bitte zunächst noch die beiden folgenden Teile 2 und 3 des Schnellkurses. Kurz, aber enorm wichtig, ist der Teil über die Bedienfehlerteufel. Schließlich sind in allen komplexen Computersystemen einige typische Bedienfehler trotz aller Sorgfalt der Nutzer nie ganz auszuschließen. Ab Seite 44 haben wir die von uns beobachteten häufigsten Fehlbediensituationen beschrieben.

Lesen Sie sich doch alle Befehle des Bedieninterpreters einmal in Ruhe durch! Sie finden dort die detaillierte Information, die hier keinen Platz fand. Einige Bedienbefehle haben Sie ja schon benutzt, wenn auch z. T. nur mit einem Teil der möglichen Parameter. Zur Sprache PEARL selbst finden Sie in diesem Handbuch nur wenige Angaben. Dazu empfehlen wir Ihnen das PEARL90 Referenzbuch, welches die Fachgruppe 4.4.2 „Echtzeitprogrammierung PEARL“ der Gesellschaft für Informatik bereithält, oder eines der PEARL-Lehrbücher.

Auch wenn heute scheinbar alles über Windowtechnik erledigt wird und Sie das **RTOS–UH**-eigene Multiwindowsystem benutzen: eine Beschäftigung mit der Shellsprache ist weiterhin lohnend! Die Technik der Skripte ermöglicht im **RTOS–UH** nämlich verblüffend einfach die Steuerung von Fensteroperationen u. ä.

2.2 Schnellkurs Teil 2: Schnittstellen und Dations

Grundsätzlich existieren zwei Typen von Datenstationen, je nachdem ob sie annähernd mit Prozessorgeschwindigkeit die Daten übertragen können oder nicht.

- D/A–Wandler, Digitalkoppler etc. sind quasi jederzeit bereit. Sie werden darum nicht vom Betriebssystem als Betriebsmittel verwaltet, der Compilercode greift direkt darauf zu.
- Terminalschnittstellen, Druckerports, Floppydisks etc. sind langsamer als der Prozessor und werden daher nur unter Kontrolle von **RTOS–UH** zugänglich gemacht. Die Bearbeitung erfolgt ähnlich wie am Postschalter: Es werden „Warteschlangen“ aufgebaut, und zu einem bestimmten Zeitpunkt wird nur das jeweils vorne befindliche Datenpaket bearbeitet.

Die folgende Beschreibung gilt demnach nur für die Stationen des zweiten Types mit Warteschlangen.

Zu jeder Station existiert eine Warteschlangennummer, `LDN` genannt („Logical Dation–Number“). Benutzen mehrere anscheinend eigenständige Geräte (z. B. Floppy 0, Floppy 1) gemeinsame Bausteine (Floppykoppler), so werden sie dennoch nur durch eine einzige `LDN` repräsentiert. Die einzelnen Geräte werden dann durch die sog. Untergliederungsnummer (`DRIVE`) unterschieden. Oft wird die Untergliederungsnummer auch nur benutzt, um verschiedene Betriebsarten eines einzigen Gerätes anzuwählen. Die nächst feinere Unterteilung erfolgt durch den „FILE–Namen“ oder noch genauer durch eine baumförmige „Path–List“.

Jede Warteschlange besitzt einen „Bediener“, quasi der Beamte am Postschalter. Wir nennen diesen Bediener die „Betreuungstask“ der Warteschlange. Auch die Bezeichnung „I/O-Dämon“ haben wir für diesen autonomen guten Geist schon kennengelernt. Er ist ein Dämon, weil er Betriebssystemaufgaben zur Wahrung der Systemreaktivität in die Welt der Nutzerprozesse verlagert. Die Aufgabe der Betreuungstask bzw. dieses Dämonen besteht einfach nur darin, die Warteschlange möglichst schnell abzubauen. Wie im täglichen Leben gibt es auch hier „ganz eilige Kunden“, die sich frech nach vorne drängen, evtl. direkt bis zum Schalter. Wie weit man sich vordrängen kann, hängt von der Wichtigkeit des Auftraggebers ab: der Priorität der lese–/schreibwilligen Task. (Der Name des I/O-Dämonen ist für den Anwender uninteressant, er ist meist analog zur Gerätebezeichnung gewählt.)

In PEARL–Programmen kann man nun den einzelnen Geräten frei wählbare Namen zuordnen. Das geschieht mit Hilfe des „Systemteiles“.

So bedeutet etwa

```
Drucker:/LP ;
```

folgendes: Im Programm wird das Symbol „Drucker“ verwendet, als physikalisches Gerät wird das Gerät „`/LP`“ des zur Laufzeit benutzten Betriebssystemes verwendet. Ein solches Gerät muß dort vorhanden sein, sonst erfolgt beim Laden des Programmes eine Fehlermeldung. Es ist aber auch möglich, Datenstationen direkt über ihre `LDN` und Laufwerksnummer anzusprechen:

```
XYZ:LD/5.3/abcd/efg ;                    ! adressiert LDN 5, Drive 3
```

Die Station heißt `XYZ`, besitzt die Warteschlangennummer `5` und benutzt das Laufwerk Nummer `3`. Über das „Directory“ namens „abcd“ wird der File „efg“ adressiert. Man beachte, daß die pathlist bei Bedarf auch noch während des Programmlaufes ganz (d.h. inklusive Gerätebezeichner) oder im Filebezeichnerteil durch das PEARL-statement „`OPEN BY IDF...`“ verändert werden kann. Dabei kann auch das aktuelle Working-Directory mit berücksichtigt werden. Durch weitere Zusätze kann auch der Betriebsmode modifiziert werden.

`PUT` und `GET` benutzen stets solche Stationen mit Warteschlangen.

Für alle Stationen gibt es im Bediensystem Bezeichner, zu denen `LDN` und `DRIVE` automatisch generiert werden, z. B. `/A1` für das Konsolenterminal = `LD/0.0/`. Im Systemteil des Compilers darf man auf der rechten Seite der Namenszuordnung auch Bezeichner verwenden, die der Compiler nicht kennt und/oder die im aktuellen Entwicklungssystem dem Bediensystem nicht bekannt sind. Solche Bezeichner listet der Compiler unter „Extra Devices“ bei der Modulbilanz auf. Sie müssen später im Zielsystem beim Laden vorhanden sein, sonst verursacht ihre Ansprache einen Laufzeitfehler. Auch beim Linken solcher Module können dem Linker die Vereinbarungen über Datenstationen des Zielsystemes mitgegeben werden. Der Lader warnt beim Fehlen einer Station, sodaß derartige Fehler nicht erst später bei der echten Benutzung der Station, sondern schon in der Entwicklungsphase des Programmes erkennbar sind.

Folgende Standarddatenstationen sind in allen Systemen enthalten:

Mnemo	LDN	DRIVE	Bemerkung
A1	0	0	1. Ser. Schnittstelle (A–Mode)
B1	0	2	1. Ser. – " – (B–Mode)
C1	0	6	1. Ser. – " – (C–Mode)
ED	1	0	EDFM–Filesystem (ASCII–Mode)
EDB	1	1	EDFM–Filesystem (Binär–Mode)
A2	2	0	2. Ser. Schnittstelle (A–Mode)
B2	2	2	2. Ser. – " – (B–Mode)
UL	2	3	2. Ser. – " – (UL–Mode)
C2	2	6	2. Ser. – " – (C–Mode)
VO	7	0	virtueller Out-channel (#VDATN)
VI	8	0	virtueller In-channel (#VDATN)
XC	9	0	external Commandprocessor
PP	10	0	parallel Port (falls #PPORT)
NIL	15	0	Nil–Dation

In Ihrem konkreten System sind sicherlich weitere `DATION`s vorhanden, ein typisches Beispiel:

Mnemo	LDN	DRIVE	Bemerkung
F0	3	0	1. Floppy
F1	3	1	2. Floppy
H0	3	2	1. Partition Harddisk
H1	3	3	2. Partition Harddisk
H2	3	4	3. Partition Harddisk
H3	3	5	4. Partition Harddisk

Bei manchen Systemen ist auf der CPU–Platine eine 3. serielle Schnittstelle vorhanden, dann gelten meist folgende Zuordnungen:

Mnemo	LDN	DRIVE	Bemerkung
A3	4	0	3. Ser. Schnittstelle (A–Mode)
B3	4	2	3. Ser. – " – (B–Mode)
C3	4	6	3. Ser. – " – (C–Mode)
D3	14	0	3. Ser. Schn. Duplexkanal (out)

Die Zuordnungen weiterer `DATION`s entnehmen Sie bitte den Unterlagen zu Ihrer Systemimplementation, oder benutzen Sie „`HELP-D`“.

2.3 Schnellkurs Teil 3: Typische Bedienungsfehler

Hier erfahren Sie etwas über die typischen Bedienungsfehlerteufel — soweit sie uns bekannt sind.

Der `Ctrl S`–Teufel
In diese Falle tappt man, wenn man aus Versehen statt `Ctrl A` einmal das Zeichen `Ctrl S` anschlägt. Dann kann der Fehlerteufel nämlich das Terminal totlegen. Das `Ctrl S` ist an sich sehr notwendig, um etwa rasende Ausgaben anhalten zu können (X_{off}) — und dafür benutzen wir es ja auch absichtlich. Befreien Sie sich aus der Situation und bieten Sie dem System `Ctrl Q` an (X_{on}), mehrmals schadet nichts.

Der „No–scroll"–Teufel
Er siedelt nur in einigen besonderen Terminals, die eine „No scroll"–Taste besitzen. Oft hilft dann `Ctrl Q` nicht, sondern nur die terminalseitige Aufhebung der No–scroll–Bedingung.

Der Input–queue–Teufel
Ein schwer zu bekämpfender Parasit, der den „Beamten" am Postschalter befällt, er kann den aktuellen Auftrag nicht ausführen. Eine Task veranlaßt zum Beispiel eine Leseoperation (`GET`) vom Terminal. Solange auf dem Gerät die erwartete Zahl von Zeichen nicht eingegeben wird, geht's in der Schlange nicht voran. Trotz der hohen Priorität kommt auch die Bedientask nicht durch. Scheinbar reagiert das System auf `Ctrl A` nicht. Uns bleibt nichts anderes zu tun als `Ctrl A` anzuschlagen, evtl. die Daten einzugeben oder ersatzweise so oft die Carriagereturn–Taste zu betätigen, bis der Eingabeprompt der Shell erscheint.

Der Index–error–Teufel
Diese Art von Programmfehlern ist mit ihren möglichen Effekten die Inkarnation der Teufelei schlechthin! Er kann wohl alles Mögliche an Schäden im System anrichten, von zerstörten Zeigern bis zu Systemabstürzen, die erst Stunden später auftreten. Gegen ihn gibt es nur eine Waffe: Die „`T`"–Option des PEARL–Compilers. Damit wird der unerlaubte Zugriff durch überschrittene Indexgrenzen in jedem Fall verhindert. Hat der Bold aber bereits zugeschlagen, sollten Sie von Ihren Files retten, was zu retten ist, und die „RESET"–Taste aktivieren.

Der Parameter–Teufel

Der PEARL90-Compiler prüft lückenlos, ob Prozeduren mit den richtigen Parametern aufgerufen werden. Bei ihm bekannten Prozeduren aus dem eigenen Modul ist dieser Test lückenlos. Bei externen Prozeduren kann er sich nur auf die Korrektheit der ihm mitgeteilten Spezifikation verlassen. Stimmt die innere Struktur der zur Laufzeit aufgerufenen Prozedur mit der Spezifikation nicht überein, so sind katastrophale Folgen wie beim Index-Error möglich. Genau wie beim Indexerror-Teufel so hilft auch hier die „`T`“–Option wirkungsvoll. Es wird auch nur sehr wenig Prozessorzeit für den Parameter-check verbraucht. Das Programm sollte aber nach der Fehlermeldung **keinesfalls** fortgesetzt werden, da eine Notreparatur wie beim Indexfehler nicht möglich ist!

Der Doppellade–Teufel

Seine Spezialität besteht darin, uns zu einem unüberlegten `LOAD`–Befehl zu verführen, obwohl eine gleichnamige Task noch in der Verwaltung existiert. Mit jeder Aktivierung der Task erwischen wir dann immer nur die erste (meist alte) und wundern uns, daß Programmänderungen ohne Wirkung geblieben sind. Wir sollten darum stets sicher gehen, daß Doppelladen nicht auftreten kann.

2.4 Interpretation von Fehlermeldungen

2.4.1 Der Error-Dämon

Der Betriebssystemkern von **RTOS–UH** bekommt bei seiner Arbeit Hilfe durch die im Einführungsteil vorgestellten „Dämonen“. Das sind im Prinzip normale Nutzerprozesse (Tasks), die quasi eigenverantwortlich arbeitende gute Geister des Systemes sind. Dämonen finden wir bei **RTOS–UH** auch im Ein-/Ausgabesystem und als Netzwerkdämonen. Hier interessiert uns zunächst nur der Error-Dämon mit dem Tasknamen `#ERRDM`. Er hat die höchste Priorität aller Tasks im System und ist dazu gedacht, über eventuelle Irregularitäten bei der Arbeit des Systemkernes zu berichten. Sein zweiter – hier nicht interessierender – Aufgabenbereich ist das Starten des sogenannten „primären Shellprozesses“ beim Anschlag der `Ctrl A` Taste.

Für jeden Nutzerarbeitsplatz gibt es ein sogenanntes User-Environment. Es enthält einen Puffer für Fehlermeldungen, der ringförmig verwaltet wird. Seine Größe ist implementierungsabhängig, in der Regel ist Platz für 6 bis 12 Meldungen. Kommen neue Fehlermeldungen an, bevor der Error-Dämon durch Textausgabe der alten wieder Platz im Puffer schaffen konnte, so gehen die neuen Meldungen verloren. Aus diesem Grund sollte man sich nicht an vorbeihuschenden Fehlerinformationen erfreuen, sondern daran denken, daß vielleicht wichtigere Nachrichten als die momentan visualisierten verloren gehen können. `#ERRDM` liegt in der Priorität höher als der Bedieninterpreter, bei manchen Terminals muß man bei vorbeirasenden Meldungen ziemlich hartnäckig auf die `Ctrl A` Taste hacken, um wieder in das System zu kommen.

Jede Meldung über `#ERRDM` beginnt mit dem typischen „>>“ am linken Rand der Terminalzeile. Diesem String folgt typischerweise der Name der verursachenden Task, eine Adresse oder ein weiterer Name, sowie die eigentliche Botschaft. Je nach Schwere des Fehlers wird die Task suspendiert oder kann weiterlaufen.

Die Fehlermeldung wird in den Error-Datenkanal desjenigen Nutzers geschrieben, der dafür verantwortlich gemacht wird. Verantwortung übernimmt ein Nutzer durch Absetzen eines Kommandos oder durch die Aktivierung einer fehlerhaften Task. Ist die Verantwortlichkeit nicht feststellbar, z. B. bei Interruptprozessen, so wird der Konsolennutzer angesprochen.

Der Errordatenkanal, in den der Dämon schreibt, kann mit einem besonderen Kommando (`PER` = Permanent Error Redirection) permanent umdefiniert werden. Dies ist sinnvoll, wenn man die Fehlermeldungen unbedienter Systeme in einem File sammeln möchte.

2.4.2 Beispiele für Fehlermeldungen

Wir studieren hier exemplarisch einige Meldungen:

```
>> ABCD: 00008022 wrong op-code trap
```

Die Task `ABCD` ist auf einen unbekannten Befehl gelaufen, der Programmzähler steht jetzt auf `008022`. Der falsche Befehl (Speicherfehler?, Feldüberschreitung ohne Tester?) muß also kurz vor dieser Adresse zu finden sein. Wenn die Task `ABCD` eine PEARL–Task ist und mit der `/*+M */`–Option übersetzt wurde, wird mit `/L`*xxxxx* die Nummer der letzten registrierend überlaufenen Programmquellzeile ausgegeben. Diese kann man sich auch mit dem `DL`–Kommando (Display Line) ausgeben lassen.

```
>> COPY/26: (terminate)
```

Dies ist keine Fehlermeldung. Die Ausgabezeile dient als Hinweis, daß der Sohnprozeß `COPY/26` jetzt seine Arbeit beendet hat.

```
>> --??--: (terminate)
```

Auch hier ist kein Fehler aufgetreten, sondern ein Sohnprozeß hat sich terminiert und wollte sich verabschieden. Da der Error-Dämon `#ERRDM` aber gerade mit anderen Dingen beschäftigt war, konnte er die „Fertig–Meldung“ erst ausgeben, nachdem der Sohnprozeß schon aus dem Speicher verschwunden war. Ein Name war zu diesem späteren Zeitpunkt nicht mehr zu ermitteln. Die Meldung tritt nur noch bei alten Shellmodulen auf, die nicht in der Lage sind, eine eigene Ausnahmebehandlung auszuführen.

```
>> **V200:0008FFA2 wrong address
```

Die Sterne bedeuten, daß sich keine Task zuordnen läßt, weil der Fehler in einem Interruptprozeß aufgetreten ist. Dabei bedeutet `V200`, daß der Interruptprozeß über den Exceptionvektor `$200` angeschlossen ist. Fast immer bedeutet eine solche Meldung eine bedrohliche Entwicklung. Tatsächlich konnte der Kollaps des Systemes nur durch den integrierten Interruptrückfallmechanismus von **RTOS–UH** verhindert werden. Anders als in vielen anderen Systemen hängt der Interruptprozeß trotz der gravierenden Fehlfunktion nicht, sondern wurde geordnet abgebrochen.

```
>> MASTER:MURKS not suspended (continue)
```

Die Task `MASTER` hat eine `CONTINUE`–Anweisung für die Task `MURKS` ausgeführt. Das Betriebssystem stellt aber fest, daß `MURKS` gar nicht suspendiert ist und die Anweisung somit ohne Wirkung bleiben muß. Keine schlimme Sache, irgendein kleinerer Denkfehler des Programmierers beim Tasking.

```
>> x:INTRPT overflow (activate)
```

Die Task `x` sollte vom Systemkern durch einen Interrupt (Zeittakt oder externes Signal) aktiviert werden. Die Task `x` kann aber aus irgendwelchen Gründen dem Aktivierungstakt nicht folgen. Vielleicht ist sie zu langsam und wird darum nicht rechtzeitig fertig oder sie wurde suspendiert. Diese Prüfeigenschaft des Systemkernes ist ein wichtiges Element der Echtzeitqualitäten des Systemes.

2.4.3 Der Exception-Handler

Neuere Software in der **RTOS–UH**-Welt stößt bei Irregularitäten, für die sie selbst verantwortlich ist, nicht mehr den Error-Dämonen an. Stattdessen wird bei solcher Software dem Systemkern die Adresse einer eigenen Fehlerbehandlungsroutine mitgeteilt. Diese Fehlerbehandlungsroutine wird in der Fachsprache auch „Exception Handler“ genannt. Alle Shellprozesse benutzen diesen Weg. Damit wird ein korrektes Weiterleben des Prozesses selbst nach fatalen Situationen möglich. Exekutiert man etwa von der Shell aus einen `DM`-Befehl mit Zugriff auf einen nicht vorhandenen Speicher, dann bringt der `BUS-ERROR` die Shell nicht zum Halt. (In alten Versionen des Betriebssystemes wurde die Shell wie eine normale Nutzertask durch solche Fehler blockiert und mußte mit Break wiederbelebt werden).

Meldungen der Exception-Handler werden wie die des Error-Dämonen interpretiert, denn es werden in beiden Fällen die gleichen Texterzeugungsroutinen benutzt. Die Meldungsausgabe erfolgt nun von der fehlerverursachenden Task mit deren eigener Priorität.

Da die Shellprozesse die Fehlerausgabe selbst erledigen, kann man deren Meldungen mit dem Befehl `ER` (Error Redirect) auch temporär umlenken, ohne daß der Error-Dämon von dieser Umlenkung betroffen ist.

2.5 Das Pathlist-Konzept von RTOS–UH/PEARL

Wie bereits im Schnellkurs 2 erläutert, gibt es in der **RTOS–UH**–Welt zwei Typen von Datenstationen: Typ I, der dermaßen schnell arbeiten kann, daß keine Wartephasen für den Prozessor anfallen (Digitale E/A, A/D– und D/A–Wandler etc.) und Typ II, der beim Datentransport den Prozessor derart wenig belastet, daß es sinnvoll ist, die kostbare Prozessorleistung für andere Aufgaben verfügbar zu machen. Der Typ I wird nicht vom Betriebssystem unterstützt, da seine Ansprache durch Nutzertask–eigenen direkten Maschinencode erfolgt. Dieser Typ ist daher auch nicht über Bedienbefehle ansprechbar, allenfalls in bestimmten Fällen durch seine Hardware–Adresse mit dem „`SM`“– bzw. „`DM`“–Befehl.

Uns interessiert hier nur der Typ II, der in **RTOS–UH** mit den auf Seite 46 schon erwähnten „Dämonen“ realisiert ist. In diesem Fall handelt es sich um die Dämonen des Ein-/ Ausgabesystemes, die „I/O-Dämonen“. Sie sind wie alle Dämonen normale Tasks, auch der Begriff „Warteschlangenbetreuungstask“ beschreibt sie gut. Jeder I/O-Dämon kümmert sich um eine ihm zugeordnete Warteschlange, in der seine Aufträge stehen. Wenn ein Nutzerprozeß z. B. auf dem Drucker etwas ausgeben will, so reiht er in Wirklichkeit nur einen Ausgabeauftrag in die Warteschlange des Druckerdämonen ein. Diese Warteschlange ist, wie im Schnellkurs beschrieben, nach Prioritäten geordnet. Die moderneren I/O-Dämonen laufen mit variabler, selbstanpassender Priorität. Damit tragen sie der aktuellen Auftragslage optimal Rechnung, behindern aber Nutzerprozesse, die wichtiger als ihre Auftraggeber sind, praktisch nicht. Auch sonst erinnern sie an Chamäleons: Sie verwenden die Nutzeridentifikation desjenigen Prozesses, für den sie gerade arbeiten. (Wes Brot ich eß, des Lied ich sing . . .).

Wenn verschiedene Geräte eine gemeinsame, nur einmal vorhandene Komponente teilen müssen (z. B. ein Controller für mehrere Floppy–Laufwerke), so gibt es hierfür nur einen gemeinsamen Dämon und einegemeinsame Warteschlange, was gewisse Einschränkungen bringen kann: Eine physikalisch echt zeitlich parallele Arbeitsweise der beiden Geräte ist dann nicht möglich.

Für die Ansprache der Datenstationen über die sogenannte „Pathlist“ spielen diese internen Details allerdings keine Rolle. Die Adressierung jeder über Bedienbefehle oder PEARL–Programme erreichbaren Datei geschieht auch im **RTOS–UH** so wie es heute allgemein üblich ist, nämlich durch einen hierarchisch aufgebauten Pfad, eben jene bereits erwähnte „Pathlist“:

```
/Gerätebez/subdir/subdir/.../subdir/filename
/Netzrechner/Gerätebez/subdir/.../subdir/filename
```

Durch das Zeichen „/“ wird die Pfadliste auf der allerobersten Ebene, der Root– (Wurzel–) Ebene begonnen. Anschließend muß entweder über einen im System definierten Gerätebezeichner oder explizit numerisch eine Warteschlange (Warteschlangennummer= LDN) samt zugehöriger Untergliederungsnummer (Laufwerk= DRIVE) folgen.

Zwischen Klein– oder Großschreibung wird beim Gerätebezeichner im Gegensatz zum Rest der Pathlist nicht unterschieden. Eine numerische Angabe der beiden Parameter LDN und DRIVE wird durch einen Sondergerätebezeichner „LD/*integer.integer/*“ ermöglicht:

/LD/7.6/ steht für LDN=7, DRIVE=6.

Der endende Slash dieses Konstruktes darf niemals fehlen, auch nicht, wenn die Pathlist hinter der DRIVE–Zahl ohnehin beendet ist. Wenn für DRIVE keine Zahl angegeben wird, so substituiert das System den Wert 0. So ist z. B. /LD/3/ identisch zu /LD/3.0/.

! → Die maximale Länge der Pfadliste ist begrenzt, allerdings ist diese Zahl keine interne RTOS-Konstante. Sie hängt ausschließlich von Ihrer Implementierung ab. Bei den älteren Systemen durften maximal 24 Zeichen benutzt werden – wobei der Gerätebezeichner nicht mitzählt. Dieser Wert ist für die sehr kleinen Mikrokontrollersysteme immer noch eine gute Empfehlung, weil dadurch an vielen Stellen Speicherplatz gespart wird. VME-Systeme und die Apple-Systeme arbeiten heute meist mit einer Maximallänge von 64 Zeichen.

! → Auch der Querverkehr zu irgendwelchen anderen Rechnern im Netz (RTOS–PDV–Bus–Netz, Ethernetkopplung, RTOS–Profi–Bus–Netz etc.) zweigt von der Root–Ebene ab. Der Bezeichner fixiert in diesem Fall sowohl das angewählte Netz als auch die untergeordnet selektierte Rechnerstation. In dem angewählten Rechner befindet man sich dann erneut auf dessen Root–Ebene und steigt von dort weiter über einen dort gültigen Gerätebezeichner ab.

/F0	Eigene Floppy, Laufwerk 0
/A1	Konsolterminal
/ST3	PDV–Netz, Rechner No. 3
/LD/0,2/	Numerisch: LDN=0, DRIVE=2 (Konsole)

Wenn die Pathlist hinter dem Gerätebezeichner endet, so wird sie vom System gedanklich um „/--“ verlängert, wobei dieser „Defaultfile“ mit Namen „--“ aber nicht explizit (z. B. /F0/-)

angegeben werden kann. Der Defaultfile hat für manche Programme eine sinnvolle Bedeutung, z. B. bei `COPY`, wo er die automatische Substitution eines Teiles der Partnerpathlist bewirkt.

Außer bei den nicht weiter untergliederungsfähigen seriellen (z. B. `/A1`, `/B2` etc.) oder parallelen (z. B. `/PP`) Rechnerschnittstellen ist hinter dem Gerätebezeichner, angehängt durch „`/`“, ein weiterer Bezeichner nötig. (Es sei denn, daß man seine ganze Floppy über `/F0` als einen einzigen File „`--`“ benutzt . . .). Dieser Fortsetzungsbezeichner kann folgendes bedeuten:

1. Ein Unterinhaltsverzeichnis (Subdirectory).
2. Der Defaultfile in einem Unterinhaltsverzeichnis. (Dann endet hier die Pathlist hinter dem „`/`“).
3. Name einer Datei (Dann endet hier die Pathlist).
4. Ein Gerätebezeichner (erzeugt `LDN` und `DRIVE`) passend für den Rechner, der durch den vorhergehenden Bezeichner über das Netz angewählt wurde. Hier ist jedoch aus Sicherheitsgründen keine numerische Angabe möglich, sondern nur ein im Zielrechner freigegebener Bezeichner.

Beispiele zu 1 bis 4:

1. `/F0/SYSTEM/...` `SYSTEM` ist ein Subdirectory
2. `/F0/SYSTEM/` Defaultfile Names „`--`“ in `/F0/SYSTEM`
3. `/F0/Daten` File `Daten` auf Floppy Nr. 0
4. `/ST4/A1` Konsole des Rechners Nr. 4 im Netz
   `/ST6/ED/...` ED–File des Rechners Nr. 6 im Netz

Beispiele: Kopieren mit vollständigen längeren Pathlists:

```
copy /st4/ed/mueller/prog.p > /st3/pp
type /ed/maier/test
cp /ed/prog>/F0/save/ (Geht nach /F0/save/prog!)
(cp ist die Kurzform von COPY)
```

Wie zu erkennen ist, darf der Punkt als Namensbestandteil verwendet werden. Dies ist hauptsächlich gedacht, um beim Hantieren mit Disketten anderer Softwareanbieter deren „Extensions“ verwenden zu können. Bezeichner dürfen Ziffern enthalten.

! → Wegen der Kompatibilität zu älteren **RTOS**–**UH**–Versionen, bei denen Punkt und Doppelpunkt eine spezielle Bedeutung haben

— nämlich Trennung des Gerätebezeichnerstrings vom Rest der Pathlist — muß vor Mißdeutungen gewarnt werden, z. B. wird

```
ed.xy = ed:xy = /ed.xy = /ed:xy = /ed/xy
```

vom System als gleichwertig akzeptiert. Es sollte nur die letzte Form Verwendung finden, da die ältere Syntax sehr leicht zu Fehlern bei der Adressierung im Rechnernetz führen kann.

! → Ebenfalls aus Kompatibilitätsgründen wird als Gerätebezeichner auch noch der String „L*x*“ angenommen, wobei „*x*“ für eine Zahl oder einen Buchstaben steht. Auch hier wird empfohlen, diese Syntax fortan nicht mehr zu benutzen. (Die Zuordnung war dabei wie folgt: `L4` = `/LD/4,0/`, `LA` = `/LD/10,0/`, `LG` = `/LD/16,0/` etc.). Logischerweise sind daher `LO` und auch `LD` Gerätebezeichner, wenn man sie durch ein Versehen im falschen Kontext benutzt!

! → Mit dem Bedienbefehl „`CD`“ verändert man ein sogenanntes „Working–Directory“. Dies ist ein vorderer Teil der Pathlist, der immer dann automatisch vom System vorangestellt wird, wenn eine Pathlist nicht mit dem Wurzelsymbol „`/`“ startet.

Beispiel: `CD /ed/mueller`

`type mist` == `type /ed/mueller/mist`

! → Es ist immer irgendein Working–Directory definiert. Beim Start des Systemes ist es „`/`“, so daß man in diesem Fall direkt mit dem Gerätesymbol starten kann, d. h. „`load b2`“ wird dann wie „`load /b2`“ behandelt. Es ist aber absolut nicht zu empfehlen, sich diese nur um ein Zeichen kürzere Adressierung anzugewöhnen — es sei denn, daß man grundsätzlich nie mit einem Working–Directory arbeitet ... (wehe aber, man hilft dann einem anderen Nutzer, der mit Working–Directory arbeitet, bei der Abfassung eines Bedienbefehles!)

2.6 Einige technische Daten

Maximale Anzahl gleichz. aktiver Tasks:	Nur durch Speicher begrenzt.
Maximale Anzahl von Semaphorvariablen:	Nur durch Speicher begrenzt.
Maximale Anzahl von Prozeßinterrupts:	32
Reaktionszeit, Prozeßinterr., PEARL:	$3 \ldots 200\ \mu s$
	je nach Prozessor

Prozeßumschalter („Dispatcher“): Rein ereignisgesteuert, kann auch Systemaufrufe niedriger priorisierter Tasks – das sind Supervisormodesequenzen – abbrechen („echte“ Preemption).

Task–Wechselzyklus A–B–A: etwa $7 \ldots 400\ \mu s$

I/O–System: Prioritätsgeordnete Warteschlangen mit eigenen Betreuungsprozessen (I/O–Dämonen, I/O–Tasks). Die Ein– und Ausgabe arbeitet asynchron und ist ohne Wartephasen des Auftraggebers möglich, wobei die Information nicht umkopiert werden muß. Durch einen Trap (`XIO`) wird ein unabhängiger I/O–Dämon gestartet. Damit werden wesentliche Nachteile der sonst üblichen unterprogrammgesteuerten Ein– und Ausgabe vom Prinzip her bereits vermieden.

Nutzerdatenstationen können permanent oder temporär hinzugefügt werden.

Bedieninterpreter: In Universal–Maschinensprache codiert. Hierarchisches, erweiterbares Shellkonzept mit einer leistungsfähigen Shellsprache.

PEARL90-Compiler: In virtuellem (VCP–) Code geschrieben, damit extrem kompakt. Eine Version in native Code ist ebenfalls verfügbar. Voll kompatible Crossversionen für MS-DOS u. a. Systeme.

Portabilität: Durch Austausch der Implementierungsscheibe an alle Rechner anpaßbar, für die ein Transferassembler existiert. (zur Zeit 680xx und PowerPC 603/4)

(Leere Seite vor neuem Kapitel)

Kapitel 3: Bedienung des Systems

3.1 Struktur der RTOS-Shell

3.1.1 Die 8 Ebenen der Shell

Dieser Abschnitt ist nicht für den eiligen Leser gedacht, sondern für diejenigen, die gerne detaillierte Kenntnisse über das System erwerben möchten. Beim ersten Lesen dieses Handbuches kann man evtl. gleich zu 3.2 auf Seite 61 weiterblättern.

Der Kern des Betriebssystemes kennt keine Shell. Viele eingebettete Systeme benötigen auch keine. Shell-lose Systeme entstehen, indem man bei der Komposition des Systemes die entsprechenden Scheiben fortläßt. Jede menschliche Kommunikation erfolgt mit Hilfe einer aus Systemsicht völlig gewöhnlichen Task, der Shelltask, auch „Shellprozeß“ genannt. **RTOS–UH** ist kein PC-Betriebssystem, seine Shellphilosophie ähnelt eher den „...ix“-Systemen aus der Multiuser-Workstationwelt – jedoch ohne deren echtzeit-riskanten Strategien und unter Einsparung der Schutzmechanismen gegen „böswillige“ Mitbenutzer. Es wurde optimiert auf effiziente und sichere Echtzeiteigenschaften. Die Forderung, quasi jederzeit Einblick und Eingriff nehmen zu können – und dies ohne die laufenden Echtzeitprozesse zu stören! –, stand bei der Entwicklung im Vordergrund.

Es gibt verschiedene Arten von Shellprozessen. Gewissermaßen die „Väter“ jeder Bedienkommunikation mit dem System sind die „primären Shellprozesse“. Primäre Shellprozesse können „permanent“ oder „temporär“ sein.

Die Zahl **permanenter primärer Shellprozesse** wird bei der Komposition des Systemes festgelegt. Sie ändert sich nach dem Einschalten nicht mehr. Permanente primäre Shellprozesse laufen typischerweise durch den Anschlag von `Ctrl A` auf einer ihnen fest (!) zugeordneten Eingabetastatur an und schreiben einen Eingabeaufforderungsprompt. Sie repräsentieren gewissermaßen fest installierte Nutzerarbeitsplätze des Systemes.

Die **temporären primären Shellprozesse** benötigt man z. B. für das Fern-Einloggen über ein Netz. Es entstehen damit vorübergehend quasi weitere Nutzerarbeitsplätze. In aller Regel wird aus Sicherheitsgründen den temporären primären Shellprozessen eine deutlich niedrigere Priorität als den permanenten zugewiesen.

Alle primären Shellprozesse besitzen ein „User-Environment“ und sind der Ort, an dem die „User-ID“ entsteht. Die sogenannten „sekundären“ Shellprozesse sind Handlanger oder Abkömmlinge primärer Shellprozesse. Sie erben deren User-ID und erhalten bestimmte Daten aus deren User-Environment als Kopie. Die primären Shellprozesse sind unsere eiserne Zugriffsreserve auf das System. Sie kennen meistens eine Art „Notruf“ über die `BREAK`-Funktion der Tastatur.

In Wirklichkeit ist jeder „Shellprozeß“ eine vom Code her geradezu winzige Task, nur ca. 200 Bytes groß. Die komfortable „RTOS–Shell“ entsteht erst durch das „Ebenenmodell“. Die Elemente der einzelnen Ebenen sind wiedereintrittsfeste Unterprogramme. Dazu gehört das „Shell Subroutine Package“, abgekürzt „`SSRP`“. Auch die Dekodierung der Befehle der Grundshell „`SHL`“ ist in entsprechenden Unterprogrammen realisiert. Gleiches gilt für alle weiteren Zusatzshellmodule. Weil der Code des „`SSRP`“, des „`SHL`“ und aller korrekt kodierten Zusatzshells wiedereintrittsfest ist, können ihn verschiedene Shellprozesse gleichzeitig benutzen, ohne sich gegenseitig zu beeinflussen. Das Ebenenmodell kann man sich in etwa wie folgt vorstellen:

E	Ebene	Funktion	Identifikation
1	Shell–Prozeß	Call E2	`#USERxy` ,`#XCMMD`
2	Shell–Subroutine–package	UP-Sammlung	`SSRPxy`-slice
3	Die Grund–Shell	Standardbefehle	`SHLxy`-slice
4	Extra–Shell	Optionalbefehle	??
5	Nutzer–Shellbefehle im ROM	nach Wunsch	ROM–slices
6	Nutzer–Shellbefehle im RAM	nach Wunsch	Speichersektion
7	Transiente Befehle	nach Wunsch	z.B. `/F0/ABCD`
8	Skript in Shellsprache	nach Wunsch	z.B. `/H0/skript1`

Dieses 8–Ebenen–Schema beschreibt recht genau, wie die Verarbeitung eines jeden eingegebenen Bedienbefehles funktioniert. Wir unternehmen eine Reise durch die Ebenen und beginnen mit dem Anschlag von `Ctrl A` auf unserem Terminal (bzw. auf der Tastatur, während der Eingabefocus im Kommandofenster unseres Schirmes ist).

Ebene 1: Wenn wir nach Anschlag von `Ctrl A` den Eingabeprompt (z. B. das Zeichen „*“) erhalten haben, so hat die Ebene 1, in diesem Fall eine „primäre permanente Shelltask“ (in der sich das „User–Environment“ befindet) diesen Prompt geschrieben und anschließend die Eingabe angestoßen, auf die sie nun wartet. Sekundäre Shellprozesse, so auch die spezielle Shell-Task mit Namen `#XCMMD`, (die gleichzeitig auch eine I/O-Task ist) schreiben keine Eingabeaufforderung, weil sie den Bedien-Befehlstext auf irgend eine Weise bereits erhalten haben.

Wir lassen den Shellprozeß nicht lange warten, sondern geben ihm nun den Text „`EinTest`“ als Befehl ein, Abschluß mit CR.

Ebene 2: Der Shell–Prozeß ruft mit diesem String das `SSRP` auf (es muß also vorhanden sein). Mit Hilfe des `SSRP`-Unterprogrammes sucht der Shell–Prozeß nun zunächst in der Tabelle der Grundshell
Ebene 3: nach einem Befehl dieses Namens. Dabei wird zwischen Groß- und Kleinschreibung nicht unterschieden. In unserem Fall wird der Befehl „`EinTest`“ dort natürlich nicht gefunden und die Suche geht automatisch in den im ROM (bzw. gebooteten RAM,
Ebene 4: in dem das System sitzt) vorhandenen Extra–Shell–slices weiter. Auch hier wird nicht zwischen Groß- und Kleinschreibung unterschieden, wäre ein Befehl „`EINTEST`“ dort definiert, so würde er
Ebene 5: unser Kommando übernehmen. Auch die Befehle, die der Nutzer selbst zum System hinzugefügt hat, werden am Ende dieser Suche erfaßt.

Nachdem auch diese Suche fehlgeschlagen ist, untersucht der Shell–Prozeß (der immer noch im `SSRP`-Unterprogramm steckt) nun, ob es eine Task mit dem Namen „`EinTest`“ in der Systemverwaltung gibt, die dann behandelt würde, als ob man den Befehl „`activate EinTest ...`“ eingegeben hätte. Hierbei wird dann wie beim `ACTIVATE`-Befehl zwischen Groß- und Kleinschreibung unterschieden.

Ebene 6: Die Suche geht weiter im von **RTOS–UH** verwalteten RAM. Alle `SMDL`-Sektionen und auch alle `PMDL`-Sektionen werden untersucht, ob es darunter einen Befehl „`EinTest`“ gibt – ohne Unterscheidung von Groß- und Kleinschreibung. (`SMDL`, `PMDL` siehe Seite 201)

Ebene 7: Nun wird der Befehl „`EinTest`“ versuchsweise als Bezeichner eines Files interpretiert. Weil er nicht mit dem Zeichen „`/`“ beginnt, werden ihm nacheinander die vorhandenen Execution-Directories vorangestellt. Vorher werden alle Kleinbuchstaben endgültig in Großbuchstaben verwandelt. Wird kein passender File mit Namen „`EINTEST`“ gefunden, so bricht der immer noch im `SSRP` laufende Shell–Prozeß mit einer Fehlerantwort nach `Stderr` die weitere Befehlsdekodierung ab. In der Fehlermeldung erscheint unser Eingabebefehl jetzt als `EINTEST`, weil die Umwandlung nicht rückgängig gemacht wird.

Wir nehmen an, ein File wurde gefunden. Er wird jetzt vom Shell–Prozeß mit Hilfe entsprechender Code-Sequenzen des `SSRP` gelesen, und der Anfangstext wird analysiert. Wenn der Inhalt mit `S0...` beginnt und wenn im System der RTOS-Lader vorhanden ist (er ist eine eigene Scheibe und mit dem integrierten `LOAD`-Befehl der Ebene 4 zuzuordnen), so wird der File `/.../.../EINTEST` geladen. Enthält er einen passenden Shell-Befehl, so wird dieser als „transientes Kommando“ ausgeführt, anderenfalls erfolgt ein Fehlerabbruch („`....cannot execute`“ nach `Stderr`). Transiente Kommandos erzeugen einen der Shell unterlagerten eigenen Prozeß, im **RTOS–UH** auch Sohnprozeß genannt, der das Laden übernimmt und danach wieder verschwindet. Der Shell–Prozeß wartet während des Ladens auf diesen Sohn. Das in den Speicher geladene Modul verschwindet nach Abarbeitung des Befehles automatisch.

Ebene 8: Wenn der Inhalt des gefundenen Files nun nicht mit `S0...` anfing und im System der optionale Shellspracheninterpreter vorhanden ist, so wird ein Sohnprozeß eingerichtet, der die interpretative Abarbeitung des Shellsprachprogrammes übernimmt – ein sekundärer Shellprozeß entsteht. Die primäre Shell–Task übergibt diesem sekundären Shellprozeß bestimmte Daten aus ihrem „User-Environment“ und startet diesen. Die RTOS-Shellsprache ist in einem eigenen Abschnitt beschrieben, der auf Seite 74 dieses Handbuches beginnt.

Rekursion: Man beachte, daß der Shellspracheninterpreterprozeß der Ebene 8 nun selbst wieder das `SSRP` aufrufen kann und damit erneut alle Folgeebenen durchlaufen werden können. Wegen der Rekursivität und Wiedereintrittsfestigkeit der **RTOS–UH**–Software gibt es keine Probleme.

3.1.2 Prozeßphilosophie der RTOS–UH–Shell

Die „Shell“ als Partner unserer Kommunikation ist aus Sicht des Systemes eine ganz gewöhnliche Task, ein „Prozeß“ im Sinne der Informatik. Allerdings ist dies meist ein Prozeß mit sehr hoher Priorität. Da Prozesse mit hoher Priorität generell hinsichtlich der Echtzeitreaktivität bedenklich sind, ist der Shellprozeß so strukturiert, daß er nur solche Aktionen selber ausführt, die nur sehr wenig Prozessorleistung erfordern. Es wäre keine gute Lösung, nun etwa die Priorität der „primären Shellprozesse“ abzusenken – sonst kann man nämlich sehr leicht durch länger laufende Nutzerprozesse aus dem Systemzugriff herausgedrückt werden. Die Priorität der permanenten primären Shellprozesse kann dennoch bei der Zusammenfügung des Systemes über die User–Environment–Scheibe frei festgelegt werden – für Einsatzfälle, bei denen jede Bedienung immer den sonstigen Aufgaben untergeordnet sein muß.

Das Problem dieser widersprüchlichen Anforderungen – Hohe Echtzeitgüte und gleichzeitig einen immer schnell möglichen Shellzugriff – wird beim **RTOS–UH**-System durch Delegation von Aufgaben an niedriger priorisierte „Sohnprozesse“ gelöst. Typische Sohnprozesse sind etwa die Compiler und Assembler. Auch die vom Nutzer in PEARL kodierten Shellfunktionen laufen grundsätzlich als Sohnprozesse ab – aus Sicherheitsgründen, schließlich könnte der Nutzer aus Versehen eine Unendlichschleife programmiert haben. Der Interpreter für die Shellsprache läuft aus dem gleichen Grund ebenfalls als eigener Prozeß ab, dessen Prioriät im Nutzer-Bereich liegt und damit deutlich niedriger als die der primären Shellprozesse ist.

Bei den Shellbefehlen, die Sohnprozesse generieren, kann man den Namen des Prozesses selbst bestimmen und den Prozeß damit weiteren Manipulationen zugänglich machen. Außerdem kann mit Hilfe des „`WAIT`“-Befehles auf das reguläre oder irreguläre Ende eines Sohnprozesses gewartet werden. Mit Hilfe der Namenswahl könnten z. B. durch zyklische Einplanung einer `COPY`–Subtask beliebige Befehle automatisch wiederholt werden:

```
COPY.X /ED/X1>/XC/;T X;ALL 10 SEC X
```

Der `COPY`-Prozeß schickt nun alle 10 Sekunden den Inhalt des Files `/ED/X1` an den Shellprozeß `#XCMMD`, der die Befehle wie oben beschrieben auswertet (Näheres in 3.2.4 auf Seite 64).

No.	Prozeß	Priorität	Bemerkung
1	Primärer Shellprozeß Netz-Shellprozeß	sehr hoch hoch	Wird durch Ctrl A aktiviert. Remote Login über Netz
2	Sohnprozeß von 1,2,3 und/oder Sekundärer Shellprozeß —— “ ——	mittel	Compiler, Assembler etc. PEARL-kod. Shell-Befehl Shell-Folgebefehle des Sohnes oder Shellsprachinterpreter
3	`XCMMD`-Prozeß – “ –	variabel	gleichzeitig I/O-Prozeß und sekundärer Shellprozeß
4	Lader für trans. Befehl	hoch	Flüchtiger Sohn von 1... 4

Tabelle 3.1: Übersicht über mögliche Shellprozesse

3.1.3 Das User-Environment

Bei der Montage des Systemes wurden sogenannte „User-Environment“–Scheiben eingebunden, in denen u. a. die Zugriffspfade für „`Stderr`“, „`Stdin`“ und „`Stdout`“ vordefiniert wurden. Das User-Environment liegt im Verwaltungskopf jedes primären Shellprozesses, der in der Tabelle im Abschnitt 3.1.2 mit der lfd. Nummer 1 als „Vater aller Shell-Aktivitäten“ zu finden ist. Bei temporären primären Shellprozessen wird das User-Environment durch den Log-In des Netzwerkhandlers eingerichtet. Im User-Environment ist auch der Platz für das Working-Directory und mehrere (bei der Systemmontage parametrierbare Zahl) Execution-Directories . Sie bestimmen die Pfadlisten, mit deren Hilfe die Shell in den Ebenen 7 und 8 nach Kommandos sucht.

Die im User-Environment abgelegten Daten sind folgende:

Systemname	Bedeutung	beeinflußbar?
Error-Buffers	Meldungsspeicher für Fehler	nur ungewollt
`NXD`	No. of Execution-Directories	bei Systemgenerierung
`STDE...`	Stderr-path	über `PER`-Befehl
`STDI...`	Stdin-path	über `PI`-Befehl
`STDO...`	Stdout-path	über `PO`-Befehl
`WXDIR`	W/X-Directory	`CD`- und `CXD`-Befehle
`ENVADR`	Pointer to Extra-Environment	mit `ENVSET`-Befehl

Die Daten des User-Environmentes werden vom primären Shellprozeß für seine eigenen Aufgaben und die seiner Abkömmlinge (Sohnprozesse, sekundäre Shellprozesse) unterschiedlich ausgewertet und weitergegeben.

- `NXD` wird stets unverändert weitergegeben.
- Von `STDE...` ,`STDI...` und `STDO...` wird eine lokale Kopie angefertigt, auf die die Befehle „`I`“, „`O`“ und „`ER`“ für die Dauer einer kompletten Befehlszeile einwirken. Mit jedem Aufwecken durch `Ctrl A` fertigt sich der primäre Shellprozeß die Kopie erneut an. Alle Sohnprozesse und sekundären Shellprozesse erhalten im Moment ihrer Entstehung ihre eigene Kopie dieser E/A-Pfadbeschreibungen.
- Beim `WXDIR` arbeitet ein primärer Shellprozeß stets mit seinem Original. Die Befehle „`CD`“ und „`CXD`“ überdauern darum mit ihrer Wirkung Ende und Neuaktivierung eines permanenten primären Shellprozesses. Sohnprozesse und sekundäre Shellprozesse erhalten dagegen eine Kopie und arbeiten damit. Wenn solche Prozesse Working-/Execution-Directories im User-Environment verändern sollen, so sind die Befehle „`CUD`“ und „`CUXD`“ zu verwenden. Bei primären Shellprozessen ist die Wirkung von „`CD`“ und „`CUD`“ sowie „`CXD`“ und „`CUXD`“ logischerweise jeweils völlig identisch.

3.2 Umgang mit der Shell

3.2.1 Aufbau der Anweisungszeile

In einer Anweisungszeile dürfen keine, eine oder mehrere Anweisungen stehen. Mehrere Anweisungen in einer Zeile werden durch ein Semikolon oder das Doppelminuszeichen „`--`“ voneinander getrennt. Durch Semikolon getrennte Anweisungen werden soweit möglich parallel abgearbeitet. Wenn die Anweisungen zeitlich nacheinander abgearbeitet werden sollen, sind sie durch „`--`“ zu trennen (siehe 3.2.5 auf Seite 65). Leeranweisungen sind auch zwischen Semikolons zulässig. Zwischenräume (Blanks) sind zwischen Schlüsselworten und Parametern erforderlich, wenn andernfalls Mißverständnisse entstehen könnten. Mehrere Parameter in einer Parameterliste können wahlweise durch Blanks oder Kommata getrennt werden. Eine Zeile wird durch das Zeichen „Carriage–Return“ beendet.

Beispiele für zulässige Eingaben:

```
DM 5000;;;                                  (mit Leeranweisungen)
SM 1000,1200;S;L;XYZ PRIO 50;TERMINATE ABCD;
T XYZ; TRACE TEST L344,23;
/HO/XD/QP SI /ED/Test LO /ED/Liste;         (trans. Befehl)
```

3.2.2 Bedienung durch den primären Shellprozeß

Die Zugriffszeit (nach Anschlag von `Ctrl A`) hängt von der dem Terminal zugeordneten primären Shelltask mit Namen `#USER`*x* und der aktuellen Systembelastung ab. Die typische Priorität permanenter primärer Shellprozesse ist allerdings die zweithöchste — nach dem `#ERROR`–Dämon — im ganzen System, so daß die Reaktion prompt erfolgen sollte.

! → Eine eventuell noch „hängende“ Eingabe (z. B. erwartet eine Task eine Eingabe) kann den Zugriff verhindern. Erscheint das Zeichen „`*`“ nicht, daher probeweise „Carriage–Return“ eingeben.

Eine Eingabezeile darf **max. 128 Zeichen** lang sein.

Der Shellprozeß kann durch `BREAK` abgebrochen werden. Es erscheint dann ein neuer Eingabeprompt. Die `BREAK`-Taste ist der letzte Rettungsanker, weil sie auch Wartezustände – z. B. mit „`WAIT`“-Befehl erzeugte – aufheben kann. Ausgabevorgänge des Shellprozesses werden sobald irgend möglich gestoppt.

Trotz der hohen Priorität des Shellprozesses werden andere laufende Aktivitäten durch ihn praktisch nicht behindert, so daß keinerlei Grund zur Zurückhaltung z. B. während einer laufenden Compilation besteht. Zeitlich aufwendige Operationen werden vom Shellprozeß ja bekanntlich durch Erzeugung von Subtasks (Sohnprozesse) erledigt und damit auf eine niedrigere Prioritätsebene verlagert. Derartige Sohnprozesse oder sekundäre Shellprozesse werden nicht durch `BREAK`, sondern mit Hilfe von `UNLOAD` abgebrochen!

3.2.3 Bedienung durch einen sekundären Shellprozeß

Sekundäre Shellprozesse entstehen durch das Anhängen von Befehlstext an Sohnprozesse der Shell mit Hilfe zweier Minuszeichen. Genaueres dazu in 3.2.5 auf Seite 65. Mit Hilfe des `DEFINE`-Befehles ist es möglich, quasi einen „leeren“ Sohnprozeß zu erzeugen, dessen einzige Aufgabe die Execution von Shellbefehlen unabhängig von unserer primären Shell ist. Auf diese Weise kann man praktisch jedes beliebige Shell-Kommando einplanbar machen.

Soll zum Beispiel ein Befehl eingeplant werden, der alle 10 Sekunden den Inhalt der Speicherzelle `$4712` anzeigt, so kann das wie folgt erreicht werden:

```
DEFINE.DM4712--DM 4712; ALL 10 SEC ACTIVATE DM4712
```

Die Namenswahl „`DM4712`“ erfolgte frei. Es ist aber natürlich sinnvoll, die Aufgaben sekundärer Shellprozesse in ihrem Namen erkennbar zu halten. Wenn man seiner Tätigkeit überdrüssig ist, kann man „`DM4712`“ selbstverständlich mit `PREVENT` ausplanen oder ihn mit `UNLOAD` völlig eliminieren.

Dem Shellprozeß (Name im Beispiel „DM4712“) wurde im Moment der Ausführung des DEFINE-Befehles Kopien von Stdin, Stdout, Stderr und WXDIR mitgegeben. Verändert man diese Daten später, so erreicht das den sekundären Shellprozeß nicht. Umgekehrt – und auch das ist sehr erwünscht – beeinflussen Umlenkungen mit „I“, „O“, „ER“, „CD“ und „CXD“ des sekundären Shellprozesses den „Vater“-Shellprozeß nicht. Ausnahmen sind hier die Umlenkungen mit „PI“, „PO“, „PER“, „CUD“ und „CUXD“, die **stets** auf den primären Shellprozeß zurückwirken – und darum sehr gefährlich sein können: Lenkt man z. B. mit PI die Eingabe auf eine Quelle um, von der keine Anweisungen kommen, so ist die entsprechende primäre Shell praktisch unbrauchbar geworden!

! → Wird ein sekundärer Shellprozeß, der ja eine normale Task ist, von einer anderen als der einrichtenden primären Shell aktiviert, so erbt er nicht die Environment-Daten dieses Shellprozesses, da seine bei der Definition gemachten Kopien nur von ihm selbst verändert werden können. Allerdings übernimmt er die User-ID-Nummer des Aktivierers. (Die jedoch bei gleichzeitiger mehrfacher Aktivierung von verschiedenen primären Shells nicht gepuffert wird!)

Eine gewisse Umlenkung der Datenausgabe sekundärer Shellprozesse ist dennoch mit Hilfe der Datenstation /TY möglich, wie an folgendem Beispiel erkennbar ist:

```
DEFINE.X--O /TY--LU
```

Es wird das Terminal (bzw. Fenster), das der primären Shell des Aktivierers zugeordnet ist, durch Umlenkung innerhalb der sekundären Shelltask (mit Namen „X“) als Ausgabe genommen. Man beachte, daß das Konzept der sekundären Shells nicht für kompliziertere Aufgaben gedacht ist; umfangreichere Aktivitäten erledigt man besser mit Hilfe von Skripten in der Shellsprache (siehe 3.5 auf Seite 74).

3.2.4 Bedienfunktionen mit Hilfe der Datenstation /XC

Jedes ASCII–Record, welches durch COPY oder Programmbefehle in die Ausgabestation /XC geschrieben wird, wird genauso behandelt, wie eine vom Nutzer über seinen Shellprozeß (Bedientask) abgesetzte Zeile. Damit ist auch das automatische und einplanbare Laden, Entladen, Compilieren etc. möglich.

Solange mit dem „O“–Kommando nicht anderes vereinbart, wird als Ausgabegerät das jeweilige Standardausgabegerät „Stdout“ aus dem User-Environment des auftraggebenden Nutzers angenommen. Bei Umschaltung durch das „O“–Kommando gilt dieses nur für den Rest der Zeile bzw. bis zum nächsten „O“–Kommando.

Beispiel 1: Beginnend um 12:00:00 Uhr soll alle 2 Sekunden der Zustand der Task TRANS aufgelistet werden. Die Zustände sollen in die Datei /ED/ZUST geschrieben werden. Um 12:02:50 soll der letzte Eintrag erfolgen:

```
COPY /A1/ > /ED/CMMD          (Erstellung der Befehlsdatei)
```

```
=O /ED/ZUST;SHOW TRANS;TERMINATE LIST
=Ctrl D    (EOT)
```

Eingabevorgang

```
COPY.LIST /ED/CMMD > /XC/;T LIST

AT 12:00:00 ALL 2 SEC UNTIL 12:02:50 LIST
```

Man kann die Kommando–Datei natürlich bequemer mit einem Editor erstellen! Man beachte, daß durch den Befehl COPY.LIST ein benamter Kopierprozeß (Name = „LIST“) entsteht, der zunächst nicht läuft (wegen T LIST). Mit der letzten Zeile wird der Prozeß „LIST“ dann eingeplant.

Beispiel 2: Ein in PEARL geschriebenes Modul (TEST) soll sich selbst spurlos aus der Verwaltung von **RTOS–UH** eliminieren:

```
MODULE TEST;
SYSTEM; Shell:/XC;
PROBLEM; SPC Shell DATION OUT ALPHIC;
....
ABCD:TASK;
....
 PUT 'UNLOAD TEST*' TO Shell BY A,SKIP;
 SUSPEND;
END;
....
MODEND;
```

! → Die Priorität der Betreuungstask für die „/XC“–Warteschlange hängt von der Priorität des schreibenden Prozesses ab und liegt um eine Einheit höher als dessen Priorität. Dennoch kann der im letzten Beispiel eingebaute SUSPEND–Befehl evtl. zur Ausführung kommen, weil noch andere Aufträge weiter vorne in der Warteschlange stehen können und die mit dem „PUT“-Befehl abgesetzten Anweisungen dadurch verzögert werden.

3.2.5 Zeitliche Hintereinanderschaltung von Befehlen

Bei längerdauernden Aktivitäten benutzt die Shell Sohnprozesse, und diese führen dann zu einer quasi parallelen Abarbeitung mehrer Bedienbefehle. Auf diese Weise können Wartephasen – z. B. wenn der Compiler von der Platte lesen muß – einzelner Aktivitäten zugunsten anderer Bedienbefehle genutzt werden: So wird eine optimale Ausnutzung des Prozessors erreicht. Dennoch kann es wünschenswert sein, verschiedene Bedienbefehle statt wie üblich parallel nun zwangsweise sequentiell ablaufen zu lassen. Prinzipiell gibt es dafür zwei verschiedene Möglichkeiten.

- Mit Hilfe des „`WAIT`“-Befehles wird der Shellprozeß gezwungen, mit der nächsten Anweisung in der Zeile erst zu beginnen, wenn die vorhergehende Bedienanweisung beendet ist. Eine genauere Beschreibung von „`WAIT`“ ist auf Seite 223 zu finden.

  ```
  WAIT;P;LOAD;ACTIVATE TASK1
  ```

  Während der Compilation und beim Laden laufen Sohnprozesse ab, auf die unser Shellprozeß jetzt jeweils wartet. Meistens wird es nachteilig sein, daß der Shellprozeß dadurch während der Bearbeitung der Zeile nicht mehr für andere Aktivitäten zur Verfügung steht. Auf `Ctrl A` reagiert ein primärer Shellprozeß in dieser Zeit nicht, lediglich unser „Notruf“ mit `BREAK` kann ihn aus der Blockierung befreien. Sohnprozesse geben in der Regel einen Fehlerstatus an den übergeordneten Shellprozeß zurück, wenn dieser im „Wait-Mode“ auf ihn wartet. Würde der Compiler einen Fehler im Programm finden, so führt die Shell alle in der Zeile folgenden Befehle nicht mehr aus, sondern schreibt „*Befehl* `: operation failed`“ nach `Stderr`. Das Laden unterbleibt dann, ebenso die Aktivierung.

- Wenn man die Anweisungen statt durch Semikolon durch zwei Minuszeichen trennt, so werden automatisch an den richtigen Stellen sekundäre Shellprozesse erzeugt, die die weitere Bearbeitung der Folgeanweisungen übernehmen. Der Shellprozeß wird also durch die Bearbeitung der Anweisungsfolge nicht blockiert.

  ```
  P--LOAD--ACTIVATE TASK1; DM 500
  ```

  Nach dem Übersetzen verwandelt der PEARL-Compiler sich in einen sekundären Shellprozeß, der die Anweisungen `LOAD--ACTIVATE TASK1` zu bearbeiten hat. Sohnprozesse, die sich in sekundäre Shellprozesse verwandeln, unterdrücken ihre Endemeldungen. Der sekundäre Shellprozeß (er trägt noch den Namen des Compilers, braucht aber nicht mehr seinen ganzen Speicher) führt das Laden allerdings nur aus, wenn er selbst keinen Fehler detektiert hat. Bei dieser Lösung werden fehlerhafte Programme nicht geladen, und die (primäre) Shell bleibt frei für andere Aufgaben: Der `DM 500`-Befehl wird schon in Bearbeitung genommen, während der Compiler läuft. Interessant ist, daß der Lader sich selbst am Ende in einen sekundären Shellprozeß verwandelt und die Anweisung `ACTIVATE TASK1` mit auf den Weg bekommen hat. Die Aktivierung unterbleibt, wenn beim Laden ein Fehler auftrat.

3.2.6 Antwort der Shell im Fehlerfall

Wenn ein unzulässiger Bedienbefehl eingegeben wurde, antwortet der veranlassende Shellprozeß mit einer Meldung nach „`Stderr`“, der Standard-Fehlerausgabedatei (bzw. Gerät), die zum Zeitpunkt der Ausführung des fehlerhaften Befehles vereinbart war. Mit den Bedienbefehlen `ER` und `PER` kann diese Ausgabe umgelenkt werden, z. B. in eine „Alertbox“ des Windowsystemes oder in eine Sammeldatei auf Festplatte etc. Die Fehler führen bis auf wenige Ausnahmen immer zum Abbruch der kompletten Bedienzeile. Bedienbefehle hinter dem fehlerhaften kommen also in der Regel nicht mehr zur Ausführung. Ausnahmen gibt es z. B. beim `UNLOAD`-Befehl, der die Nichtexistenz von Modulen oder Tasks zwar beklagt, aber dennoch weitermacht, weil das gewünschte Ziel ja erreicht wurde.

Die Fehlermeldungen der Shell haben folgendes Aussehen:

Bedienbefehl : *Fehlerinfo*
oder
< *Shellprozessname* > *Bedienbefehl* : *Fehlerinfo*

Die erste Form gilt für primäre Shellprozesse, die zweite für sekundäre. Unter *Bedienbefehl* ist der inkriminierte Bedienbefehl samt allen Parametern – bis zum nächsten Semikolon bzw. dem Zeilenende – zu finden. Die *Fehlerinfo* ist abhängig von der Art des Fehlers und selbsterklärend, bzw. bei den einzelnen Bedienbefehlen beschrieben.

Der optional vorhandene Interpreter für die Shellsprache benutzt zum Teil die Grundmechanismen der Shell, hat aber darüber hinaus weitere eigene Fehlermeldungen, z. B. bei Verletzung von Konventionen der Shellsprache.

3.3 PEARL–codierte Bedienbefehle

Der modulare Aufbau der Shell (Bedieninterpreter) und ein entsprechender Sonderteil im PEARL–Compiler lassen es zu, daß zuladbare oder systemresidente (ROM, Bootdisk) Bedienbefehle in PEARL geschrieben werden können, wobei ein evtl. Parametersatz des Bedienbefehles als Text an das PEARL–Modul transferiert werden kann. Außerdem werden die drei Datenstationen `Stdin`, `Stdout` und `Stderr` des momentanen Shellprozesses für die PEARL–Welt zugänglich gemacht:

1. Standard Input = Eingabegerät der Aufrufershell. Es kann mit dem `I`– und `PI`–Befehl (Vorsicht!) umgelenkt werden.

2. Standard Output = Ausgabegerät der Aufrufershell. Es kann mit dem `O`– und `PO`–Befehl umgelenkt werden.

3. Standard Error = Error–Message–Sammeldatei der Aufrufershell. Es kann mit den Befehlen `ER` und `PER` umgelenkt werden. Meist ist `Stderr` nicht umgelenkt und ist dann das Terminal des Bedieners bzw. das Fenster für die Kommandoeingabe.

Die Shell generiert einen Sohnprozeß, dessen Name und/oder Priorität vorgegeben werden kann. Dieser Sohnprozeß ruft mit dem unten beschriebenen Parametersatz eine PEARL–Prozedur auf. Diese gibt einen Fehlerstatus (Erfolg ja/nein) zurück. Wenn es Shellfolgebefehle (mit 2 Minuszeichen angehängt) gibt und der Fehlerstatus „o.k.“ ist, so verwandelt sich der Sohnprozeß in einen sekundären Shellprozeß, der die Folgebefehle interpretiert. Betrachten wir das Beispiel in der **Abbildung 3.3** (auf Seite 71).

Nach dem Übersetzen (nur mit einem Compiler später als `P16.6-D` wenn die `XHELP`-Unterstützung in `{..}` nach dem Schlüsselwort `PROC` genutzt wird!) und Laden des Shellmodules könnte man dann folgende Bedienbefehle eingeben:

```
xhelp;
```

Auf dem eigenen Terminal erscheint dann Zeile für Zeile eine Liste der Bedienbefehle und diese enthält folgende Zeile:

```
ECHOTX gibt Aufruftext aus.
```

Nun benutzen wir den neuen Befehl:

```
echotx Dies ist ein Test-Text;
```

Auf dem eigenen Terminal erscheint dann die folgende Zeile:

```
Dies ist ein Test-Text
```
(+CR wg. SKIP)

Anderes Aufrufbeispiel:

```
O /ed/murks; echotx Soll in den Murks-file;
```

Nun wurde die Zeile „`Soll in den Murks-file`“ in den File `/ed/murks` geschrieben.

Weiteres Aufrufbeispiel:

```
echotx.z2 prio 12 PEARL-Compilation folgt -- P;
```

Es erscheint der Text „`PEARL-Compilation folgt`“ und anschließend wird der PEARL–Compiler gestartet. Nur zur Demonstration wurde ein Sohnprozeßname (`z2`) und eine Priorität angegeben.

`--->` Der übermittelte Text beginnt immer mit dem ersten Zeichen hinter der Lücke nach dem eigentlichen Bedienbefehl und endet vor dem Semikolon oder dem Zeichen `--`, das den evtl. Folgebefehl einleitet. Der Längenparameter erhält genau den dazu passenden Wert. Es ist Aufgabe des Programmierers, den Text — etwa mit passenden Unterprogrammen — selbst zu analysieren, um z. B. Zahlenwerte, Parameterbezeichner etc. zu erkennen.

`--->` Der so erzeugte Shell–Befehl ist i. a. nur dann mehrbenutzerfest, wenn im Shellmodule weder Tasks noch globale (d. h. auf Modulebene deklarierte) Variable benutzt werden. Gleiches gilt auch für in einem evtl. vorhandenen `SYSTEM`–Teil deklarierte `DATION`s. Der Grund liegt einfach darin, daß derartige Objekte nur einmal vorhanden sind und zu einem Zeitpunkt nur einem einzigen Herrn dienen können. Gleichwohl sind natürlich auch Fälle denkbar, in denen es für mehrere Nutzer nur eine Server–Task geben darf, die sich ihre Aufträge z. B. über einen Ringpuffer von den verschiedenen Nutzern holt (Nur aus diesem Grund läßt der PEARL–Compiler auch in den Shellmodulen Modulvariable und `DATION`s zu).

---> Die Bildung einer „Scheibe“, um die Bedienbefehle EPROM– (oder Bootdisk–) resident zu machen, geschieht auf die übliche Weise, wie sie später (z. B. Seite 630) noch beschrieben wird, durch Angabe von `CODE`– und `VAR`–Adressen. (Wobei – s. o. – die 13–er Scheibe des Variablen-Blockes evtl. leer ist und nicht in das EPROM übernommen zu werden braucht). Der Compiler gibt eine Meldung aus, ob der Einsatz des `PROM`–Befehls (bzw. des Linkers) nötig ist oder ob das entstandene Modul frei verschieblich ganz unkompliziert im EPROM (oder auf der Bootdisk) abgelegt werden darf.

---> Shell–Module dürfen nur entladen werden, wenn sichergestellt ist, daß kein Kommando dieses Modules mehr ausgeführt wird. Ein `UNLOAD` *modname*`*` entlädt nur das Shell–Modul und nicht auch die evtl. auf diesem Code noch laufenden Sohnprozesse!

```
SHELLMODULE test;   ! Eroeffnung eines 'Shellmodules'

   Sages:'ECHOTX';
   ! Bedienbefehl 'ECHOTX' wird definiert und mit interner
   ! PEARL-Prozedur Sages verbunden xxxx:'YYYYYY'; Evtl.
   ! weitere solche Verbindungen: PEARL-Name der Prozedur
   ! links, rechts vom Doppelpunkt der String des neuen
   ! Bedienbefehles. Nur Grossbuchstaben bei YYYYYY!

PROBLEM;  !   SYSTEM oder PROBLEM beendet den speziellen
          !   Shell-Definitionsteil

Sages: PROC{'gibt Aufruftext aus.'} ((Stdin,Stdout,Stderr)
            DATION INOUT ALPHIC IDENT,
            Length FIXED, Text CHAR(255)) RETURNS(BIT(1));

    !  Bis auf die Formalparameternamen und den evtl. Text
    !  fuer XHELP ({}) muss die Prozedurdefinition immer
    !  genau so aussehen. Oft ist aber eine individuelle
    !  Richtungsangabe fuer die 3 Dations zweckmaessig:
    !  Stdin DATION IN ..., etc., da ja evtl. ein anderes
    !  Geraet als das Terminal gemeint sein kann

   DCL Istokay BIT(1) INIT ('1'B);

   OPEN Stdout;
   Istokay = Istokay AND ST(Stdout) EQ 0;
   PUT Text TO Stdout BY A(Length), SKIP;
   Istokay = Istokay AND ST(Stdout) EQ 0;
   CLOSE Stdout;
   Istokay = Istokay AND ST(Stdout) EQ 0;
   /* Info an Shell ob alles gelungen ist */
   RETURN(Istokay);  ! wird von Shell ausgewertet, wenn
                     ! Folgekommando mit '--' folgt oder
                     ! auf das Ende dieses Kommandos mit
                     ! 'WAIT' gewartet wird.

END;     ! An Shell angeschlossene Prozedur  Sages
MODEND;  ! Ende des Shell-Modules
```

Abbildung 3.3 Beispiel für ein Shell–Modul in PEARL

3.4 Besonderheiten bei transienten Kommandos

Wird von der Shell die Ausführung eines Bedienbefehles verlangt, den sie in den Ebenen 3 ... 6 nicht finden konnte, und existiert auch keine Task mit diesem Namen in der Systemverwaltung, so wird vermutet, daß es sich um ein „transientes Kommando“ oder um einen „Skript“ in Shellsprache handelt (siehe 3.5.2 auf Seite 76). Wenn ein Systemlader vorhanden ist (er kann darum nicht als transientes Kommando benutzt werden!), wird zunächst die Annahme „transientes Kommando“ probiert, anderenfalls erfolgt der Übergang in Ebene 8 zum Shellsprachinterpreter – falls vorhanden.

Wenn der Bedienbefehl mit „`/`“ beginnt, so wird angenommen, daß ein Gerätebezeichner folgt, z. B. `/F0/...`, anderenfalls wird dem Befehl nacheinander jedes vorhandene Execution-Directory vorangestellt. Die sich so ergebenden File-Bezeichner werden versuchsweise geöffnet, und der Inhalt wird auf Ladbarkeit (Beginn mit „`S0`“) geprüft. Wird ein File des Namens gefunden und ist er ladbar, so generiert die Shell einen Sohnprozeß als Lader, der den Namen des Befehles plus laufender Nummer erhält. Die Shell wartet auf diesen Prozeß! Anschließend wird das Kommando im hinzugeladenen transienten Modul gesucht. Nach Bearbeitung des Befehles verschwindet das Modul wieder aus dem Speicher.

Wenn man einen Befehl häufiger benötigt, so sollte man zur Zeitersparnis mit Hilfe des ganz normalen „`LOAD`“-Befehles das Shellmodule laden. Es wird dann zu einer Erweiterung der Ebene 6 (Shell-Modul im RAM) und zukünftig nicht mehr transient ausgeführt.

Beispiel:

Das Execution–Directory sei „`/F0/CMD`“. Hier befindet sich ein übersetztes Shellmodul (S–Records) mit dem Kommando `MORE`. Die Datei muß auch den Namen `MORE` haben.

Es müssen einige **Restriktionen** für transiente Kommandos beachtet werden. Die Angaben unten gelten für neuere Versionen von **RTOS–UH**, erschienen nach Februar 1994.

1. Der Filename muß mit dem Kommandonamen übereinstimmen und darf nur Großbuchstaben enthalten. Bei neueren Systemen werden alle Kleinbuchstaben des eingegebenen Befehlsnamens in Großbuchstaben verwandelt. Erst danach wird der File gesucht. Mit Hilfe des „`LINK`“-Befehles können weitere alternative Filebezeichner angelegt werden, falls das Shellmodul mehrere Befehle beherbergt.

   Beispiel: `xy123` sucht nach File `XY123`.

2. Bei vorhandenen Execution-Directories muß nur der Kommandoname eingegeben werden. Im Beispiel:

   `More;` bei `XD=/F0/CMD`

3. Ohne Execution-Directory muß die gesamte Pathlist (inklusive Device und Kommandoname) eingegeben werden:

   `/F0/CMD/MORE;` oder auch
   `/f0/cmd/more;` wegen Umwandlung in Großbuchstaben.

4. Das Testlesen erfolgt im I/O-Mode „no errors“. Wenn der fileverwaltende I/O-Prozeß diese Unterdrückung nicht beherrscht, kann es zu von ihm produzierten Fehlermeldungen, z. B. „`FILE-NOT-FOUND`“ kommen – obwohl die Suche anschließend erfolgreich ist.

5. Falls beim Laden eines transienten Kommandos die primäre Shell über den „Notruf“ der `BREAK`–Taste gerufen wird, so kümmert sie sich nicht weiter um ihren Sohnprozeß. Gleiches gilt, wenn ein sekundärer Shellprozeß von außen abgebrochen wird, während er auf den transienten Lader wartet. Der Sohnprozeß (Lader) führt die Ladeoperation zu Ende, aber der transiente Bedienbefehl kommt nicht mehr zur Ausführung. Dadurch lebt nun aber leider niemand mehr, der den Entladevorgang ausführen könnte. Zur Speicherersparnis empfiehlt es sich, das Modul dann gelegentlich von Hand zu entladen, denn die vom transienten Lader geladenen Shellmodule können nicht als Shellerweiterung der Ebene 6 benutzt werden. (Für Insider: die AEB1-Scheibe wird vom Transientlader neutralisiert, damit niemals mehr als ein Shellprozeß auf dem Code laufen kann.)

6. Sehr wenige Befehle, z. B. „`MSFILES`“ und „`RTOSFILES`“ lassen sich prinzipbedingt nicht durch transienten Aufruf benutzen, weil deren Code noch nach Abarbeitung des Kommandos benötigt wird. Ansonsten kann man alle S-Rekords, die als Scheibe für die Shell-Ebenen 3 ... 6 geeignet sind, auch transient benutzen.

3.5 Die Shell-Sprache

3.5.1 Aufruf von Shellskripten

Programme in Shellsprache werden wie in der UNIX-Welt auch in **RTOS–UH** „Skripte“ genannt. Von H. Husmann stammt die Software, mit der solche Skripte auf drei verschiedene Arten zur Ausführung gebracht werden können:

1: *path* [*positpara*]
2: `EX` [`.`*sonprocname*] [`PRIO` *integer3*] [`SZ` *hexnum6*] [*path*] [*positpara*]
3: `SHELL` [`PRIO` *integer3*] [`SZ` *hexnum6*] [*path*] [*positpara*]

Beim Aufruf eines Shell-Skriptes wird ein flüchtiger, unabhängiger Sohnprozeß generiert, der den über *path* angesprochenen Textfile interpretiert.

1 Ist *path* nur ein Name, so wird eine solche Datei in den Execution-Directories gesucht. Als „*sonprocname*“ wird „*name/xx*“ generiert, wobei „*xx*“ eine zweistellige Hexadezimalzahl mit automatischer Weiterschaltung ist. Steht das auszuführende Skript nicht in den Execution-Directories, ist bei *path* der vollständige Pfad anzugeben.

2 Mit dem Aufruf eines Skriptes über die Befehle `EX` oder auch (in der Langform) `EXECUTE` können zusätzlich der Sohnprozeßname (*.sonprocname*), die Priorität des Sohnprozesses (`PRIO`) und die Arbeitsspeichergröße (`SZ`) festgelegt werden. Fehlt der Sohnprozeßname, wird der Defaultname „`EX/`*xx*“ vergeben, wobei „*xx*“ eine zweistellige Hexadezimalzahl mit automatischer Weiterschaltung ist. Ist bei *path* nur ein Name angegeben, wird eine gleichnamige Datei im Working-Directory gesucht.

3 Beim Aufruf über den Befehl `SHELL` gelten außer beim Sohnprozeßnamen die gleichen Aufrufparameter wie beim Aufruf über `EX`. Als Name erhält der Sohnprozeß „`#BSH`*xx*“, wobei *xx* die Usernummer des Aufrufers ist. Zusätzlich wird der Sohnprozeß als sekundäre Shell in das Userenvironment eingetragen und damit beim Anschlag der Taste „`CTRL A`“ des Users fortgesetzt. Die primäre Shell ist dann nur noch über die „`BREAK`“-Taste erreichbar. Das SHELL-Skript sollte in einer Endlosschleife Befehle einlesen, ausführen und sich dann für den nächsten Anschlag der Taste „`CTRL A`“ suspendieren. Damit kann man sich eine Shell mit eigenem Environment und geringerer Priorität einrichten. Die sekundäre Shell kann mit dem `EXIT`-Befehl beendet werden. Nach einem Warmstart läuft das SHELL-Skript neu an und bleibt als sekundäre Shell aktiv. Pro User ist nur eine sekundäre Shell einrichtbar, der SHELL-Befehl darf nicht gestapelt abgesetzt werden.

Speicher: `SZ` *hexnum6* oder `SZ=`*hexnum6*. Mit *hexnum6* kann der dynamische Arbeitsspeicher des Sohnprozesses bestimmt werden (Default und Minimum: `SZ=2000`).

path: Der Parameter *path* muß angegeben werden und bezeichnet die auszuführende Datei. Es wird eine Kopie mit dem Namen `/ED/filename/`*xxxx* angelegt, um das mehrfache, parallele Ablaufen eines Skriptes zu ermöglichen.

positpara: Einem Shell-Skript können beim Aufruf „Positionsparameter“ mitgegeben werden. Der in *positpara* enthaltene Text wird wortweise (durch Leerzeichen getrennt) den Variablen `$1` bis `$n` zugewiesen. Sollen Parameter Leerzeichen enthalten, sind sie in Apostrophs einzuschließen.

Beispiele:

```
RMD /ed/*
```

Die Datei `RMD` wird in den aktuellen Execution-Directories gesucht und dann interpretiert. Der Text „`/ed/*`“ ist im Skript über die Shell-Variable `$1` erreichbar.

```
EX PRIO 40 SZ 4000 /H0/TEST parameter1 'parameter 2'
```

Das Shell-Skript in der Datei `TEST` auf dem Festplattenlaufwerk `H0` wird interpretiert. Die `EX/`*xx* Subtask besitzt die Priorität 40 und erhält `$4000` Bytes Speicher. Die Texte „`parameter1`“ und „`parameter 2`“ sind über die Shell-Variablen `$1` und `$2` im Shell-Skript erreichbar.

```
SHELL /H0/XD/SHELL
```

Es wird eine sekundäre Shell für den aufrufenden User eingerichtet. Im einfachsten Fall könnte das Skript `SHELL` folgendermaßen aussehen:

```
: Privates Environment setzen,
: z.B. mit den Befehlen CD, CXD, ...

WHILE TRUE            : Endlosschleife
DO
  SUSP;               : warten auf CTRL-A
  ECHO -N >;          : Prompt ausgeben
  READ Kmd;           : Kommando lesen
  IF EXEC $Kmd; THEN  : Kommando ausfuehren und
  FI;                 : Fehlerabbruch durch
                      : IF...THEN FI; verhindern
DONE;
```

Hinweis: Der Interpretercode ist wiedereintrittsfest, so daß beliebig viele Shell-Skripte gleichzeitig abgearbeitet werden können (sofern noch dynamischer Speicher zur Verfügung steht).

3.5.2 Sprachumfang Shell-Interpreter

Ein Shell-Skript ist eine Aneinanderreihung von Anweisungen, wobei eine Anweisung ein Bedienbefehl oder ein Kommando aus dem Befehlssatz des Shell-Interpreters sein kann. Innerhalb eines Interpreterbefehls können Bedienbefehle an jeder Stelle stehen, an der eine Anweisung stehen darf. Beim Auftreten eines Fehlers in einem Bedienbefehl wird der logische Wert „falsch“ zurückgegeben. Bei Bedienbefehlen, die eine Subtask generieren, kann mit dem Shellbefehl „`WAIT`“ auf die Beendung gewartet werden. Nur im „Waitmode“ erreicht ein Fehler der Subtask das Shellskript.

Die Syntax der Shell-Skripte ist an die der „UNIX Bourne-Shell“ angelehnt. Im folgenden werden alle Kommandos des Shell-Interpreters mit ihrer Syntax und ihrer Bedeutung erklärt. Der Shell-Interpreter akzeptiert Schlüsselwörter in Groß- und Kleinschreibung. Eine vollständige Liste der Schlüsselwörter befindet sich in Tabelle 3.5.10.8, in der Syntaxbeschreibung sind sie durch die Schriftart „`teletype`“ gekennzeichnet. In eckigen Klammern stehende Teile der Syntaxbeschreibung sind optional und können auch entfallen. Beschreibungsteile in geschweiften Klammern können beliebig oft oder überhaupt nicht eingesetzt werden. Alternativen sind in Klammern gesetzt und durch das Zeichen | getrennt.

Über den Befehl **.**[/*Gerät* /*Pfad* /]*Name* können sogenannte „Subskripte“ aufgerufen werden. Fehlen *Gerät* und *Pfad*, wird die Datei *Name* in den Execution-Directories gesucht. Ein Subskript ist ein Unterprogrammaufruf ohne Übergabeparameter. Es kann jedoch auf alle Variablen des Aufrufers zugegriffen werden. Mit dem Ende des Subskriptes (`EXIT`-Befehl) kehrt man in das aufrufende Skript zurück. Für ein Subskript wird kein eigener Prozeß gebildet.

3.5.3 Kommentare

Kommentare beginnen mit der Zeichenfolge Doppelpunkt Leerzeichen und enden mit dem Zeilenende. Sie können beliebig in den Prozedurtext eingestreut werden. Die Zeichenkette Doppelpunkt Leerzeichen verliert ihre Bedeutung nur, wenn sie in Anführungszeichen („:“) oder in Apostrophs (´:´) eingeschlossen wird.

3.5.4 Metazeichen

Zeichen mit spezieller Bedeutung werden Metazeichen genannt. Dazu gehören die Zeichen: `*`, `?`, `$`, `` ` ``, `:`, `\`, “, ´, `;`, `CR`. Die Zeichen Stern und Fragezeichen sind sogenannte Wildcards und werden bei Zeichenvergleichen eingesetzt. Der Stern repräsentiert dabei beliebig viele beliebige Zeichen, während das Fragezeichen genau ein beliebiges Zeichen vertritt. Das Dollarzeichen kennzeichnet mit darauffolgendem Namen den Wert einer Variablen (siehe 3.5.5). Durch das Einschließen eines Bedienbefehls in Hochkommata (z. B. `` `S` ``) wird seine Ausgabe in eine Variable umgelenkt (siehe 3.5.5.1). Das Voranstellen des Zeichens `\` bewirkt für ein folgendes Metazeichen, daß es seine besonderen Eigenschaften verliert und als normales Textzeichen interpretiert wird. Folgt dem Zeichen `\` eine dezimale Zahl, so wird ein Zeichen mit dem der Zahl entsprechenden ASCII-Code eingesetzt (z. B. `\4` entspricht dem Zeichen `CTRL-D`; `\42` entspricht dem Zeichen „`*`“). Einige fest vereinbarte Sonderzeichen erreicht man über `\B`, `\F`, `\N`, `\O`, `\R` und `\T`, siehe Tabelle Seite 94. Durch Einschließen in Anführungszeichen (“ ... “) werden folgende Metazeichen zu normalem Text

```
* ? : ; CR \´
```

durch Einschließen in Apostrophs (´ ... ´) dagegen

```
* ? : ; CR \ $ ` “
```

3.5.5 Shell-Variablen

Shell-Variablen enthalten grundsätzlich nur Zeichenketten (im folgenden auch als „String“ bezeichnet). Auch Zahlen, die mit dem `EXPR` Kommando (siehe 3.5.9.2 auf Seite 86) verarbeitet werden, sind als String gespeichert. Den Wert einer Variablen erhält man, indem man dem Namen ein Dollarzeichen (`$`) voranstellt. Variablennamen müssen mit einem Buchstaben oder einer Ziffer beginnen und dürfen nur Buchstaben und Ziffern enthalten.

Die Variablen `$1`, `$2`, ... werden beim Aufruf des Shell-Interpreters mit den „Positionsparametern“ der Kommandozeile besetzt. Diese Positionsparameter enthalten der Reihe nach, bei `$1` beginnend, die einzelnen Textworte der Kommandozeile:

```
/HO/XD/XYZ Dies ist ein test
```

erzeugt die Inhalte `$1` = ´`Dies`´ , `$2` = ´`ist`´ , `$3` = ´`ein`´ usw.

In der Variablen `$#` findet man die Anzahl der Positionsparameter. Außerdem werden noch `$?`, `$@` und `$$` vom Interpreter gesetzt (siehe Tabelle).

$1- $n	Positionsparameter der Aufrufzeile
$#	Anzahl der Positionsparameter
$@	alle Positionsparameter durch ein Leerzeichen getrennt
$?	Austrittsstatus des zuletzt ausgeführten Bedienbefehls Kein Fehler –> $?='0'; Fehler –> $?='1'
$$	Name der Interpreter-Subtask
$0	Name der interpretierten ED-Datei
$EOF	Austrittsstatus des letzten READ-Befehls

Vorbesetzte Shell-Variablen

3.5.5.1 Wertzuweisung an Shellvariable

Shell-Variablen kann durch die Verwendung des Gleichheitszeichens ein Wert zugewiesen werden. Das Ende der zugewiesenen Zeichenkette ist entweder das Zeilenende oder das Semikolon.

Syntax: *variablenname* `=` *string* (`;` | `CR`)

```
DIR=/F0/SYS  : $DIR wird /F0/SYS zugewiesen
CD $DIR      : entspricht CD /F0/SYS
```

```
TEXT = ´Diese Zeile enthaelt Metazeichen: * ? $ ` : \ " ; ´
TEXT = "Diese Zeile enthaelt Metazeichen: * ? : \ ´ ; "
```

Die Verkettung von Zeichenketten ist durch einfaches Aneinanderhängen möglich. Folgt einem Variablennamen bei einer Verkettung direkt ein Buchstabe oder eine Ziffer, dann muß er durch geschweifte Klammern vom folgenden Text abgegrenzt werden.

```
DIR = /F0/SYS          : $DIR wird /F0/SYS zugewiesen
DAT = $DIR/DATEI;      : $DAT wird /F0/SYS/DATEI zugewiesen
COPY $DAT>/ED/TEXT     : /F0/SYS/DATEI wird nach /ED/TEXT kopiert

DIR2 = $DIRTEM;        : DIR folgt ein Buchstabe, daher wird die
                       : Variable mit dem Namen DIRTEM angesprochen

DIR2 = ${DIR}TEM;      : $DIR2 wird /F0/SYSTEM zugewiesen
COPY $DIR2>/ED/SYS     : /F0/SYSTEM wird nach /ED/SYS kopiert
```

Es ist weiterhin möglich, die Ausgabe eines Bedienbefehls in eine Variable einzulesen. Dazu muß der Bedienbefehl in Hochkommata eingeschlossen werden:

```
Speicher = `S`    : Die Ausgabe des S Kommandos wird der
                  : Variablen $Speicher zugewiesen
```

3.5.5.2 Implizite Wertzuweisung

Syntax: *Variablenname1* = ${ *Variablenname2* (– | = | ?)*String* }

Der Wert einer nicht gesetzten Variablen ist die leere Zeichenkette. Wenn z. B. `$WERT` nicht gesetzt ist, würde der Befehl `VAR = $WERT`, der Variablen `$VAR` also nichts zuweisen. Ein impliziter Wert kann mit Hilfe der folgenden Notationen zugewiesen werden:

Bsp. 1:
```
DIR = ${DIR2 - DIR2 ist nicht gesetzt}
```

Ist `$DIR2` gesetzt, wird der Wert von `$DIR2` zugewiesen, anderenfalls wird `$DIR` der Text „`DIR2 ist nicht gesetzt`“ zugewiesen.

Bsp. 2:
```
DIR = ${DIR2 = /FO/SYS}
```

Ist `$DIR2` gesetzt, wird der Wert von `$DIR2` zugewiesen, anderenfalls wird `$DIR2` und dann auch `$DIR` der Text „`/FO/SYS`“ zugewiesen.

Bsp. 3:
```
DIR = ${DIR2 ? DIR2 ist nicht gesetzt}
```

Ist `$DIR2` gesetzt, wird der Wert von `$DIR2` zugewiesen, anderenfalls wird die Meldung „`DIR2 ist nicht gesetzt`“ an das Standard-Ausgabegerät ausgegeben, und die Ausführung wird abgebrochen.

3.5.6 E/A-Befehle

3.5.6.1 Der Ausgabe-Befehl ECHO

Syntax: `ECHO [-N]` *string* (`;` | `CR`)

Der dem `ECHO`-Kommando folgende *string* wird auf das „Standard-Out-Gerät“ des Skriptes ausgegeben. Mit dem „`O`“-Kommando kann die Ausgabe umgelenkt werden (siehe Seite 179). Der auszugebende String wird durch das Zeilenende oder durch ein Semikolon beendet. Ohne die `-N` Option wird ein Zeilenvorschub angehängt, mit `-N` erfolgt kein Zeilenvorschub.

Beispiel: `DIR=/FO/SYS; ECHO \7´Der Wert von $DIR ist : ´$DIR`

Es wird der Text „`Der Wert von $DIR ist : /FO/SYS`“ mit Bell (`\7` –> ASCII-Code 7 entspricht Bell) ausgegeben.

Beispiel: `O /ED/TEXT; ECHO -N "Die Uhrzeit ist : `CLOCK`"`

Der Text „`Die Uhrzeit ist : xx:yy:zz ...`“ wird in die Datei `/ED/TEXT` geschrieben.

3.5.6.2 Der Einlese-Befehl READ

Syntax: `READ` *variable* {*variable* } (`;` | `CR`)
oder `READ [-N|-E]` *variable* (`;` | `CR`)

Der `READ`-Befehl liest Eingaben vom „Standard-In-Gerät“ des Users (Die Eingabe kann durch das „`I`“-Kommando umgeleitet werden, siehe Seite 152). Erreicht man beim Lesen von einer Datei das Dateiende, enthält die Shell-Variable `$EOF` eine „`1`“, ansonsten eine „`0`“. Ohne `-N` Option wird die eingelesene Zeile einer oder mehreren Variablen, die dem `READ`-Befehl folgen, zugewiesen. Dabei wird den ersten Variablen jeweils ein Wort (Wortgrenze ist das Leerzeichen) und der letzten Variablen der Rest der Zeile zugewiesen. Als Zeilenende gelten „`Carriage-Return`“=`$0D`, „`Linefeed`“=`$0A` und „`End of Text`“=`$04`, wobei das Zeichen „`Carriage-Return`“ nicht zugewiesen wird.

Mit `-N` Option wird einer Variablen die komplette eingelesene Zeile einschließlich evtl. vorhandener führender Leerzeichen zugewiesen. Mit `-E` Option wird ein Einzelzeichen gelesen, dabei werden auch alle Steuerzeichen zugewiesen.

Beispielprogramm: `READ X Y Z`
Eingabe: `Diese Zeile wird gelesen (CR)`
Ergibt: `$X: Diese  $Y: Zeile  $Z: wird gelesen`

Beispielprogramm: `READ -N X`
Eingabe: `Diese Zeile wird gelesen (CR)`
Ergibt: `$X: Diese Zeile wird gelesen`

Beispielprogramm: `REWIND /ED/TEXT; I /ED/TEXT; READ ZEILE`
Ergibt: `$ZEILE:` Erste Zeile aus /ED/TEXT

3.5.7 Ablaufsteueranweisungen

3.5.7.1 Die IF-Anweisung

Syntax: `IF` { *Anweisung* }
`THEN` { *Anweisung* } [`ELSE` { *Anweisung* }] `FI` (`;` | `CR`)

Die Anweisungen hinter dem `IF`-Kommando werden als Bedingungsfolge ausgeführt. Es sind sowohl Interpreteranweisungen als auch Bedienbefehle anwendbar. Die Bedingungsfolge wird beim ersten Auftreten des Ergebnisses falsch oder fehlerhaft abgebrochen, und sofern vorhanden, der `ELSE`-Zweig ausgeführt. Liefert sie den Wert „wahr“ oder „fehlerfrei“, wird der `THEN`-Zweig ausgeführt. Danach wird die Ausführung der Shell-Prozedur hinter dem `FI`-Kommando fortgeführt. Tritt innerhalb der Anweisungsfolge die Bedingung „unwahr“ oder ein Fehler auf, wird die `IF`-Anweisung abgebrochen und der Wert „unwahr“, ansonsten „wahr“ zurückgegeben. Die Befehlsfolge `ELSE IF` kann als „`ELIF`“ abgekürzt werden.

```
IF TEST $1 = TEXT                    : wenn $1 gleich TEXT  ist
THEN                                 : dann
 ECHO ´$1 ist gleich TEXT´           : Ausgabe: $1 ist ...
ELSE                                 : sonst
 ECHO ´$1 war nicht gleich TEXT´     : Ausgabe: $1 war ...
 1 = TEXT                            : $1 = TEXT
FI                                   : Ende der IF-Anweisung
```

3.5.7.2 Die CASE-Anweisung

Syntax: `CASE` *Variable* `IN`
{*Muster* { | *Muster* } `)` {*Anweisung*} `;;` }
`ESAC` (`;` | `CR`)

Die `CASE`-Anweisung gestattet eine Mehrfachverzweigung. Sie vergleicht den Wert der Variablen mit den Mustern. Das Zeichen „|“ ist als „oder“ zu verstehen. Stimmt der Wert der Variablen mit einem Muster überein, wird die zugehörige Anweisungsfolge ausgeführt und danach die `CASE`-Anweisung verlassen. Im Muster dürfen auch die Wildcards „Stern“ und „Fragezeichen“ verwendet werden.

Beispiel:

```
CASE $# IN
 1  |  2  ) ECHO "ein oder zwei Parameter" ;;    : $# ist 1 oder 2
 ?      ) ECHO "drei bis neun Parameter" ;;      : $# ist ein Zeichen
 *      ) ECHO "mehr als neun Parameter" ;;      : $# ist beliebig
ESAC
```

3.5.7.3 Die FOR-Anweisung

Syntax: `FOR` *Steuervariable* [`IN` [`-`(`W`|`L`)] *String* (; | CR)]
`DO` {*Anweisung* } `DONE` (; | CR)

Ohne den `IN`-Befehlsteil werden der *Steuervariablen* nacheinander die Positionsparameter (`$1 - $n`) zugewiesen. Mit dem `IN`-Befehlsteil und `-L` („Line“) oder fehlender Option wird der String zeilenweise, mit `-W` („Word“) Option wortweise, der Steuervariablen zugewiesen. Die Anweisungsfolge wird bei jedem Schleifendurchlauf ausgeführt, bis alle Werte zugewiesen wurden. Der aktuelle Wert der Steuervariablen ist über `$`*Steuervariable* erreichbar.

Beispiel 1:

```
FOR VAR IN `DIR /HO`     : liest die Ausgabe von DIR /HO  zeilenweise
DO                       : führe aus
  ECHO $VAR;             : gibt die aktuelle Zeile aus
DONE                     : Ende der FOR-Schleife
```

Beispiel 2:

```
FOR VAR                  : liest die Positionsparameter $1 - $n
DO                       : führe aus
  ECHO $VAR;             : gibt nacheinander $1 - $n aus
DONE                     : Ende der FOR-Schleife
```

3.5.7.4 Die WHILE- und die UNTIL-Anweisung

Syntax: `WHILE` {*Anweisung*} `DO` {*Anweisung*} `DONE` (`;` | `CR`)
oder: `UNTIL` {*Anweisung*} `DO` {*Anweisung*} `DONE` (`;` | `CR`)

Die dem `WHILE`-Kommando folgenden Bedingungsanweisungen werden ausgeführt und beim ersten Auftreten der Bedingung „unwahr“ oder „Fehler“ abgebrochen. Solange die Bedingungsanweisungen den Wert „wahr“ liefern, wird der mit dem `DO`- und `DONE`-Befehl eingeschlossene Schleifenkern wiederholt. Bei der `UNTIL`-Variante wird der Schleifenkern solange ausgeführt, bis die Bedingungsfolge den Wert „wahr“ liefert. Sowohl bei der `WHILE`– als auch bei der `UNTIL`–Anweisung wird vor dem Schleifenkern die Bedingungsanweisungsfolge ausgeführt.

Beispiel:

```
DATE =´0´
UNTIL ER /NIL; DATESET $DATE  : bis das Datum richtig gesetzt ist
DO                            : führe aus
 IF TEST $DATE !=´0´; THEN
  ECHO ´Die Eingabe war falsch!´
 FI
 ECHO -N ´Geben Sie das Datum in der Form TT-MM-JJJJ ein :´
 READ DATE                    : lies DATE  vom Terminal
DONE                          : Ende der UNTIL-Schleife
```

3.5.8 Bedingungs-Anweisungen

Bedingungsanweisungen werden in der `WHILE`-, `UNTIL`- und in der `IF`-Anweisung benötigt, um Programmverzweigungen zu realisieren. Es kann grundsätzlich jede Anweisung des Befehlssatzes und jeder Bedienbefehl als Bedingung dienen. Ist die Anweisung keine spezielle Bedingungs-Anweisung aus diesem Kapitel, dann wird bei fehlerfreier Ausführung „wahr“ und beim Auftreten eines Fehlers „falsch“ zurückgegeben.

3.5.8.1 TRUE- und FALSE- Anweisung

Syntax: (`TRUE` | `FALSE`) (; | CR)

Die `TRUE`- und `FALSE`-Anweisung kann beim Programmtest eingesetzt werden, um bestimmte Programmzweige zwingend zu durchlaufen. Dabei liefert `TRUE` den Wert „wahr“ und `FALSE` den Wert „falsch“.

Beispiele für Endlosschleifen:

```
WHILE TRUE          UNTIL FALSE
DO                  DO
DONE                DONE
```

3.5.8.2 Die TEST-Anweisung

Syntax: `TEST` *log_Ausdruck* {`(-A|-O|-E)` *log_Ausdruck* } (`;` | `CR`)

log_Ausdruck : `[!]` (`(-Z|-N)` `$`*Variablenname* |
String_1 [!] = *String_2* |
Zahl_1 `(-EQ|-NE|-GT|-GE|-LT|-LE)` *Zahl_2*)

Das `TEST`-Kommando liefert in Abhängigkeit vom folgenden Ausdruck den logischen Wert „wahr“ oder „falsch“. Logische Ausdrücke können miteinander „UND“ (`-A`), „ODER“ (`-O`) oder „EXKLUSIV-ODER“ (`-E`) verknüpft werden. Bei einem `TEST`-Befehl über mehrere Zeilen sollte man `-A`, `-O` oder `-E` an das Zeilenende setzen, um die Beendigung durch das `CR` aufzuheben.

Die Ausdrücke werden von links nach rechts abgearbeitet, es können jedoch durch Einschließen in Klammern Gruppen gebildet werden. Das Ausrufungszeichen („`!`“) bewirkt eine Negation des folgenden Ausdrucks. Die logischen Ausdrücke haben folgende Bedeutung:

`-Z $`*Variablenname*: wahr, wenn die Variable nicht existiert
: oder die Länge Null ist
`-N $`*Variablenname*: wahr, wenn die Länge ungleich Null ist
String_1 `=` *String_2* : wahr, wenn *String_1* gleich *String_2* ist
String_1 `!=` *String_2*: wahr, wenn *String_1* ungleich *String_2* ist
Zahl_1 `-EQ` *Zahl_2* : wahr, wenn *Zahl_1* gleich *Zahl_2* ist
Zahl_1 `-NE` *Zahl_2* : wahr, wenn *Zahl_1* ungleich *Zahl_2* ist
Zahl_1 `-GT`*Zahl_2* : wahr, wenn *Zahl_1* größer *Zahl_2* ist
Zahl_1 `-GE`*Zahl_2* : wahr, wenn *Zahl_1* größergleich *Zahl_2* ist
Zahl_1 `-LT`*Zahl_2* : wahr, wenn *Zahl_1* kleiner *Zahl_2* ist
Zahl_1 `-LE`*Zahl_2* : wahr, wenn *Zahl_1* kleinergleich *Zahl_2* ist

Beispiele:

```
IF TEST -Z $VAR                  : wenn $VAR nicht gesetzt ist
...
IF TEST -N $VAR                  : wenn $VAR gesetzt ist
...
IF TEST $FNAME = /ED/* -O $FNAME = /ed/*
: wenn $FNAME gleich /ED/irgendwas oder /ed/irgendwas ist
...
IF TEST $# -GE 2 -A $# -LE 4
: wenn die Anzahl der Positionsparameter zwischen 2 und 4 liegt
...
WHILE TEST $ZAEHLER -LE 10       : solange $ZAEHLER kleiner 10
DO  ...
```

3.5.9 Zeichenketten-Behandlung

3.5.9.1 Die LEN-Anweisung

Syntax: `LEN(`$ *Variablenname*`)` (`;` | `CR`)

Die `LEN`-Anweisung berechnet die Länge der folgenden Variablen, sie kann überall dort stehen, wo ein String oder eine Zahl stehen darf.

```
LAENGE = LEN($VAR)           : LAENGE wird die Länge von VAR
                             : zugewiesen
IF TEST LEN($VAR) -LT 10     : wenn VAR kürzer als 10 ist
```

3.5.9.2 Die EXPR-Anweisung

Syntax: `EXPR` *arithmetischer Ausdruck* (`;` | `CR`)

Der dem `EXPR`-Befehl folgende String wird als arithmetischer Ausdruck angesehen. Es sind die Operationen +, –, *, / und % (% `--` `>` Bestimmung des Divisionsrestes) erlaubt. Bei Addition und Subtraktion sind Integerzahlen von `-30000000` bis `+30000000` erlaubt, bei der Multiplikation und Division von `-32000` bis `+32000`. Der Ausdruck wird von rechts nach links abgearbeitet, wobei Multiplikation, Division und Restbestimmung Vorrang vor Addition und Subtraktion haben. Eine Gruppenbildung durch Einschließen in Klammern ist möglich. Der `EXPR`-Befehl kann überall dort eingesetzt werden, wo eine Zahl oder ein String stehen darf.

Hinweis:

Folgt dem `EXPR`-Befehl innerhalb einer `TEST`-Anweisung direkt ein Vergleichsoperator (`-EQ` etc.), kommt es zu einem Syntaxfehler: das Minuszeichen wird fälschlich als Rechenoperator interpretiert. Zur Abhilfe kann man den Vergleichsoperator an den Anfang der nächsten Zeile setzen. Ein Zeilenende oder Semikolon beendet die Textanalyse des `EXPR`. Natürlich hilft auch die vorherige Zuweisung des `EXPR`-Ergebnisses in eine Variable, die dann im `TEST` verwendet wird.

Beispiele:

```
ERGEBNIS=EXPR (3+4)*-5        : ERGEBNIS ist ´-35´
COUNT=0;
WHILE TEST $COUNT -LT 10      : solange COUNT kleiner 10
DO
 COUNT=EXPR $COUNT+1          : increment COUNT
...
DONE
```

3.5.9.3 Die SEG-Anweisung

Syntax: `SEG [ [`*Begin*`] , [`*End*`] ] (`*String*`) ( ; | CR)`

Mit Hilfe des `SEG`-Befehls ist eine Bildung von Teilstrings möglich. Durch *Begin* wird der Anfang und durch *End* das Ende des Teilstrings im *String* festgelegt. *Begin* und *End* müssen Zahlen sein oder Zahlen ergeben. Wenn *Begin* nicht angegeben oder negativ ist, dann beginnt der Teilstring am Anfang des Strings. Fehlt die Angabe von *End*, dann endet er mit dem Ende des Strings. Ist *End* minus *Begin* kleiner Null oder liegt der gewählte Bereich außerhalb des Strings, hat der Teilstring die Länge Null.

```
VAR=SEG[2,5](abcdefghij)      : VAR ist gleich ´bcde´
VAR=SEG[,7](abcdefghij)       : VAR ist gleich ´abcdefg´
VAR=SEG[4,](abcdefghij)       : VAR ist gleich ´defghij´
VAR=SEG[12,](abcdefghij)      : VAR ist leer ´´

VAR=SEG[3,EXPR LEN($VAR)-2](abcdefghij)
: LEN($VAR) ist 10; 10–2 ist 8; VAR ist gleich ´cdefgh´
```

3.5.9.4 Die SET-Anweisung

Syntax: `SET [ `*String*` ] ( ; | CR)`

Fehlt der Parameter *String*, wird eine Liste der Variablen auf dem aktuellen Ausgabegerät (mit dem „`O`“-Kommando umlenkbar, siehe Seite 179) ausgegeben.

Ist ein String als Parameter vorhanden, wird er expandiert und dann wortweise den Variablen `$1` ... `$`*n* zugewiesen. Wortgrenzen sind dabei das Leerzeichen sowie Steuerzeichen mit dem ASCII-Code kleiner `29`. `$#` enthält die Anzahl der gebildeten Variablen *n*. Aus vorangegangenen Operationen gebildete Variablen größer `$`*n* bleiben vom `SET`-Befehl unbeeinflußt.

Beispiele:

```
SET                      : Ausgabe der Liste der Variablen
SET "para1 para2"        : $1 = para1; $2 = para2; $# = 2
SET `pwd`                : die Ausgabe von pwd wird wortweise
                         : $1 ... zugewiesen
```

3.5.9.5 Die TOCHAR-Anweisung

Syntax: `TOCHAR(`*arithmetischer Ausdruck*`)` (`;` | `CR`)

Die sich durch den arithmetischen Ausdruck ergebende Zahl wird in das zugehörige ASCII-Zeichen gewandelt. Das Ergebnis ist ein String (aus einem Zeichen) und kann überall eingesetzt werden, wo ein String stehen darf.

```
ECHO TOCHAR(66)          : der Text 'B' wird ausgegeben

VAR=TOCHAR(EXPR(66+1))   : entspricht VAR=C
```

3.5.9.6 Die TOFIX-Anweisung

Syntax: `TOFIX(`*Einzelzeichen*`)` (`;` | `CR`)

Diese Funktion wandelt ein Einzelzeichen in eine Zahl um. Das Ergebnis ist ein String und kann überall dort stehen, wo eine Zahl oder ein String zugelassen ist.

```
ECHO TOFIX(A)         : der Text '65' wird ausgegeben

VX=EXPR(TOFIX(A)+1)   : ergibt VX=66
```

3.5.10 Verschiedene Anweisungen

3.5.10.1 Die BREAK-Anweisung

Syntax: `BREAK [n]` (`;` | `CR`)

Durch das `BREAK`-Kommando kann eine `FOR`-, `WHILE`- oder `UNTIL`-Schleife verlassen werden. Ist der Parameter `n` angegeben, dann wird die Schleife beim `n`-ten Durchlauf abgebrochen, fehlt `n`, erfolgt der Abbruch unmittelbar. Es wird immer die innerste Schleife unterbrochen.

Beispiel:

```
DATE =´0´
UNTIL ER /NIL; DATESET $DATE  : bis das Datum richtig gesetzt ist
DO                            : führe aus
 BREAK 5                      : maximal 5 Versuche
 IF TEST $DATE !=´0´; THEN
  ECHO "Die Eingabe war falsch!"
 FI
 ECHO -N ´Geben Sie das Datum in der Form TT-MM-JJJJ ein :´
 READ DATE                    : lies DATE vom Terminal
DONE                          : Ende der UNTIL-Schleife
```

3.5.10.2 Die CONT-Anweisung

Syntax: `CONT [n] ( ; | CR)`

Mit Hilfe des `CONT`-Kommandos kann an den Anfang einer `FOR`-, `WHILE`- oder `UNTIL`-Schleife gesprungen werden. Ist der Parameter `n` angegeben, dann wird der Schleifenzähler auf `n` gesetzt; fehlt `n`, erfolgt der Sprung an den Anfang der Schleife. Es ist immer nur die innerste Schleife betroffen. Der Schleifenzähler wird bei der `WHILE`- und `UNTIL`-Schleife nur vom `BREAK`-Kommando ausgewertet. Bei `FOR`-Schleifen mit Positionsparametern bestimmt er den nächsten Parameter; bei der `FOR`-Schleife mit „`IN`“ und „*String*“ bestimmt er die nächste Position im String.

Beispiel:

```
FOR LINE IN -L `S`      : lies S zeilenweise ein
DO                      : führe aus
 IF TEST -Z $MARK       : wenn MARK  nicht gesetzt ist
 THEN                   :
  MARK = ´SET´;         : MARK wird gesetzt
  CONT 15;              : beginne mit der 15. Zeile
 FI                     : Ende wenn MARK nicht gesetzt ist
 ECHO $LINE             : gibt S ab der 15. Zeile aus
DONE                    : Ende der FOR-Schleife
```

3.5.10.3 Die EXEC-Anweisung

Syntax: `EXEC` *String* (`;` | `CR`)

Der Parameter *String* wird als Anweisung ausgeführt. Dabei wird der String zunächst expandiert und anschließend in einem zweiten Durchlauf interpretiert. Sind nach der Expansion noch Metazeichen enthalten, so werden sie vom Interpreter auch als solche behandelt! Erlaubt sind Bedienbefehle und die Anweisungen `ECHO`, `EXIT`, `SHIFT`, `SLEEP`, `SET`, `UNSET` sowie alle Zeichenkettenbefehle von `LEN` bis `TOFIX` der Seiten 85 ... 88. Beim ersten Fehler bricht das `EXEC` Kommando ab und gibt als Ergebnis „falsch“ zurück.

Beispiel:

```
kommando ="DIR /H0/PFAD;"
IF EXEC $kommando
THEN ECHO "ok";              : DIR /H0/PFAD Kommando fehlerfrei
ELSE ECHO "fehler"; FI;      : DIR /H0/PFAD fehlerhaft
```

3.5.10.4 Die EXIT-Anweisung

Syntax: `EXIT [n]` (; | CR)

Der `EXIT`-Befehl bewirkt ein sofortiges Beenden der Shell-Prozedur. Der Parameter `n` ist der Fehlercode des Skriptes:

```
EXIT (-1)          : fehlerfrei mit Endemeldung
EXIT (0)           : fehlerfrei ohne Endemeldung
EXIT (1)           : fehlerhaft
```

3.5.10.5 Die SHIFT-Anweisung

Syntax: `SHIFT` (`;` | `CR`)

Bei Anwendung des `SHIFT`-Befehls werden die Positionsparameter wie folgt umbenannt: `$`*n*`--> $`*n*`-1`, wobei `$1` verlorengeht. `$#` wird dabei dekrementiert.

Hinweis:

`$`*n* bleibt nach dem `SHIFT` erhalten! Die letzte gültige Variable erhält man nur über `$#`.

Beispiel:

```
WHILE TEST $# -GT 0      : solange Positionsparameter vorhanden sind
DO
 ECHO $1                 : gibt $1 aus
 SHIFT                   : $2 -> $1 ... ; $# = $#-1
DONE
```

3.5.10.6 Die SLEEP-Anweisung

Syntax: `SLEEP` *n* (`;` | `CR`)

Die Fortführung der Shell-Prozedur wird für *n* Sekunden unterbrochen. Die Zahl `n` muß eine Integerzahl zwischen 1 und 32767 sein.

3.5.10.7 Die SUSP-Anweisung

Syntax: `SUSP` (`;` | `CR`)

Durch den `SUSP`-Befehl wird die Interpreter-Subtask suspendiert. Insbesondere beim Aufruf eines Bediener-Skriptes über den Befehl „`SHELL`“ kann mit dem `SUSP`-Befehl auf das „`CTRL A`“ von der Bedienerkonsole gewartet werden.

3.5.10.8 Die UNSET-Anweisung

Syntax: `UNSET [$` *Variablenname*] (`;` | `CR`)

Mit Hilfe der `UNSET`-Anweisung können *Variablen* aus der Verwaltung des Shellsprachinterpreters entfernt werden. Der dafür vorher belegte Speicherplatz steht damit dem System (und damit auch dem Interpreter) wieder für andere Zwecke zur Verfügung.

Beispiel: `UNSET $VAR1 $HILFE` : Löschen der Variablen `$VAR1`und `$HILFE`

BREAK	: Abbruch einer Schleife
CASE	: Mehrfach-Verzweigung
CONT	: Sprung an den Anfang einer Schleife
DO	: Beginn des Anweisungsteils einer Schleife
DONE	: Ende des Anweisungsteils einer Schleife
ECHO	: Textausgabe
ELSE	: Alternativzweig einer IF-Anweisung
ELIF	: Abkürzung für ELSE IF
ESAC	: Ende der CASE-Anweisung
EXEC	: Ausführung eines Kommandos
EXIT	: Abbruch des Shell-Skriptes
EXPR	: Berechnung eines arithmetischen Ausdrucks
FALSE	: liefert den logischen Wert „falsch“
FOR	: Anfang der FOR-Schleife
FI	: Ende der IF-Anweisung
IF	: Beginn der IF-Anweisung
IN	: Beginn des Strings in einer FOR-Schleife
IN	: Beginn der CASE-Musterliste
LEN	: Bestimmung der Länge einer Variablen
READ	: Anfordern einer Eingabe
SEG	: Bilden eines Teilstrings
SET	: Wortweise Zuweisung an die Positionsparameter
SHIFT	: Verschieben der Positionsparameter
SLEEP	: Unterbrechung der Shell-Prozedur für bestimmte Zeit
SUSP	: Suspendiert die Interpreter-Subtask
TEST	: Ermittlung einer logischen Bedingung
THEN	: Anweisungsteil einer IF-Anweisung
TOCHAR	: Ausdruck in ASCII-Zeichen wandeln
TOFIX	: Zeichen in Zahl des ASCII-Codes wandeln
TRUE	: liefert den logischen Wert „wahr“
UNSET	: Variable aus Verwaltung eliminieren
UNTIL	: Beginn der UNTIL-Schleife
WHILE	: Beginn der WHILE-Schleife

Tabelle 3.2: Schlüsselworte der Shellsprache

$1 - $n	Positionsparameter der Aufrufzeile
$#	Anzahl der Positionsparameter
$@	alle Positionsparameter durch ein Leerzeichen getrennt
$?	Austrittsstatus des zuletzt ausgeführten Bedienbefehls Kein Fehler -> $?='0'; Fehler -> $?='1'
$$	Name der Interpreter-Subtask
$0	Name der interpretierten ED-Datei
$EOF	Austrittsstatus des letzten READ-Befehls

Tabelle 3.3: Die vorbesetzten Shellvariablen

*	Wildcard; beliebige Anzahl beliebiger Zeichen
?	Wildcard; genau ein beliebiges Zeichen
$	mit darauffolgendem Namen: Variablenwert
`	Ausgabe eines Bedienbefehls wird eingelesen
:	mit Doppelpunkt+Leerzeichen beginnt ein Kommentar
\	bitte in der nächsten Tabelle nachsehen!
”	Einschließen einer expandierten Textkonstanten
´	Einschließen einer Textkonstanten
;	Anweisungsende
CR	Zeilen- oder Anweisungsende

Tabelle 3.4: Metazeichen der Shellsprache

mz	Metazeichen *mz* als normales ASCII-Zeichen
dz	ASCII-Code der Dezimalzahl *dz*
\B	Backspace, Code $08
\F	Formfeed, Code 12=$0C
\N	Newline, Code 10=$0A
\O	End of Text, EOT, Code 4=$04
\R	Carriage Return, CR, Code 13=$0D
\T	Tabulator, Code 9=$09

Tabelle 3.5: Sonderzeichen der Shellsprache

3.6 Tabelle der Bedienbefehle

Bedienbefehle, die nur optional vorhanden sind, werden in Schrägschrift dargestellt.

A oder ’	tasknamelist	Aktivierung von Tasks ggf. mit Angabe der Laufpriorität.
ACTIVATE	tasknamelist	Wie „A“
AFTER	schedule,task	Einplanung zeitverzögert
ALL	schedule,task	Einpanung, zyklisch
AS	paralist	Assemblieren durch Sohnprozeß
ASM	paralist	Zusatzname für „MINI“–Assembler
ASSEM	paralist	Wie „AS“
AT	schedule,task	Einplanung für Zeitpunkt
AUTOSTART	tasknamelist	Task autostartfähig machen, s. „PROM“
BADBLOCK	device/block	Markieren eines ungültigen Blocks
C	tasknamelist	Continue suspended Tasks
CD	pathlist	Working-Directory festlegen
CF	pathlist-list	Platten/Disketten montieren etc.
CLEAR	[devbez.]	Löschen von **RTOS–UH**–CE’s
CLOCK		Uhrzeit + nächste Einplanung ausgeben
CLOCKSET	time-specif.	Rechneruhr stellen
CONTINUE	tasknamelist	Wie „C“
COPY	paralist	Kopieren und/oder mischen von Files
CP	paralist	wie „COPY“
CUD	pathlist	Working-Directory festlegen
CUXD	pathlist-list	Execution-Directories festlegen
CXD	pathlist-list	Execution-Directories festlegen
DATE		Systemdatum ausgeben
DATESET	date	Systemdatum setzen
DD	device	Parameterbytes anzeigen
DEFINE	paralist	temporäre Shelltask erzeugen

DIR	pathlist-list	Directories auflisten
DISABLE	eventcode	Prozeßinterrupt(s) abklemmen
DL	taskname	Aktuelle Zeilennummer ausgeben
DM	adress-parameter	Display Memory
DR	taskname	Display Registers of Task
ECHO	textstring	Ausgabe des Textstrings
ED	paralist	Einloggen Texteditor
ENABLE	eventcode	Prozeßinterrupt(s) scharf machen
ENVSET	paralist	Umgebungsvariable setzen
ER	pathlist	Stderr umlenken
ERASE	pathlist-list	Löschen von Files, s. „RM“
FILES	pathlist-list	Auflisten der aktiven Files
FIND	pathlist-list	File-Index ausgeben
FORM	paralist	Platte/Floppy formatieren
FREE	pathlist	Freien Platz auf dem Medium ausgeben
GO	paralist	Prozeß auf angegebener Adresse starten
HELP	optionlist	Hilfefunktion der Shell
I	pathlist	Stdin umlenken
L	[options]	Alle Tasks mit Zuständen auflisten
LE	[options]	Line-Edit installieren / konfigurieren / entladen
LIBSET	file-list	Library einrichten
LINEDDIT	[options]	Line-Edit installieren / konfigurieren / entladen
LINK	path>newname	Aliasname für Datei anlegen
LNK	paralist	Linken von Modulen in S–Rekords
LOAD	paralist	Linken und Laden von Modulen
LOADX	paralist	LOAD + extended search
LU		Usertasks mit Zuständen auflisten
MKDIR	pathlist	Directory neu einrichten

MSFILES	pathlist	Umschalten auf MS–DOS kompatibles Filehandling
NOTRACE	taskname	Adressen– und Zeilenüberwachung abstellen
O	pathlist	Stdout umlenken
P	paralist	PEARL–Programm kompilieren
PEARL	paralist	Langform von „P“
PER	pathlist	Permanent Stderr umlenken
PI	pathlist	Permanent Stdin umlenken
PO	pathlist	Permanent Stdout umlenken
PREVENT	tasknamelist	Einplanungen löschen
PROM	modulelist	Erzeugung von S–Record von PEARL–Programmen für EPROM–Betrieb
PWD		Working-Directory und Execution-Directory ausgeben
QAS	paralist	Programm schnell assemblieren
QP	paralist	PEARL–Programm schnell kompilieren
RELEASE	sema–adr–list	Semaphore freigeben
RENAME	pathlist>newname	Umbenennen eines Files
RETURN	pathlist–list	Files zurückgeben
REWIND	pathlist–list	Files zurückspulen
RM	pathlist–list	Wie „ERASE“
RMDIR	pathlist–list	Directories löschen
RTOSFILES	pathlist–list	**RTOS–UH** Filemanagement einschalten
S	[options]	Speicherbelegung ausgeben
SB	device	Setze Baudrate, serielles Port
SD	device	Parameterbytes setzen
SHARE	prio	Timesharing
SH	tasknamelist	Taskzustände ausgeben
SHOW	tasknamelist	Langform von `SH`
SM	adr.expr., value	Speicherzelle(n) setzen

SU	tasknamelist	Tasks suspendieren
SUSPEND	tasknamelist	Langform von „SU“
SYNC	device	Synchronisieren des Filesystemes
SYSTEM_ABORT		Warmstart durchführen
SYSTEM_RESET		Kaltstart durchführen
T	tasknamelist	Tasks beendigen
TAPP	paralist	Transferassembler PowerPC
TERMINATE	tasknamelist	Langform von „T“
TOUCH	options,pathlist	File-Erstellungsdatum zeigen/ändern
TRACE	taskname,adr/line	Adreß–/Zeilenüberwachung einschalten
TRIGGER	eventcode	Interrupt simulieren
TYPE	paralist	Auflisten eines Files
UNLOAD	namelist	Tasks/Module entfernen
WAIT		Warten auf Sohnprozeß
WHEN	event, schedule	Task für Interrupt einplanen
WHO		Primäre Shellprozesse auflisten

3.7 Beschreibung der Bedienbefehle

Die Beispiele auf den folgenden Seiten gehen der Einfachheit halber immer von einer Bedienung über die primäre Shell der ersten seriellen Schnittstelle (`#USER1`) des Rechners aus. Bei „embedded“ Mikrokontrollern existiert manchmal nur diese „Console“ als einziger Bedienzugriff. Die „Consolen-Shell“ hat stets die User-ID 1 (die systemintern als 0 abgelegt ist). Bezüglich nicht angegebener Parameter der typischen sohnprozeßgenerierenden Befehle (`COPY, P, AS, ...`) gelten für die Consolen-Shell folgende Default–Parameter:

`/A1`	Stdout, Stdin, Stderr (`/TY`) der primären Shell
`/ED/SI`	Default-Source-Input und ED-Arbeits-File
`/ED/SR`	Default-Output S-Rekords für Compiler, Assembler
`/ED/LB`	Default-Library-File für den Lader bei undef. Symbole
`/ED/SC`	Default-Scratch-File für Assembler

Bei der Bedienung über eine Shell (primär oder sekundär) mit der User-ID n>1 verändern sich die Default-Vorbesetzungen durch einen Anhang unmittelbar an den Filenamen. Dieser Anhang besteht aus einem Zeichen und ist identisch mit der User-ID. Die User-ID zählt `1,2,3, ... ,9,A,B, ... ,Z`. (Systemintern sind allerdings mehr möglich).

Statt `/ED/SI`	heißt der File	`/ED/SI`n, z. B. `/ED/SI5`
Statt `/ED/SR`	heißt der File	`/ED/SR`n, z. B. `/ED/SR9`
Statt `/ED/LB`	heißt der File	`/ED/LB`n, z. B. `/ED/LBC`
Statt `/ED/SC`	heißt der File	`/ED/SC`n, z. B. `/ED/SC3`

wobei n z. B. mit Hilfe des „`L`“- oder „`WHO`“-Befehles (siehe Seite 225) inspiziert werden kann.

! → Man beachte, daß ein primärer Netzshellprozeß nach dem Ausloggen seine User-ID verliert. Beim Neueinloggen kann eine andere User-ID zugeordnet werden. Man ist darum gut beraten, seine File-Situation bezüglich eventueller Defaultfiles vor dem Ausloggen zu ordnen.

A / A C T I V A T E — Activate Task (by Priority)

SYNTAX: `ACTIVATE` *taskname* `[PRIO` *integer3*`]`
'taskname `[PRIO` *integer3*`]`
`A` *taskname* `[PRIO` *integer3*`]`
taskname `[PRIO` *integer3*`]`

Beschreibung: Die Task *taskname* wird mit der Priorität *integer3* aktiviert.

Ist die Task bereits aktiv, so wird der Aktivierungszähler der Task erhöht (die Anzahl der gepufferten Aktivierungen wird erhöht); eventuell bestehende Einplanungen der Task auf Zeitpunkte oder Interrupts werden nicht beeinflußt.

Fehlt der Zusatz `PRIO`, so wird der Defaultwert aus der Task–Definition eingesetzt. Das gleiche gilt, wenn *integer3* gleich Null ist. *integer3* ist eine maximal 3–stellige Ganzzahl.

Die Shell prüft selbst nicht, ob die Task vorhanden ist. Der Aktivierungsbefehl kann also zur Auslösung eines Fehlersignals durch den Kern von **RTOS–UH** (hier `... not loaded`) führen.

Beispiel: `ACTIVATE XYZ PRIO 123;test;'ABCDE`

Die Task `XYZ` wird mit der Priorität `123` aktiviert, während die Tasks `test` und `ABCDE` unter Verwendung ihrer Default–Prioritäten aktiviert werden.

Hinweis: Wenn der Taskname gleich einem Bedienkommado ist, kann die Kurzform (nur Taskname) nicht angewendet werden, da zuerst auf ein gültiges Kommando geprüft wird.

Delayed Activation or Continuation **A F T E R**

Syntax: `AFTER` *duration* `ACTIVATE` *taskname* `[PRIO` *integer3*`]`
`AFTER` *duration* `CONTINUE` *taskname*
`AFTER` *duration* `ALL` *duration* `UNTIL` *clock*
`ACTIVATE` *taskname* `[PRIO` *integer3*`]`
`AFTER` *duration* `ALL` *duration* `DURING` *duration*
`ACTIVATE` *taskname* `[PRIO` *integer3*`]`

Mit diesem Befehl kann man die zeitverzögerte Aktivierung oder Fortsetzung einer Task vorplanen. Bestehende Einplanungen für eine Aktivierung (bei `ACTIVATE`) bzw. zur Fortsetzung (bei `CONTINUE`) werden gelöscht, und die angegebene Einplanung wird eingetragen.

Wird bei `ACTIVATE` keine Priorität angegeben, so wird die taskeigene Default–Priorität eingesetzt. Die aktuelle Priorität einer laufenden Task wird jedoch nicht geändert, sondern erst, wenn die Einplanung zur Aktivierung führt.

duration: *integer5* `HRS` *integer5* `MIN` *integer5*`[.`*integer3*`]` `SEC`
Dabei ist *integer5* eine maximal 5–stellige Ganzzahl und *integer3* ein max. 3–stelliger Dezimalbruch. Bis zu 2 Zeiteinheiten (`HRS`, `MIN`, `SEC`) dürfen fehlen, die Reihenfolge `HRS - MIN - SEC` muß jedoch eingehalten werden.

ALL: siehe `ALL`–Schedule (Seite 102). Eine mit `DURING` angegebene Zeitdauer rechnet ab der ersten Aktivierung. Diese Kombination ist im „DIN–Basic–PEARL“ nicht erlaubt, aber im **RTOS–UH**–PEARL wie hier implementiert.

Hinweis: An Stelle der Schlüsselworte `ACTIVATE` und `CONTINUE` sind auch deren Kurzformen Hochkomma (`'`) bzw. `C` zulässig. Die angegebene Verzögerungszeit rechnet ab dem Eintritt des nächsten Clock–Ticks. Der Abstand der Clockticks beträgt bei heutigen 680*xx*-Implementierungen 1 msec, bei älteren z. T. auch 2, 3 oder 4 msec. Ist bei solchen Systemen die Anzahl der Millisekunden für die Gesamtverzögerung nicht durch den Abstand der Clock–Ticks ohne Rest teilbar, so tritt die Aktivierung bzw. Fortsetzung mit dem ersten Clock–Tick nach Ablauf der Zeitspanne ein.

Beispiele:

```
AFTER 0.25 SEC CONTINUE XYZ

AFTER 10 MIN 5 SEC ALL 1 SEC ACTIVATE XYZ PRIO 70

AFTER 5HRS59MIN22.55SEC ACTIVATE XYZ
```

A L L — ALL-Schedule

Syntax: `ALL` *duration* `ACTIVATE` *taskname* `[PRIO` *integer3*`]`
`ALL` *duration* `UNTIL` *clock* `ACTIVATE ...`
`ALL` *duration* `DURING` *duration* `ACTIVATE ...`

Es wird eine zyklische Einplanung für die angegebene Task definiert und evtl. bestehende zeitliche und ereignisgekoppelte (`WHEN`) Einplanungen zur Aktivierung gelöscht. Die erste Aktivierung erfolgt mit dem nächsten Clock–Tick und wiederholt sich – ggf. bis zur Endzeit – von da an zyklisch.

Wird die Priorität (3–stellige Ganzzahl) nicht angegeben, so wird die taskeigene Priorität genommen.

duration: ist vom Typ: *integer5* `HRS` *integer5* `MIN` *integer5*`.[`*integer3*`] SEC`

dabei ist *integer5* eine maximal 5–stellige Ganzzahl und *integer3* ein max. 3–stelliger Dezimalbruch.

Es dürfen bis zu 2 Zeiteinheiten (`HRS`, `MIN`, `SEC`) weggelassen werden, die Reihenfolge `HRS - MIN - SEC` muß jedoch stets eingehalten werden.

clock: ist vom Typ: *integer2:integer2:integer2*`[.`*integer3*`]`

integer2 ist eine 1 bis 2–stellige Dezimalzahl und *integer3* ein max. 3–stelliger Dezimalbruch.

Hinweis: An Stelle des `ACTIVATE` ist auch die Kurzform mit Hochkomma vor dem Tasknamen zulässig.

Zeitdauer und Uhrzeit werden intern als Vielfache von Millisekunden gerechnet. Je nach Implementierung werden jedoch Clock–Ticks von 1 oder 4 ms als Interruptbasis benutzt. Ist der Zyklus nicht ohne Rest durch diese Basis teilbar, so werden die Zeitintervalle länger oder kürzer, aber im Mittel richtig, realisiert.

Beispiele:

```
ALL 0.02 SEC ACTIVATE XYZ PRIO 30
ALL 13 HRS 2.005 SEC 'XYZ
ALL 2 MIN UNTIL 13:05:55.66 ACTIVATE XYZ
ALL 7000 SEC DURING 14000 SEC ACTIVATE ABC
```

`ABC` wird 3 mal aktiviert – sofort, nach 7000 sec und nach 14000 sec.

Assemble Program **A S / A S S E M**

Syntax: `AS.`*sonprocname* `[PRIO` *integer3*`][`*size--spec*`] [`*parameterlist*`]`
`AS [PRIO` *integer3*`] [`*size--spec*`] [`*parameterlist*`]`
`ASSEM.`*sonprocname* `[PRIO` *integer3*`] ...`
`ASSEM [PRIO` *integer3*`] [`*size--spec*`] [`*parameterlist*`]`
— zusätzlich für den „MINI“–Assembler (nur 68000-Befehle):
`ASM [PRIO` *integer3*`] [`*size--spec*`] [`*parameterlist*`]`

Der Befehl dient zum Übersetzen von Programmen, die in der 680*xx*-Maschinensprache formuliert sind. Die Shell generiert zu diesem Zweck einen eigenständigen Sohnprozeß mit vom Nutzer vorgegebenem Namen oder, falls in der zweiten Form benutzt, einem Systemnamen `AS/`*xx* oder `ASSEM/`*xx*. Für *xx* wird eine zweistellige Hexzahl mit automatischer Weiterschaltung eingesetzt. Die Priorität des Sohnprozesses kann vorgegeben oder dem System überlassen werden (Default: 20).

Ebenso kann der dynamische Arbeitsspeicher des so erzeugten 2–Pass–**RTOS**–**UH**–Assemblers vorgewählt oder die vom System standardmäßig gewählte Größe von 5 kB benutzt werden. Wenn das Feld *size–spec* benutzt wird, so ist dies bis maximal `SZ=10100` (64 kB) sinnvoll.

Da der Assembler wiedereintrittsfest codiert ist, können, solange der Speicherplatz reicht, beliebig viele Programme gleichzeitig assembliert werden, wobei stets derselbe, im ROM gespeicherte Assembler benutzt wird.

parameterlist: Es werden die Elemente `SI` (`S`ource `I`nput), `LO` (`L`ist `O`utput), `CO` (`C`ode `O`utput) und `SC` (`SC`ratch–pad) akzeptiert. Die Reihenfolge ist bedeutungslos, die Liste darf auch leer sein. Für fehlende Angaben werden die „Default–Werte“ des Systems eingesetzt (`SI=/ED/SI`, `LO=/A1/`, `CO=/ED/SR`, `SC=/ED/SC`).

Wird ein Parameter auf den Wert `NO` gesetzt, so gelten dennoch davor oder dahinter gemachte Vereinbarungen bzw. die Defaultwerte, wenn der Assembler das Gerät im Ausnahmefall benötigt.

Das `SC`ratch–pad wird nur benötigt, wenn das Gerät für `SI` nicht rückspulbar ist (z. B. `/VI/`, `/A2/`). Ist das Gerät für `SI` rückspulbar, so erfolgt sowohl bei `SC=NO` als auch bei fehlendem `SC`–Parameter zweimaliges Lesen der `SI`–Datei ohne `SC`ratch–pad Benutzung. Wird beim Aufruf des Assemblers nur eine Datei

angegeben, so wird diese als SI–Datei betrachtet.

Beispiele: `AS.T PRIO 40 SZ 4000 /F1/QUELLE>/A2/ LO NO`

Name des Sohnprozesses ist „T“. Nur fehlerhafte Zeilen mit Fehlerbeschreibung über /A1/ ausgeben — keine Liste. Ausgabe der S–Records über /A2/. Da SI als Floppy–Datei rückspulbar und SC nicht angegeben ist, wird der File QUELLE auf /F1/ zweimal gelesen und kein SCratch–pad benutzt.

`ASSEM`

Aufruf ASSEM/*xx* mit Default–Werten.

`ASSEM.X /A2/>NO LO /A1/ SC /F1/BX`

Name des Sohnprozesses ist „X“. Nur Syntaxprüfung der über /A2/ eingegebenen Programmzeilen, Ausgabe der Liste über /A1/. S–Records werden nicht erzeugt. Der Eingabetext wird auf der Datei BX auf Floppy /F1/ zwischengespeichert (und ist dort später verfügbar).

Hinweis: Soweit rückspulbare Files benutzt werden, erfolgt automatisch zu Beginn eine REWIND– und zum Abschluß eine RETURN–Operation.

Der generierte Sohnprozeß verschwindet nach Abschluß der Assemblierung vollständig aus dem System, es sei denn, daß evtl. Folgebefehle (mit 2 Minuszeichen angehängt) den Sohn zunächst noch in eine sekundäre Shell verwandeln.

Weitere Erläuterungen zum Assembler finden Sie ab Seite 419.

At given time activate or continue **A T**

Syntax: `AT` *clock* `ACTIVATE` *taskname* `[PRIO` *integer3*`]`
`AT` *clock* `CONTINUE` *taskname*
`AT` *clock* `ALL` *duration* `ACTIVATE` *taskname* `[PRIO` *integer3*`]`
`AT` *clock* `ALL` *duration* `UNTIL` *clock* `ACTIVATE ...`
`AT` *clock* `ALL` *duration* `DURING` *duration* `ACTIVATE ...`

Mit diesem Befehl ist die Einplanung zur Aktivierung oder Fortsetzung einer Task zu einem bestimmten Zeitpunkt möglich. Der Befehl funktioniert ansonsten wie eine Einplanung mit `AFTER`, siehe Seite 101; statt der relativen Zeitspanne bei `AFTER` wird bei `AT` ein absoluter Zeitpunkt festgelegt.

clock: *integer2:integer2:integer2*`[.`*integer3*`]`

integer2/3 sind max. 2– bzw. 3–stellige Dezimalzahlen.

ALL: siehe `ALL`. Die Kombination `AT...ALL` ist im „DIN–Basic–PEARL“ nicht erlaubt, jedoch im Compiler wie hier implementiert.

Beispiele:
```
AT 0:10:00 CONTINUE test
```
```
AT 7:00:0 ALL 2 SEC UNTIL 9:0:0 ACTIVATE XYZ PRIO 8
```

Hinweis: Ist die angegebene Uhrzeit kleiner als die Istzeit, so wird der Wert von clock um 24 Stunden inkrementiert.

B A D B L O C K — Badblock setting

SYNTAX: `BADBLOCK /dev/Bnnnn`

Wenn auf einer Diskette oder Festplatte ein einzelner Sektor unbrauchbar geworden ist, so muß entweder der ganze Datenträger neu formatiert werden (Verlust aller darauf gespeicherten Daten) oder dem zuständigen Filehandler muß mitgeteilt werden, daß er in Zukunft diesen defekten Sektor nicht mehr benutzt. Dazu ist dieser Befehl geeignet. Weil jeder Sektor Teil eines sogenannten „Blockes“ ist, der eine logische Verwaltungsnummer trägt, kann stets nur der komplette Block aus der Systemverwaltung herausgenommen werden. Die Blocknummer erhält man aus den Fehlermeldungen des Filehandlers.

Beispiel: `BADBLOCK /H0/B20`

Es wird der Block mit der Nummer 20 aus der Verwaltung der Festplatte `H0` entfernt.

Anwendung: Sobald der Filemanager einen Block mit der Meldung „`ID-Field not found`“ oder „`CRC-Error ...`“ meldet, sollte daran gedacht werden, mit Hilfe des `BADBLOCK`–Kommandos den angezeigten Block aus der Verwaltung herauszunehmen. Weil der gesamte File, in dem der defekte Block steht, zunächst gelöscht werden muß, sollte man immer erst mehrmals versuchen, ob sich die Datei nicht doch noch (zumindest bis zur defekten Stelle) retten läßt.

Angenommen, der Filemanger gab die folgende Meldung aus:

`>>`*drivername*`../dev/path: ID-Field not found in Block 20`

Aktion des Nutzers:

`RM /dev/path`
`RM /dev/path` (muß explizit ein zweites Mal eingegeben werden)
`BADBLOCK /dev/B20`

Change Directory **C D**

Syntax: `CD` *devpath*

Der Befehl `CD` erlaubt es, für nachfolgende Bedienbefehle der ausführenden Shell ein „Working-Directory“ zu vereinbaren oder die bisherige Vereinbarung zu ändern. Das Working-Directory wird bei allen Befehlen, die mit Devices und Files arbeiten, immer dann eingesetzt, wenn der Dev-File-Bezeichner nicht auf der Root–Ebene, d. h. nicht mit dem Zeichen „`/`“ beginnt. Das aktuell gültige Working-Directory kann mit dem Befehl „`PWD`“ abgefragt werden (Seite 189).

Jeder Nutzer kann sich sein individuelles Working-Directory einrichten und wird damit bei der Verwaltung hierachisch organisierter Dateien unterstützt. Der `CD`-Befehl wirkt allerdings nur auf die Umgebung der ausführenden Shell. Bei primären Shellprozessen ist das das User-Environment.

Normale sekundäre Shellprozesse verändern mit `CD` nur ihre eigene nach außen abgeschlossene Umgebung. `CD` ist darum nicht geeignet, um in einem Auto-Exec-File, der nach `XCMMD` kopiert wird, das User-Environment zu verändern. Zu diesem Zweck ist der Bedienbefehl `CUD` vorgesehen.

Mit dem `SHELL`-Befehl erzeugte sekundäre (Bourne-) Shells benutzen jedoch das gleiche Execution- und Working-Directory wie die primäre Shell, von der sie abstammen. Hier wirkt `CD` dann genau wie ein `CUD`.

devpath: Bezeichnet einen Filezugriffspfad im System, der wie ein Directory nach rechts verlängerbar ist, z. B.

`/ED`, `/F0`, `/H0/Maier/simul` oder `/A1`.

Beginnt *devpath* nicht auf der Root-Ebene, so wird ihm das zu dem Zeitpunkt vereinbarte Working-Directory vorangestellt.

Backpath: Man kann sich mit Hilfe des Befehles

```
CD ..
```

im aktuellen Working-Directory um einen Pfadabzweig rückwärts bewegen, wie im folgenden dargestellt:

```
WD=/H0/TEST/UGRUP        -> CD .. ->        WD=/H0/TEST
```

Entsprechend kann mit

```
CD ../..
```

usw. gleich um mehrere Abzweige zurückgegangen werden.

Löschen: `CD NO` löscht das Working-Directory. Ein gelöschtes Working-Directory erscheint mit dem Text `WD=/-`

Beispiel 1: `CD /ED/` — Die Shell antwortet:
```
WD=/ED/-
XD=/-
```
Damit ist als Working-Directory /ED/ vereinbart
Ein `COPY`-Befehl könnte jetzt so aussehen:

```
COPY /F0/TEST>TEST1
```

Beispiel 2a: `CD /H0/PROG` — Die Shell antwortet:
```
WD=/H0/PROG
XD=/-
```
Zur Compilation könnte man schreiben:

`P MESS>/ED/TESTSR LO NO` — Das Programm `/H0/PROG/MESS` wird übersetzt.

Beispiel 2b: `CD MIST` — Die Shell antwortet:
```
WD=/H0/PROG/MIST
XD=/-
```

! → Man beachte vorsorglich, daß man keine Working-Directories vereinbart, die später die Restriktionen bestimmter File-Handler verletzen. Alle bekannten File-Handler beherrschen jedoch mindestens jeweils 7 Zeichen zwischen den Pfadtrennern „/“.

! → Die maximale Länge des Working-Directorys – der eröffnende „/“ und der Gerätename samt folgendem „/“ zählen dabei nicht! – ist implementierungsabhängig. Defaultimplementierungswert sind 64 Zeichen. Bei Verletzung der Obergrenze reagiert die Shell mit „`... path too long`“ und Abbruch der Kommandozeile.

Change Filesystemstate **C F**

SYNTAX: `CF /discdevice/extrainfo`
`CF /discdevice/`

Beschreibung: Es wird dem System mitgeteilt, daß sich der Filezustand der angegebenen Diskette bzw. der Wechsel- oder Festplatte in irgendeiner Weise ändern wird. Der Befehl ist auch geeignet, um sich zu vergewissern, daß keine Teile des Filesystems mehr im Speicher gehalten werden, man also das Laufwerk ausschalten oder die Diskette entnehmen darf.

extrainfo: Hier gibt es verschiedene Textstrings, die vom System akzeptiert werden. Wir nehmen als beispielhaftes Discdevice einmal `/F0` an:

CF /F0	Überprüfung, ob das Filesystem inaktiv ist. Falls nicht, wird eine Fehlermeldung ausgegeben. Das Filesystem bleibt jedoch aktiv.
CF /F0/FORGET	Filesystem abwerfen ohne Abgleich mit den Daten auf der Disc. Waren Daten nicht zurückgeschrieben, d. h. Files offen, so sind sie nun (außer bei vorhergehendem „`SYNC`“) nicht auf der Disc gesichert, also Vorsicht!
CF /F0/MOUNT	Das Directory wird geöffnet, um in Zukunft mit höherer Geschwindigkeit arbeiten zu können. Solange keine Files offen sind, sind die Daten auf dem Medium jedoch stets mit dem Speicherinhalt im Einklang. (Lese-Cache für das Directory mit write-thru-Betrieb)
CF /F0/RECALL	Falls das System durch einen Klappeninterrupt beim Wechsel der Diskette oder Platte alarmiert wurde, so kann nun das Wiedereinlegen angezeigt werden. Befehl muß aus Sicherheitsgründen zweimal eingegeben werden.
CF /F0/UMNT	Beendigung des montierten Zustandes.

CF /F0/Vx — x=0,1,2 oder 3. Wenn eine Floppy direkt mit einem FD-Controller gesteuert wird, so kann die Steprate des Kontrollers (0=schnellste, 3=langsamste) verändert werden. Gilt dann für alle Laufwerke an diesem Kontroller.

Bei Systemen mit Klappenabfragemöglichkeit kann entschieden werden, was nach irrtümlich entnommener Diskette geschehen soll. Beim Wiedereinlegen (`RECALL`) wird die noch im Speicher vorhandene Verwaltung weiterbenutzt, wird eine falsche Diskette eingelegt, so wird diese zerstört. `RECALL` muß nach Einlegen zweimal gegeben werden.

```
CF /F1/RECALL;DIR /F1/;CF /F1/RECALL
```

Hinweise: Man sollte sich für das Entnehmen der Disketten die Benutzung des `CF`–Befehles zur Regel machen. Die Systemwarnung soll dann zum Retten noch geöffneter Files animieren. Der `FORGET`–Mode ist als Softwarereset auch ohne Klappenabfrage sinnvoll einsetzbar — mit entsprechender Vorsicht! — z. B. nach vorherigem „`SYNC`“. Man beachte, daß die als Pfadname kodierten Kommandos stets mit Großbuchstaben geschrieben werden müssen.

Fehler: Kein Device angegeben, oder Device ist keine Disc, oder der momentane Zustand läst die Operation nicht zu.

`>>> ... :F`*x* `directory active` (Files noch offen).

Beispiele:

```
CF /F1/ ; cf /f0/ ; CF /F0/V2
```

`CF /F1/FORGET;` (Vorsicht!!)

```
CF /H0/MOUNT;
```

`CF /F0/RECALL;` (Nur bei Systemen mit Klappentest).

Clear Device (optional) **C L E A R**

SYNTAX: `CLEAR /device/`

Beschreibung: Mit diesem Kommando wird auf den Treiber einer Warteschlange ein „Continue“ abgesetzt, sodaß liegen gebliebene CE's vom Typ `RTOS` durch den Treiber entfernt werden können. Dabei sind nur besondere kundenspezifische Treiber zugelassen.

Beispiel: Falls eine Task, die ein „`GET`“ auf eine serielle Schnittstelle abgesetzt hat, von der Seite terminiert wird, so ist die Schnittstelle durch das Eingabe–CE solange blockiert, bis ein entsprechendes Endezeichen oder die angeforderte Anzahl der Zeichen erreicht ist. Mit Hilfe des `CLEAR` Befehls kann der I/O–Treiber veranlaßt werden, das CE sofort aus seiner Verwaltung zu entfernen und ist somit bereit für neue Aufträge.

```
CLEAR /B2/
```

Bemerkung: Dieses Kommando ist systemfeindlich und wird nur von wenigen I/O–Dämonen richtig bearbeitet. Insbesondere wirkt es nicht auf Datenstationen mit dem Attribut „formatierbar“, da diese offene Verwaltungsstrukturen im Speicher halten.

Hinweis: Falls auf das angegebene Device noch von einer Task CE's produziert werden, so kommt es zum Konflikt zwischen der Task und dem `CLEAR`–Befehl, wobei die Shell längere Zeit blockiert sein kann. Den `CLEAR`–Befehl kann man dann durch den „Notruf“ der Shell mit `BREAK` abbrechen.

C L O C K — Inspect computer–clock

SYNTAX: `CLOCK`

Beschreibung: Es wird der aktuelle Stand der Rechneruhr ausgegeben. Außerdem wird die Uhrzeit des zeitlich nächstfolgenden Einplanungszeitpunktes hinzugefügt.

Selbst wenn im System überhaupt keine zeitlichen Einplanungen vereinbart wurden, wird ein nächster Einplanungstermin angegeben. Dies ist die „Geisterstunde“, in der die zentrale Rückstellung der Uhr und aller Planungszeitpunkte um 24 Stunden erfolgt.

Beispiele: `CLOCK` Ausgabe: `05:24:59 NEXT SCHED 10:55:30`

`CLOCK` Ausgabe: `22:01:12 NEXT SCHED 24:00:00`

Die Ausgabe der Geisterstunde im letzten Beispiel zeigt an, daß für den laufenden Tag keine Zeiteinplanungen zur Aktivierung oder Fortsetzung mehr vorliegen.

Hinweis: Zeigt die Rechneruhr eine deutliche Tendenz zum Nachgehen, die nicht auf Ungenauigkeiten des Quarzoszillators beruhen können, so ist das Betriebssystem überlastet.

Eventuell kann der Austausch des Betriebssystems gegen eines mit größerem Abstand der Clock–Ticks oder der Austausch des Prozessors gegen eine schnellere Version erforderlich sein.

Eine Überlastung, die sich auf die Ganggenauigkeit der Uhr auswirkt, kann normalerweise nicht durch reguläre Nutzerprogramme, sondern nur durch eine zu große Zahl oder zu zeitaufwendige Interruptprozesse Ihrer Implementierung verursacht werden.

Set Computer–clock to given time **C L O C K S E T**

SYNTAX: `CLOCKSET` *clock*

Beschreibung: Die Rechneruhr wird auf die angegebene Uhrzeit gestellt. Es empfiehlt sich, vorher alle zeitlichen Einplanungen zu löschen, da bei Vorrücken der Uhrzeit u. U. eine große Zahl von Aktivierungen bzw. Fortsetzungen sofort und gleichzeitig fällig werden können.

clock: *integer2:integer2:integer2*`[.`*integer3*`]`

integer2/3 sind max. 2– bzw. 3–stelige Ganzzahlen.

Beispiel:
```
CLOCKSET 13:00:00
```
```
CLOCKSET 0:0:10.5
```

Hinweis: Es wird nur die Software–Uhr Ihres Rechners gestellt. Das Stellen der Hardware–Uhr kann implementationsabhängig auch erfolgen.

C / CONTINUE — Continue Task

SYNTAX: `CONTINUE` *taskname*
`C` *taskname*

Beschreibung: Die angegebene Task wird aus dem Zustand „suspended“ in den Laufzustand gebracht.

! → Die Shell prüft zwar, ob die Task vorhanden ist, jedoch nicht, ob sie – wie es sein sollte – im suspendierten Zustand ist. Der Auftrag wird nach der Identifikation der Task an das Betriebssystem abgesetzt. Von dort erfolgt ggf. die Auslösung des Fehlersignals `... not suspended`. Solch ein Fehler ist an sich harmlos, er führt aber dazu, daß die Shell – wie üblich – den Rest der Befehlszeile nicht mehr bearbeitet.

Beispiel:
```
CONTINUE ABCD; C XYZ; CONTINUE E
```

Kopieren **C P / C P B / C O P Y**

SYNTAX: `COPY.`*sonprocname* `[PRIO` *integer3*`] [`*size-spec*`]`*paramlist*
`COPY [PRIO` *integer3*`] [`*size--spec*`]` *paramlist*

Kurzform: `CP ...` statt `COPY ... CPB ...` statt `COPY ...`
Binärmode:

Der Befehl dient zum Kopieren und Mischen von Dateien. Die Shell generiert dazu einen eigenständigen Sohnprozeß (Task) mit vom Nutzer vorgegebenem Namen oder einem vom System erzeugten Namen `COPY/`*xx* oder `CP/`*xx*. Für *xx* wird eine zweistellige Hexzahl mit automatischer Weiterschaltung eingesetzt.
Die Priorität des Sohnprozesses kann vorgegeben oder dem System überlassen werden — Defaultwert ist 20.
Da der im Betriebssystem liegende Programmcode wiedereintrittsfest ist, können — solange der Speicher für den ca. 200 Byte großen Task-Kopf reicht — beliebig viele unterschiedliche `COPY`-Befehle abgesetzt werden, die im Multitasking parallel bearbeitet werden.
Sind sowohl Ein– wie Ausgabedatei formatierbare Geräte (siehe dazu `SD`-Befehl auf Seite 203), so findet ein binärer Transfer der Daten statt, d. h. der Kopiervorgang ist erst beendet, wenn das Ende der Eingabedatei erreicht ist. Es wird in diesem Fall nicht auf ein EOT ($04) reagiert. Daneben kann durch Verwendung des Befehles `CPB` der binäre Transfermodus erzwungen werden – sinnvoll z.B. wenn im `/ED`-Filesystem binäre Dateien abgelegt werden sollen. Allerdings wird `CPB` nur bei rückspulbaren Quellfiles akzeptiert, weil sonst (z.B. bei serieller Schnittstelle) kein Ende des Datenstromes erkannt würde.

paramlist: Es gelten die Parameter

`SI` (Source-Input)

`CO` (Copy/Corrected Output)

`SC` (Source Correction/Command)

Fehlen `SI` oder `CO`, so werden die Defaultwerte des Systems eingesetzt (`SI=/A1/`, `CO=/ED/SI`), was normalerweise nicht sinnvoll sein dürfte.

Bei der Angabe des `CO`-Parameters kann der Dateiname weggelassen werden, wenn der Name vom `SI` übernommen werden soll. Die Pathliste muß aber angegeben werden.

Es gibt — abhängig davon, ob `SC` angegeben wurde oder nicht — zwei verschiedene Betriebsfälle:

1. `SC` fehlt oder `SC=NO`

Der bei SI angegebene File wird vollständig auf den bei CO angegebenen File übertragen. Es werden max. 128 Zeichen zu je einem „Record“ zusammengefaßt und ggf. — je nach Device-Parametersatz der CO–Datei gemäß Seite 203 — nach Ergänzung eines Zeichens LF auf die Ausgabedatei übertragen. Als Ende eines „Record“ auf der SI–Seite gelten auch die Zeichen LF, CR und EOT. Die Übertragung wird beendet, wenn ein „Record“ nur aus dem Zeichen EOT besteht oder das Gerät der SI–Datei eine End–of–file–Bedingung festgestellt hat. Bei Übertragungsfehlern wird der Sohnprozeß vorzeitig beendet und verschwindet nach Ausgabe entsprechender Meldung über das Terminal. Auch die normale Beendigung wird durch *...name* `(terminate)` angezeigt.

2. SC wurde einer Eingabedatei zugeordnet.

   Es werden die Zeilen aus den Quellen SI und SC gemischt und als „Records“ auf die CO–Datei übertragen. Die Anweisungen zum Mischen und — bei rückspulbarer SI–Datei — auch zum Umschichten der aus SI stammenden Zeilen werden als Kommandos ebenfalls über das SC–Gerät eingegeben. Als Unterscheidungsmerkmal zwischen Kommandos und einzumischendem Text auf dem SC–Gerät dient das Zeichen am Anfang einer Zeile. Steht dort das Zeichen +, so wird die Zeile als Kommando interpretiert. Mit + beginnende Zeilen können also nicht eingemischt werden.

   `+33-455` Übertrage die Zeilen Nr. 33 bis einschließlich 455 aus dem SI–File in den CO–File. Anschliessend können beliebig viele Zeilen (die nicht mit + beginnen) eingegeben werden, die direkt nach CO übertragen werden.

   `+755` Übertrage nur die Zeile Nr. 755 aus dem SI–File in den CO–File.

   `+855-840` Unzulässig, die zweite Zeilennummer darf niemals kleiner als die erste sein. Es wird die „Kommando-Fehler“–Kondition (s. u.) angenommen.

   `+399-402` Die Rückkehr vor oder auf die letzte bereits aus dem SI–File übertragene Zeile ist nur bei rückspulbarem SI–File erlaubt, sonst wird die „Kommando-Fehler“–Kondition angenommen (s. u.). Nach dem Rückspulen wird die SI–Datei durch Lesen von An-

fang an auf die angegebene Stelle positioniert und die Übertragung wie gewünscht durchgeführt.

Beispiel: `COPY.Z PRIO 16 /F0/ALT>/F1/NEU`

Name des Sohnprozesses ist `Z`. Mit Priorität 16 wird der File `ALT` von Floppy 0 in den File `NEU` der Floppy 1 kopiert, bis der File `ALT` an sein Ende oder eine mit EOT beginnende Zeile gekommen ist.

`CP /F1/Quelle>/F0/Ziel SC /A1/`

Name des Sohnprozesses und Priorität werden vom System gewählt. Es werden Kommandos zum Mischen über das Gerät `/A1/` erwartet.

Annahme SI-Inhalt:	AAAAAAAAA	(1. Zeile)
	BBBBB	(2. Zeile)
	CCC	(3. Zeile)

Angenommene SC-Zeilen:	+2-3
	KKKK
	+1 (SI rückspulbar)
	JJJJ
	+2
	EOT

Ergebnisfile:	BBBBB
	CCC
	KKKK
	AAAAAAAAA
	JJJJ
	BBBBB
	EOT

Der Mischvorgang wird beendet, sobald entweder vom `SI`– oder vom `SC`–File eine Endebedingung (EOT oder End–of–file) erkannt wird, also auch dann, wenn eine Zeile hinter der letzten auf `SI` vorhandenen Zeile adressiert wird.

Fehler: Wird ein falsches Kommando erkannt (Adressierung einer Zeile hinter der letzten vorhandenen ist kein Fehler), so wird die Meldung `... wrong command` ausgegeben und die Task suspendiert. Nach der Fortsetzung wird das Kommando erneut erwartet.

```
COPY /H0/mueller/dat1>/H1/meier/
```

Die Datei `dat1` wird von `/H0/mueller` nach `/H1/meier/` kopiert. Der Name der Datei bleibt gleich.

! → `COPY` *filename* `>` oder
`COPY` *filename* `>.`
bedeutet „kopiere in das aktuelle Working-Directory mit altem Filenamen“ – sofern ein Working-Directory definiert ist (sonst Fehler).

! → Files mit dem Device-Parameter „rückspulbar“ (siehe Seite 203) werden automatisch mit `REWIND` eröffnet bzw. neu eingerichtet und zum Abschluß mit `RETURN` zurückgegeben. Wenn eine Datei auf sich selbst kopiert werden soll, bricht der `COPY`–Befehl mit „`wrong command`“ ab.

Change User-Environment-Directory

C U D

SYNTAX: `CUD` *devpath*

Dieser Befehl ist eine Sonderform des `CD`-Befehles. Es gelten alle Angaben der Seiten 107 ff. Die Besonderheit besteht darin, daß mit `CUD` nicht das momentane lokale Working-Directory neu eingestellt wird, sondern dasjenige, welches zur primären Shell des ausführenden Nutzers gehört.

Nur mit `CUD` kann man aus Shellskripten heraus das Working-Directory der primären „Ur“-Shell verändern. Die lokalen Vereinbarungen des Skriptes bleiben unberührt. Logischerweise macht der Befehl keinen besonderen Sinn, wenn er von einer primären Shell aufgerufen wird: die Wirkung von `CD` und `CUD` ist dann völlig identisch. Gleiches gilt auch für die speziellen sekundären Shells, die durch den `SHELL`-Befehl (siehe Seite 206) von einer primären Shell erzeugt wurden, da sie fortan die erzeugende primären Shell vertreten.

Beispiel: `CUD /ED`

Im Gegensatz zum `CD`-Befehl antwortet die Shell beim `CUD` nicht mit der Ausgabe der aktuellen Einstellungen.

C U X D — Change User-Environment Execution-Directory

SYNTAX: `CUXD` *devpath1[,] devpath2[,] ...*

Dieser Befehl ist eine Sonderform des `CXD`-Befehles. Es gelten alle Angaben der Seiten 121 ff. Die Besonderheit besteht darin, daß mit `CUXD` nicht die momentanen lokalen Execution-Directories neu eingestellt werden, sondern jene, die zur primären Shell des ausführenden Nutzers gehören.

Nur mit `CUXD` kann man aus Shellskripten heraus die Execution-Directories der primären „Ur“-Shell verändern. Die lokalen Vereinbarungen des Skriptes bleiben unberührt. Logischerweise macht der Befehl keinen besonderen Sinn, wenn er von einer primären Shell aufgerufen wird: die Wirkung von `CXD` und `CUXD` ist dann völlig identisch. Gleiches gilt auch für die speziellen sekundären Shells, die durch den `SHELL`-Befehl (siehe Seite 206) von einer primären Shell erzeugt wurden, da sie fortan die erzeugende primären Shell vertreten.

Beispiel: `CUXD /ED /H1/XD`

Im Gegensatz zum `CXD`-Befehl antwortet die Shell beim `CUXD` nicht mit der Ausgabe der aktuellen Einstellungen.

Change Execution-Directory **C X D**

SYNTAX: `CXD` *devpath1[,] devpath2[,] ...*

Der Befehl Change–Execution–Directory erlaubt es, die von der aktuellen Shell nach transienten Kommandos und Skripten in Shellsprache (siehe Seite 76) zu durchsuchenden Directories zu definieren. Mit jedem `CXD`-Befehl werden, mit dem ersten beginnend, alle Execution-Directories neu festgelegt, für die *devpath*–Angaben vorhanden sind. Weiter hinten folgende Vereinbarungen bleiben bestehen. Die Anzahl der definierbaren Execution-Directories ist implementierungsabhängig, Defaultwert ist 2.

Normale sekundäre Shellprozesse verändern mit `CXD` nur ihre eigene nach außen abgeschlossene Umgebung. `CXD` ist darum nicht geeignet, um in einem Auto-Exec-File, der nach `XCMMD` kopiert wird, das User-Environment zu verändern. Zu diesem Zweck ist der Bedienbefehl `CUXD` vorgesehen.

Mit dem `SHELL-`Befehl erzeugte sekundäre (Bourne-) Shells benutzen jedoch das gleiche Execution- und Working-Directory wie die primäre Shell, von der sie abstammen. Hier wirkt `CXD` dann genau wie ein `CUXD`.

devpath: Bezeichnet einen Filezugriffspfad im System, der wie ein Directory nach rechts verlängerbar ist, z. B.

`/ED, /FO, /HO/Maier/simul` etc.

Beginnt *devpath* nicht auf der Root-Ebene, so wird ihm das zu dem Zeitpunkt vereinbarte **Working**-Directory – und nicht eines der Execution-Directories! – vorangestellt. Der Befehl macht darum fast immer nur mit vollen – auf der Root-Ebene beginnenden – *devpath*-Angaben Sinn.

Löschung: `CXD NO` löscht das erste Execution-Directory. Ein gelöschtes Execution-Directory erscheint mit dem Text `XD=/-`, oder es wird nicht aufgelistet, falls nur noch unbesetzte „`XD`s“ folgen. Sollen mehrere Directories gelöscht werden, so ist die entsprechende Anzahl `NO` oder „`/`“ als Parameter anzugeben.

Beispiel 1: `CXD /ED/` Die Shell antwortet:
```
WD=/xxxx
XD=/ED
```
Damit ist als Exec.-Directory /ED/ vereinbart
Ein transienter Befehl könnte jetzt so aussehen:

`QP` *`paralist`*

Beispiel 2: `CXD /ED /HO/XD` Die Shell antwortet:
```
WD=/xxxx
XD=/ED
+ /HO/XD
```
Nun sind 2 `XD`s aktiv

! → Man beachte vorsorglich, daß man keine Execution-Directories vereinbart, die später die Restriktionen bestimmter File-Handler verletzen. Alle bekannten File-Handler beherrschen jedoch mindestens jeweils 7 Zeichen zwischen den Pfadtrennern „`/`“.

! → Die maximale Länge des Execution-Directories ist implementierungsabhängig. Dabei zählen der eröffnende „`/`“ und der Gerätename samt folgendem „`/`“ nicht mit. Der Kern defaultiert die Obergrenze zunächst auf 24 Zeichen. Die heute gebräuchlichen Entwicklungssysteme sind in der Regel auf 64 Zeichen eingestellt. Bei Verletzung der Obergrenze reagiert die Shell mit „`... path too long`“ und Abbruch der Kommandozeile.

Show Date **D A T E**

SYNTAX: DATE

Beschreibung: Es wird das aktuelle Datum der **RTOS–UH**–Datumszeile ausgegeben. Wurde diese Zelle noch nicht gesetzt (s. Befehl `DATESET`), so wird eine Folge von Minus–Zeichen ausgegeben.

Beispiel: `DATE`
Ausgabe: `01-01-1987`

`DATE`
Ausgabe: `----------`
(Datumszelle war noch nicht gesetzt)

D A T E S E T — Set Computer–Date

SYNTAX: `DATESET` *`date`*

Beschreibung: Die Datumszelle von **RTOS–UH** wird gesetzt. Implementationsabhängig kann auch eine vorhandene Hardware–Uhr gesetzt werden.

date: Eine Zeichenkette mit dem Aufbau `tt-mm-jjjj`

tt:	Tag
mm:	Monat
jjjj:	Jahr (zwischen 1984 und 2162)

Fehler: Es wird geprüft, ob Anzahl Tage/Monat und Anzahl Monate/Jahr zulässig ist. Im Fehlerfall erscheint die Meldung

```
...... : date wrong
```

Beispiel: `DATESET 29-02-1988`
Das Datum wird auf den 29.02.1988 gesetzt.

Display device–parameters **D D**

SYNTAX: `DD /device/`

Beschreibung: Für die angegebene Station wird der Inhalt der aktuellen Parameterbytes ausgegeben. Die Bedeutung der einzelnen Bits entnehmen Sie bitte der Beschreibung des `SD`-Kommandos auf Seite 203. Das Kommando wird intern über die `DM`-Funktion bearbeitet. Dadurch werden mehr Bytes ausgegeben als für die angegebene Station signifikant sind.

Beispiel: `DD /A1/`

Ausgabe Adresse des Parameterfeldes und Inhalt:

xxxxxxxx`: 3300`

Lies: LF nach CR ergänzen (`$20`), dialogfähig (`$10`), Ausgabe möglich (`$02`) und Eingabe möglich (`$01`).

`DD /ED/`

xxxxxxxx`: C780`

Lies: rückspulbar (`$80`), braucht open/close (`$40`), löschen möglich (`$04`), Ein–/Ausgabe möglich (`$03`), `DIR` erlaubt (`$80`).

`DD /VI/`

xxxxxxxx`: 0500`

Lies: löschen möglich (`$04`), Eingabe möglich (`$01`).

`DD /PP/`

xxxxxxxx`: 2200`

Lies: LF nach CR ergänzen (`$20`), Ausgabe möglich (`$02`).

Hinweis: Wenn (mit `CD`) ein Working-Directory vereinbart wurde und dem Devicenamen kein „Slash“ vorrangeht, so gilt die `DD`–Operation für das mit `CD`–fixierte Gerät („`LDN`“), also nach `CD /ED/` und `DD B2` z. B. für alle `ED`–Files!!

D E F I N E — Define Shell Process

SYNTAX: `DEFINE -- commands to execute`
`DEFINE.sonprocname -- commands to execute`

Mit dem Befehl lassen sich sekundäre Shellprozesse definieren. Dazu bildet die ausführende Shell einen Sohnprozeß mit dem Namen *sonprocname* und beauftragt ihn mit der Ausführung der bis zum Semikolon folgenden Bedienbefehle. Fehlt die Angabe von *sonprocname* – was nicht sinnvoll ist (!) –, so wird vom System ein Name der Form `DEFINE/xx` generiert. Der Sohnprozeß verschwindet nach Ausführung der Kommandos nicht, sondern kann später immer wieder namentlich aktiviert werden. Erst mit einem `UNLOAD`–Befehl wird er wieder entfernt.

Die auf diese Weise als Prozeß definierten Einzelbefehle oder Befehlsgruppen sind anschließend den üblichen Taskmanipulationen – Einplanungen auf Zeit oder Ereignis etc. – zugänglich.

Beispiel:

```
DEFINE.X--DL Y; T X                          Keine erste Aktion!
P.Y /ed/prog LO NO -- Prevent X; All 1 sec X
```

Jede Sekunde wird die Zeilennummer des Übersetzungslaufes angezeigt. (Damit's nicht so langweilig ist . . .) Der Compiler löscht bei erfolgreichem Abschluß die Einplanung für das Blockkommando `X` selbst wieder.

Entwicklung eines Programmes in `/ED/SI`. Start–Task sei `RUN`, Modulname `test`. Modul `test` wurde schon einmal geladen.

```
define.neu--ed--p lo no--unload test*--load--RUN
```

ausprobieren des Programmes etc., dann: `neu`

ausprobieren des Programmes etc., dann: `neu`

ausprobieren, am Ende der Sitzung dann:

```
unload neu,test*
```

Wie man leicht erkennen kann, ist der Turnaround–Zyklus durch das Blockkommando `neu` erheblich angenehmer geworden.

Internes: Der Ablauf entspricht genau dem von allen normalen Sohnprozessen, z. B. `P`, `COPY`, `LOAD`, `ED` etc. mit angeschlossenen Folgebefehlen — lediglich eine eigentliche Operation des Sohnes fehlt, und das Verschwinden am Ende der Aktion wird unterdrückt.

! → Der so erzeugte Shellprozeß ist nicht gut für Mehrnutzerbetrieb geeignet, da er mit eingefrorenen Kopien der Environment-Daten seiner Vatershell arbeitet. Dennoch ist eine gewisse dynamische Redirektion des Outputs mit Hilfe der Station `/TY` möglich:

```
DEFINE.X--O /TY-- LU;
```

Fehler: Wird ein *sonprocname* irrtümlich zum zweiten Mal verwendet, erfolgt die Meldung „`wrong label`“. Eine Operation findet dann nicht statt.

D I R — Directory listing

SYNTAX: `DIR [-E|-A|-EA] dev/pathlist`

Der Befehl zeigt den Inhalt eines Haupt- oder Unterverzeichnisses an. Die Ausgabe erfolgt nach `Stdout`. Die angegebene *dev/pathlist* muß auf ein untergliedertes Gerät oder ein Directory zeigen. Dies kann entweder durch Angabe eines geeigneten Gerätebezeichners – erkennbar an den Geräteparametern gemäß Seite 203, z. B. `/ED/`, `/Fx/`, `/Hx/`, etc.– geschehen oder durch Pfadgebung zu einem Unterinhaltsverzeichnis. Die zur entsprechenden `LDN` (Warteschlangennummer) gehörende I/O–Task schreibt nach Erhalt des Befehles selbsttätig eine Liste der zum bezeichneten Inhaltsverzeichnis gehörenden Files in den `Stdout`-File bzw. auf das `Stdout`-Gerät.

Obwohl der `ED`–Filehandler nur pseudohierarchisch arbeitet, physikalisch also keine Unterdirectories angelegt werden, erhält man auch bei ihm eine Liste aller über den angegebenen Pfad erreichbaren Files.

Parameter: Die folgenden Parameter können beim `DIR`–Kommando angegeben werden. Es werden dann zusätzliche Informationen ausgegeben.

- `-E` gibt Dateien mit Datum und Uhrzeit der letzten Änderung aus.
- `-A` gibt alle Dateien ab dem angegebenen Path aus (inklusive aller darunterliegenden Subdirectories mit ihren Dateien).
- `-EA` gibt alle Dateien ab dem angegebenen Path mit Datum und Uhrzeit aus.

Fehler: Wenn die Geräteeigenschaft kein `DIR`–Kommando zuläßt (siehe `SD`–Befehl Seite 203), so wird der `DIR`–Befehl mit „*befehlsstring*`: operation failed`“ abgewiesen.

Aus der angesprochenen I/O–Task sind die verschiedenen Fehlermeldungen möglich, z.B.:

```
..... DRIVE_NOT_READY (Floppy/Festplatte)
..... TRACK000_NOT_FOUND
..... DIRECTORY_NOT_FOUND
```

Beispiel:
```
DIR /F0/,/F1/,/H1/USR/MUELLER
DIR /ED/MAIER/QUELLFILES
DIR          -> Hier war mit CD ein Working-Directory fixiert!
0 /ED/FILELIST;DIR /F0/
```

```
DIR -E /F0/mist
```

Es werden die Dateien des Subdirectories `mist` mit Datum und Uhrzeit in einer Liste ausgegeben.

```
DIR -A /H1/
```

Ergibt ein Gesamtverzeichnis aller Dateien auf der Winchester `/H1/`.

Hinweis: Mit dem `O`–Befehl kann die Liste in einen beliebigen File gelenkt werden, bei den neueren Systemen auch in einen solchen, der zur `LDN` der mit `DIR` angesprochenen I/O–Task gehört.

Man denke aber daran, daß der File, in den die Ausgabe des `DIR`-Befehles umgeleitet wird, am Ende nicht automatisch geschlossen wird. Dies muß ggf. durch einen Extrabefehl (z. B. Return) nachgeholt werden.

D I S A B L E — Disable Processinterrupt

SYNTAX: `DISABLE EV` *hexnum8*

Beschreibung: Von dem Bitmuster *hexnum8*, das bei Eingabe von weniger als 8 Hexziffern durch Ergänzen führender Nullen gebildet wird, wird das 1–er–Komplement gebildet. Dieses wird logisch „UND“ mit der Event–enable–maske des Systems verknüpft und das Ergebnis nach dorthin zurückgeschrieben. Dadurch werden die durch „Einsen“ in *hexnum8* gekennzeichnete Prozeß–Interrupts gesperrt.

Beispiel: `DISABLE EV 1`

`DISABLE EV FFFFFFFF` (alle Prozeß–IR gesperrt)

`DISABLE EV 0` (unsinnig, ohne Wirkung)

Hinweis: Die Anweisung entspricht der gleichnamigen PEARL–Operation.

Display Line–number of Task **D L**

SYNTAX: `DL` *taskname*

Der Befehl erlaubt einen „Schnappschuß“ bezüglich der momentanen Aktivität der mit *taskname* angegebenen Task, sofern diese die Ausgabe einer „Zeilennummer“ unterstützt und eine Zeilenregisterzelle an der richtigen Stelle ihres Workspace besitzt. Das ist bei PEARL-kodierten Tasks immer der Fall, wenn die angegebene Task oder von ihr benutzte Prozeduren mit der Zeilenmarkiereroption übersetzt wurden und sie zum Zeitpunkt des `DL`–Kommandos Workspace besitzt. Auch die **RTOS–UH** eigenen Assembler, Compiler sowie der `COPY`-Befehl versorgen eine entsprechende Zeilenregisterzelle. Auch bei der Ausführung von Shellskripten kann mit `DL` die aktuell ausgeführte Zeilennummer angezeigt werden.

Beispiel: `DL REGEL`

Hinweis: Es wird die letzte **registrierte** Zeilennummer (bis zu 5 Dezimalstellen) ausgegeben.
Wenn nur Teile des Programms mit der Markeroption übersetzt wurden, kann es sein, daß die Zeilennummer auf einen längst verlassenen Programmpfad zeigt.

Wird der Wert `0` ausgegeben, so wurde noch keine Zeilennummer registriert.

Bei nicht für den `DL`-Befehl geeigneten, assemblerkodierten Tasks wird auch eine scheinbare Zeilennummer ausgegeben, diese stellt jedoch nur einen Zufallswert dar, weil die Zeilenregisterzelle vermutlich für andere Zwecke benutzt wird.

! → Bei älteren Systemen wurde bei Anwendung dieses `DL`–Befehles auf den Assembler, PEARL–Compiler oder einen `COPY`–Sohnprozeß wird die aktuell überlaufene Quellzeilennummer als hexadezimaler Wert ausgegeben. (Bei den aktuellen Systemen dezimale Ausgabe bis max. 65535)

Fehler: Möglich sind:

```
>>... not loaded oder
>>... not active
```

D M — Display Memory

SYNTAX: `DM` *`hex-add-expression`* (Fall A)
`DM` *`hex-add-expr1 hex-add-expr2`* (Fall B)

Beschreibung: Je nachdem, ob ein oder zwei Adreßausdrücke angegeben wurden, sind zwei Betriebsfälle möglich:

Fall A Der Wert von *hex–add–expression* wird auf die nächstkleinere oder gleiche gerade Zahl abgerundet und als Startadresse für die Ausgabe benutzt. Mit dieser Startadresse beginnend werden die Hexadezimalwerte der folgenden 8 Worte aus dem Speicher mit ihren ASCII–Werten ausgegeben. Steuerzeichen werden durch Punkte dargestellt.

Fall B Der Wert von *hex–add–expression1* wird wie im Fall A als Startadresse gewertet. Ist der (ebenfalls auf geraden Wert gerundete) Wert von *hex–add–expression2* kleiner als die Startadresse, so wird er als Anzahl der (mindestens) auszugebenden Bytes gewertet; ist er größer oder gleich der Startadresse, so werden alle Speicherzellen bis mindestens *hex-add-expression2* aufgelistet. In jedem Fall werden ganze Blöcke von 8 Worten (16 Bytes) in jeder Zeile aufgelistet.

hex–add–expression: Eine Folge von maximal 8–stelligen Hexadezimalzahlen, die durch `+/-` Zeichen miteinander verbunden sind. Damit soll dem Anwender in erster Linie das mühselige Addieren/Subtrahieren z. B. von Ladeadressen und relativem Abstand erspart werden.

Beispiel: `DM 5000` Ausgabe der Bytes `$5000` ... `$500F`

`DM 610+20,100` Ausgabe der Bytes `$630` ... `$72F`

`DM 37FF 3901` Ausgabe der Bytes `$37FE` ... `$390D`

Hinweis: Der Zugriff auf die Speicherzellen erfolgt im Usermode des Prozessors. Dadurch ist es bei manchen Rechnern nicht möglich, sich alle Speicherstellen anzusehen, da die Hardware einen „Bus-Error“ auslöst, woraufhin der Shellprozeß abgebrochen wird.

Als Erweiterung hierzu ist ein Zusatzshellbefehl `DMX` zuladbar (oder transient ausführbar), der folgende Zusatzparameter erlaubt:

`-S` Zugriff im Supervisormode, sonst

Zugriff im Usermode

`-B` Zugriff mittels Befehl `MOVE.B` und byteweise Darstellung

`-W` Zugriff mittels Befehl `MOVE.W` und wortweise Darstellung

`-L` Langwortzugriff (`MOVE.L`) und Darstellung

`-M` Zugriff mittels Befehl `MOVEP.W` und wortweise Darstellung, dabei werden nicht gelesene Byte mit einem `?` dargestellt

`-P` Zugriff über PIT–Trap, nur interessant für PBUS–Zugriffe

Fehlt in der Parameterliste des Befehls `DMX` die Angabe `-S`, wird der geforderte Zugriff im Usermode ausgeführt.
Mit dem `DMX` sind dann folgende Beispiele möglich:

`DMX -S 400` zeigt einen Speicherdump ab Adresse `$400`, dabei erfolgt der Zugriff im Supervisormode.

`DMX -B 3000 5` zeigt einen Speicherdump der Adressen `$3000` bis einschließlich `$3005`, der Zugriff erfolgt im Usermode und byteweise.

`DMX -SL 4444` zeigt die Zelle `$4444` in einem Langwort, und der Zugriff erfolgt im Supervisormode und langwortweise.

D R — Display Registers of Task

SYNTAX: `DR` *`taskname`*

Beschreibung: Die angegebene Task wird in der Speicherverwaltung gesucht. Ist sie dort nicht vorhanden, so erfolgt Meldung `>>` *`taskname`* `not loaded`. Ist die Task zwar vorhanden, aber im Zustand `DORM`, so erfolgt Meldung `>>` *`taskname`* `not active`.

Anschließend werden die Register der Task ausgegeben. Die Ausgabe erfolgt in spartanischer Schlichtheit, da die Anweisung ohnehin nur für Bitmuster–Freaks oder Assembler–Programmierer interessant ist:

Adr:	A7 (US)	A7 (SS)	D0	D1
Adr:	D2	D3	D4	D5
Adr:	D6	D7	A0	A1
Adr:	A2	A3	A4	A5
Adr:	A6	St5/4	St3/2	St1/0

Mit `St` sind die letzten 6 Worte des System–Stacks gemeint. So ist z. B. `St1/0` die Adresse hinter einer TRAP–Instruktion, wenn diese auf Taskebene ausgeführt wurde.

Beispiel: `DR XVY`

Hinweis: Bei suspendierten Tasks können die Registerinhalte mit Hilfe des `SM`–Kommandos verändert werden, wenn man die bei `DR` angegebenen Adressen (s. linker Rand) der Registerablageplätze verwendet.

Echo text **E C H O**

SYNTAX: `ECHO` *textstring*

Mit diesem Befehl läßt sich auch ohne den Shellsprachinterpreter ein (fast) beliebiger Text auf dem aktuell gültigen Standard-Output Gerät ausgeben.

```
ECHO Laden des Modules XY fertig;
```

produziert eine Ausgabe des Textes, wobei das Semikolon und das Doppelminus (Zeichenpaar `--`) nicht mehr mit ausgegeben werden, sondern als Beginn eines Nachfolgebefehles an die Shell gewertet werden.

Will man diese Zeichen ebenfalls ausgeben, so muß der entsprechende Textteil entweder mit Hochkommata (`'`) oder mit Gänsefüßchen (`"`) umrahmt werden. Bei einer Umrahmung mit Hochkommata dürfen auch Gänsefüßchen im Text stehen und umgekehrt.

```
ECHO Das Zeichen ';' wird nun gedruckt;  oder
ECHO 'Das Zeichen ; wird nun gedruckt';

ECHO "Das Zeichen ' wird nun gedruckt";
ECHO 'Das Zeichen " wird nun gedruckt';
```

Man beachte, daß es bei der Anwendung des Befehles aus der Grundshell kleine Unterschiede bei der Auflösung „umrahmter“ Texte im Vergleich zum `ECHO`-Befehl der Shell-(skript-)sprache gibt. Hat man mit dem optionalen `ENVSET`-Befehl eigene oder globale Environment-Variablen der Shell angelegt, so werden diese als Argumente des `ECHO`-Befehles auch im Falle einer Umrahmung substituiert.

```
ENVSET MIST=ABCDE;
ECHO '$MIST';
```

ergibt den Text `ABCDE` als Ausgabe.

E D — Edit a File

SYNTAX:
`ED.`*sonprocname* `[PRIO` *integer3*`]` `[`*parameterlist*`]` (Form 1)
`ED [PRIO` *integer3*`]` `[`*parameterlist*`]` (Form 2)

Beschreibung: Es wird eine flüchtige unabhängige Task mit vom Nutzer (1. Form) oder vom System (2. Form) vorgegebenem Namen erzeugt und gestartet. Ein vom System (2. Form) generierter Name hat den Aufbau `ED/`*xx*. Die Standardpriorität für den Editor ist 15 (`$0F` hexadezimal). Der Editor ist „reentrant“, es kann also mit dem gleichen Code (auch ROM–Resident) auf mehreren Terminals gleichzeitig gearbeitet werden — es müssen nur unterschiedliche Dateien editiert werden.

Parameter: Es werden die Parameter `SC` (Scratch–Datei, auf der der Editor arbeiten soll) und `SI` (Source–Input, Port, über den der Nutzer Eingaben tätigt) akzeptiert.

`/SC/` muß eine `ED`–Datei der Form `/ED/xyz` sein, sonst wird der Editor sofort mit der Meldung `wrong ldn (mode)` abgebrochen und verschwindet aus dem System. `/SI/` muß ein Sichtgerät sein, welches im System als „dialogfähig“ bekannt ist (siehe `SD`–Befehl), sonst erfolgt ein Abbruch mit der Meldung wie bei falschem `SC`–Paramter. Die Angabe von `SC` kann entfallen. Die folgenden Aufrufe sind äquivalent:

`ED /ED/`*name* bzw.
`ED SC /ED/`*name* und
`ED SC=/ED/`*name*

Bei fehlenden Parametern wird `SC=/ED/SI`*x* und `SI=/A`*x*`/` eingesetzt, wobei *x* die `LDN` der Schnittstelle des Nutzer ist.

Sichtgerät: Es ist nahezu jedes Terminal oder aber das **RTOS–UH**-Windowsystem als Arbeitsplatz geeignet, wenn von der Struktur folgende Bedingungen erfüllt werden:

- Typ: `TELEVIDEO TVI 925`, `950` oder `VT52` und kompatible.
- 24 oder 25 Zeilen mit jeweils 80 Zeichen.
- Automatischer UP–Scroll bei Lf in der untersten Bildschirmzeile.
- Autowraparound = autom. Cr + Lf nach Anschlag des 80. Zeichens einer Zeile (No Wrap parametrierbar)

- Kein automatischer LF nach Erhalt eines CR.
- Cursorsteuerung über `$0A`, `$0B`, `$0C`, `$08`, `$16` (Umparametrierung auf ESC–Sequenzen möglich).
- Betrieb wie bei der Shell im Full–Duplexmode.

Steuerung: Nach dem Start wird die erste Seite des Files aufgeblättert. Existierte der File vorher nicht, so wird er mit der Zeile

```
*File was installed by ED
```

als einziger Information neu im System eingerichtet. Der Editor arbeitet ohne Zwischendatei direkt auf dem angegebenen ED–File, Änderungen sind daher von sofortiger Wirkung auf die Datei.

Die Veränderung einzelner Zeichen erfolgt durch Anfahren der Position auf dem Bildschirm und Eingabe eines neuen Zeichens (Replace–Mode). Der Editor arbeitet stets im Replace–Mode.

Sonderfunktionen werden durch Anschlag des Zeichens `ESC` angewählt, wobei der nachfolgende Buchstabe die Operation bezeichnet. Die Operationsbezeichner wurden im Sinne einer leichten Merkbarkeit gewählt (siehe folgende Operationstafeln).

Es sollten nur Zeichen eingegeben werden, die einen ASCII–Wert von größer `$1F` haben. Enthält die Datei Zeichen, deren Wert kleiner als `$1F` ist — z. B. durch einen `COPY`-Befehl erzeugt —, so werden diese mit dem Zeichen @ abgebildet.

Abweichend von dieser Regel darf in die erste Spalte ein Zeichen `$04` (EOT) eingegeben werden, um das Ende des Textes zu markieren (für eine Übertragung via Schnittstelle wichtig).

Betriebsmodi: Der Editor kennt zwei Betriebsmodi, die durch den `SD`–Bedienbefehl der Shell, siehe Seite 203, verändert werden können.

Der Normalmode ist standardmäßig eingestellt. Der Cursor–ESC–Mode kann mit dem Kommando `SD /Ax/+1 01` eingestellt werden. Der Normalmode wird mit `SD /Ax/+1 00` eingestellt.

Die ESC–Sequenzen für Ein–/Ausfügungen wurden so gelegt, daß möglichst viele unterschiedliche Terminals benutzt werden können.

Tabelle der vom Editor genutzten Zeichen zur Cursorsteuerung:

Funktion	Normalmode	Cursor–ESC–Mode
↓	$0A, $16, ^J oder ^V	ESC B
↑	$0B oder ^K	ESC A
→	$0C oder ^L	ESC C
←	$08 oder ^H	ESC D
Insert Char	ESC →	ESC Q
Delete Char	ESC ←	ESC W / ESC P
Insert Line	ESC ↓	ESC E / ESC L
Delete Line	ESC ↑	ESC R / ESC M
Terminals:	TELEVIDEO TV925	DEC VT52

Funktionen: Die folgende Tabelle enthält die über Escape-Sequenzen anwählbaren Befehle. Der Editor "merkt" sich also keinerlei „Kommandomodus“ o. ä., sondern erkennt dies an dem Anschlag der Esc-Taste.

Mnemonik	Code	Erklärung
Forward 10 Li	ESC F	Fenster 10 Zeilen weitersetzen.
Exit	ESC X	Ausstieg aus dem File, Beendigung des Editors.
Home	ESC H	1. Zeile, 1. Spalte des Files anlaufen und Keybuffer löschen. Der File wird verdichtet.
Ins Keybuffer	ESC I	Der Keybuffer wird vor der Zeile, in der der Cursor steht, eingesetzt.
Keybuffer ed.	ESC K	Der Keybuffer wird angelaufen, und es können in ihn Zeichen eingesetzt werden.
New Numbers	ESC N	Alle Zeilennummern werden aktualisiert und der Keybuffer gelöscht.
Overlay Keyb.	ESC O	Der Keybuffer wird mit der Zeile, in der der Cursor steht, ab der Position des Cursors gefüllt.
Search Keyb.	ESC S	Es wird in dem File nach dem im Keybuffer enthaltenen Text gesucht. Die Suche beginnt ab der aktuellen Cursorposition in Richtung Fileende.
Tabulator set	ESC T	Tabulatormarke setzen.
Unmask Tabul.	ESC U	Tabulatormarke löschen.
Verify Pict.	ESC V	Fenster aus der Datei neu nachladen. Hilft bei unklaren Situationen.
Zone Select	ESC Z	Das Fenster wird auf die angegebene Zeilennummer neu positioniert.

Bei den o. a. Escape–Sequenzen dürfen auch kleine Buchstaben mit gleicher Wirkung benutzt werden. Es empfiehlt sich, von Zeit zu Zeit über `ESC H` eine Verdichtung des Files zu forcieren, insbesondere wenn nur noch wenig Platz im Speicher ist. Bricht der Editor seine Bearbeitung mit der Meldung `... no mem. suspended` ab, so kann nach Bereitstellung von genügend Speicher die Task mit `CONTINUE` fortgesetzt werden. Gelingt dies nicht, so kann der ED–File dennoch gelesen werden, allerdings ist die zuletzt aufgeblätterte Seite nicht im aktuellen Zustand. Es sollte nun unbedingt die Task mit einem `UNLOAD` entfernt werden, da bei einer Fortsetzung des Editors nach gewaltsamer Veränderung des Files ein Absturz bzw. eine `BAD POINTER. EXITUS` Meldung erscheint und Veränderungen am File nicht ausgeschlossen sind.

! → Die übergeordnete primäre Shell des Terminals ist während einer Editor–Sitzung über `CTRL A` nicht erreichbar. Sie müssen vorher über `ESC X` aussteigen. Allerdings kann über die `BREAK`–Taste dennoch die Shell bei gleichzeitig aktivem Editor aktiviert werden. Dies ist jedoch nur für den Notfall vorgesehen. Die aktuelle Seite des `ED`–Files muß danach über `ESC V` restauriert werden.

Zum Löschen der Anzeige des Keybuffers muß dieser über `ESC K` angelaufen und dann mit ↑ (Cursor UP) verlassen werden. Der Inhalt bleibt aber weiterhin erhalten (z. B. für die Funktion Suchen: `ESC S`).

Wenn das verwendete Terminal (z. B. ITOH CT 101) keinen Auto–Wrap–Mode besitzt oder dieser nicht wie oben angegeben funktioniert, so kann mit `SD /A`*x*`/+1` *yy* der Editor umparametriert werden. Das Setzen des Bit mit Wertigkeit 2 in *yy* bewirkt, daß die LF–Generierung nun vom Editor übernommen wird. Das Terminal muß nun so eingestellt werden, daß nach Anschlag des 80. Zeichen in einer Zeile der Cursor in der 80. Spalte stehen bleibt. Es können z. B. mit `SD /A`*x*`/+1 3` die Funktionen „curs–ESC“ und „no wrap around“ vereinbart werden.

Beispiel: `SD /A2/ 3301` Port2: dialog, LF nach CR, Curs=ESC, IN/OUT

`ED /ED/marion` Der File `marion` wird editiert.

Nach der Vereinbarung eines „Working-Directories“ (siehe Befehl `CD`) können die Dateien hierarchisch organisiert werden:

Der Nutzer hat das Working-Directory `/ED/ABC` angelegt. Die Eingabe von:

`ED Affe` richtet eine Datei mit dem Namen `/ED/ABC/Affe` ein, und diese wird editiert.

Durch die Verwendung von Working-Directories kann im Mehrnutzerbetrieb ein versehentliches Benutzen von Dateien anderer Nutzer vermieden werden. Dazu richten sich alle in einem System arbeitenden Nutzer unterschiedliche Working-Directories ein, über diese erfolgt dann der Zugriff auf die einzelnen den Nutzern zugeordneten Dateien.

Nutzer1 legt mit `CD /ED/Nutz1` sein Working-Directory für sich fest, Nutzer2 legt `CD /ED/Nutz2` fest usw.

Wenn nun beide Nutzer das Kommando:

`ED mein` eingeben, editieren sie die folgenden im System enthaltenen Files:

Nutzer1: `/ED/Nutz1/mein` und
Nutzer2: `/ED/Nutz2/mein` usw.

Enable Processinterrupt **E N A B L E**

SYNTAX: `ENABLE EV` *hexnum8*

Beschreibung: Mit dem aus *hexnum8* erhaltenen Bitmuster werden die Prozeßinterrupts freigegeben, an deren Bitposition eine `1` in *hexnum8* enthalten ist. Es können 32 verschiedene Prozeßinterrupts angesprochen werden. `ENABLE EV 1` Prozeßinterrupt Nr.1 wird

Beispiel: freigegeben

`ENABLE EV FF` Nr.1–8 werden freigegeben

`ENABLE EV 0` unsinnig, da ohne Wirkung

Hinweis: Diese Anweisung entspricht der gleichnamigen Anweisung der Sprache PEARL und erfüllt die gleiche Funktion auf System–Kommandoebene.

Nach einem Kaltstart des Systemes sind zunächst alle Prozeßinterrupts abgeschaltet. Die jeweils benötigten müssen daher vor ihrer Benutzung mit diesem Kommando oder durch die entsprechende PEARL-Anweisung eingeschaltet werden.

E N V S E T — Environment Set (optionaler Bedienbefehl)

SYNTAX:
```
ENVSET variable=textstring
ENVSET -G variable=textstring
ENVSET -R variable
ENVSET -R
ENVSET -G -R variable
ENVSET -G -R
ENVSET -S size
ENVSET -G -S size
ENVSET
ENVSET -G
```

Mit dieser Anweisung können lokale (arbeitsplatzgebundene) oder globale (`-G`, für alle Arbeitsplätze gleich) Environment-Variable mit einem Textwert besetzt werden. Die Parameter (R,S und G) können auch klein geschrieben werden. Mit `ENVSET` definierte lokale oder globale Variablen können mit vorangestelltem `$`-Zeichen in Shellbefehlen an Stelle des Textes, den sie beinhalten, benutzt werden. Das Objekt *textstring* endet am Semikolon bzw. am Doppelminus (`--`).

Die Environmentvariablen werden in speziellen Speichermodulen (`#ENV/x`) angelegt, deren Größe (Defaultwert ist `$200=512` Bytes) man nur mit dem allerersten `ENVSET`-Befehl (also bei noch nicht existierendem Environment!) einstellen kann:

```
ENVSET -S 1000                          |4 kB lokal vorsehen
ENVSET -G -S 2000                       |8 kB globales Env.

ENVSET PF=/H0/TEX/TEST.TEX              |Lokale Definition
COPY $PF > /ED/A                        |Benutzung

ENVSET -G SV=CP /H0/xy > /H0/xy.bak     |Globale Definition
$SV                                     |Benutzung
```

Der Befehl `ENVSET` ohne Parameter erzeugt eine Liste aller zur Zeit gespeicherten lokalen Environment-Variablen und listet zusätzlich alle eventuell vorhandenen unveränderlichen Systemvariablen auf. Der Befehl `ENVSET -G` erzeugt analog dazu eine Liste aller zur Zeit gespeicherten globalen Environment-Variablen.

Der Zusatzparameter `-R` (Remove) gestattet das Löschen aller oder einer einzelnen Environmentvariablen.

```
ENVSET                  |Listet alle lokalen Variablen, auch Sysvars
ENVSET -R SV            |Lösche die lokale Variable SV
ENVSET -R               |Lösche alle lokalen Variablen
ENVSET -G -R            |Lösche alle globalen Variablen
```

Beim lokalen `ENVSET`-Befehl werden neben den zur aktuellen Shell gehörende Environmentvariablen auch eingebaute platzgebundene Variablen angezeigt. Dabei können jedoch diese eingebauten Variablen, etwa `WORKDIR`, `EXEDIR..`, `STDOUT`, `STDIN`, `STDERR`, `EDITOR`, `TIMEBASE` und `P_TYPE` nur angezeigt, jedoch nicht gelöscht oder verändert werden. Zu deren Veränderung – soweit überhaupt möglich – bediene man sich der dafür vorgesehenen Befehle, z.B. `ER`, `I`, `O` sowie `CD` usw.

Führt man den `ENVSET`-Befehl aus einem Skript heraus aus (siehe Seiten 76 ff.), so wird der Befehl vom Shellinterpreter selbst dekodiert und neben dem globalen oder lokalen Environment wird auch eine gleichnamige Skriptvariable gesetzt oder verändert – und zwar unabhängig davon, ob das lokale oder globale (-G Option) Environment angesprochen wurde.

Beim Ersetzen der Environmentvariablen durch ihren (Text-)Wert (durch die Shell oder im Skript) wird zunächst immer im lokalen Environment gesucht. Nur wenn das Objekt dort nicht gefunden wurde, erfolgt eine Suche im globalen (für alle Nutzer gleichen) Environment.

Wird die Bourne-Shell von der primären Shell aus gestartet, so erbt sie alle zum Startzeitpunkt definierten Environmentvariablen als Kopie. Dabei werden zunächst alle globalen Variablen kopiert und als Skriptvariable angelegt. Anschließend werden alle lokalen Variablen übernommen – wobei ggf. gleichnamige aus dem globalen Environment stammende Skriptvariablen überschrieben werden. Damit ist gesichert, dass das lokale Environment quasi eine höhere Priorität besitzt.

Verändert man in einem Skript die Inhalte der in diesen Skript hineinkopierten Variablen durch Wertzuweisung (`...=...`), so ändert sich nichts am Environment – eben auch nicht an dem, aus dem die Objekte kopiert wurden.

E R — Error Redirect

SYNTAX: `ER` *pathlist*

Beschreibung: Als Standard-Error Datenstation (`Stderr`) der Shell, die diesen Befehl ausführt, wird fortan die durch *pathlist* bezeichnete Datensenke verwendet. Die Wirksamkeit beschränkt sich auf die Kommandos im Rest der Kommandozeile. Der Befehl ist darum im Gegensatz zum „`PER`“-Befehl nicht riskant: die Fehler der nächsten Befehlszeile schreibt die Shell weiterhin in die bisherige Datenstation.

Hat man ein PEARL-Shellmodul geschrieben, welches nach `Stderr` seine Daten schreibt, so kann man durch Vorschalten dieses Befehles ohne weiteren Kodieraufwand jede beliebige Datensenke des aktuellen Rechners nutzen.

Beispiel:
```
ER /HO/NIL; MKDIR /HO/POOL;
```

In diesem Beispiel erspart man sich die lästige Fehlermeldung, falls das Directory „`POOL`“ schon vorhanden war. Man bekommt allerdings auch nicht mit, wenn auf der Platte kein Platz mehr ist o. ä.

Man beachte, daß die Shell vor dem Hineinschreiben in die Datenstation `Stderr` den File nicht öffnet, das macht der Handler der Datenstation notfalls automatisch. Auch wird der File am Ende nicht geschlossen. Auf diese Weise ist das akkumulierende Sammeln von Fehlermeldungen in einem File möglich, man muß allerdings dafür Sorge tragen, daß der File irgendwann geschlossen wird oder häufig genug `SYNC`-Befehle einstreuen.

! → Eine wichtige Bedeutung hat der `ER`-Befehl in der Shellsprache, z. B. um in einem Skript Fehlermeldungen selbst anzunehmen. In der grafischen Bedienoberfläche kann man mit ihm Meldungen in eine „Alert-Box“ lenken.

List Files **F I L E S**

SYNTAX: `FILES` *`device--list`*

Es werden die Namen der auf den angegebenen Geräten zur Zeit **aktiven** Files ausgegeben. Das sind Dateien, die sich im geöffneten Zustand befinden, weil aus ihnen gelesen wurde oder etwas hineingeschrieben wurde, ohne daß es bisher einen abschließenden Close- (=`RETURN`-)Befehl gegeben hätte. Auch nach einer evtl. `REWIND`-Operation ist ein File im aktiven Zustand.

Bei Benutzung eines Working-Directories (siehe `CD`-Befehl) werden nur die dem vereinbarten Working-Directory zugeordneten Files ausgegeben.

Beispiele: `FILES /ED/ /F0/` alle aktiven `/ED/`– und `/F0/`–Files

`FILES /F1/` alle aktiven `/F1/`–Files

Fehler: Wird ein Gerät addressiert, das nicht in Files untergliedert ist, so erfolgt die Meldung

befehlsstring: `operation failed`

und die weitere Bearbeitung der Befehlszeile unterbleibt. Wird dagegen ein nicht existierendes Verzeichnis angewählt, so wird dies nicht moniert, sondern angegeben, daß darunter keine aktiven Files gefunden wurden.

F I N D — File Index

SYNTAX: `FIND` [*parameter*] *devpath1 devpath2...*

Mit dieser Anweisung läßt sich ein sogenannter File-Index erstellen, der für alle File-Verzeichnisse auf allen Geräten die stets gleiche Ausgabeform hat. Der Befehl ist im Gegensatz zu `DIR` oder `FILES` darum besonders gut geeignet, um File-Verzeichnisse von Skripten in Shellsprache normiert erstellen und abarbeiten zu können. Skripte wie `CPX` u. ä. stützen sich auf `FIND` ab.

Man erhält für jeden File die volle zur Verfügung stehende Information, so auch den „Startblock“, mit dem sich z. B. beim **RTOS–UH**-eigenen Floppy/Festplattenfilehandler Files evtl. sogar noch nach Verlust des Directories lesen lassen.

Jeder File wird mit seinem **kompletten** Zugriffspfad aufgelistet, jedoch ohne den führenden */dev/*-Pfadanteil.

devpath: Hierbei muß es sich um einen Bezeichner für ein Verzeichnis handeln. Der Befehl veranlaßt den adressierten E/A-Treiber, die unten angegebene Information nach `Stdout` zu senden. Ohne Zusatzparameter werden nur die Files und Unterverzeichnisse der mit *devpath* angesprochenen Ebene aufgelistet.

parameter: Es ist nur die Option

`-A` (oder `-a`) „alle“

vorgesehen. Damit wird erreicht, daß nun auch alle Unterverzeichnisse sowie deren Unterverzeichnisse (etc.) aufgeschlüsselt werden. Vorsicht: Das kann bei Festplatten einen fürchterlich langen File-Index erzeugen!

Beispiele:
```
FIND /H0/XD
FIND -A /ED/SIMULA
```

quasi „genormte“ Struktur einer Ausgabezeile sieht wie folgt aus:

path	no. of bytes	Uhrzeit	Datum	Start-block
`SIMULA/test`	`32627`	`09:51`	`27-05-1993`	$00041A16

Fehler: Wenn das Gerät nicht in Verzeichnisse untergliederbar ist, so wird mit „`... operation failed`“ die Bearbeitung der laufenden Befehlszeile abgebrochen.

Formatting of a Floppy or Harddisc **F O R M**

SYNTAX: `FORM S` */Floppydevice/forminfo* (single density)
`FORM D` */Floppydevice/forminfo* (double density)
`FORM D` */Harddiscdev/forminfo* (Festplatten nur double!)

Auf dem angegebenen Floppylaufwerk wird neben der eher hardwaretechnischen Softsektorierung auch das Filesystem des Filemanagers installiert. Alle bisher auf der Floppy/Festplatte gespeicherten Daten sind unwiderruflich verloren. Die auf das `FORM`–Kommando folgenden Eingaben müssen in Großbuchstaben eingegeben werden.

`S` Single–Density Disketten (kaum noch unterstützt)
`D` Double–Density Disketten

forminfo: Die Formatanweisung wird aus 3 Parametern gebildet. Im ersten wird mit einer Buchstabe/Zahl-Kombination mitgeteilt, welche Aufteilung in wieviel Sektoren und ob 5“/3,5“ oder 8“ formatiert werden soll. Der zweite Parameter gibt eine ein– oder doppelseitige Formatierung an. Im dritten Paramter wird mit einer 1– bis 4–stelligen Zahl die Anzahl der Tracks mitgeteilt.

Der 1. Parameter:

xy	seclen	sec/track	blk/track	typ. Medium
A5	256	16	1x4k	3.5“ und 5“ DD-Disk. RTOS
B5	1024	5	1x5k	3.5“ und 5“ DD-Disk. RTOS
C5	512	9	1x4.5k	3.5“ und 5“ DD RTOS+MSDOS
A8	256	26	1x6.5k	8“ und 5“ HD-Disk. RTOS
B8	1024	8	1x8k	8“ und 5“ HD-Disk. RTOS
C8	512	15	1x7.5k	8“ und 5“ HD RTOS+MSDOS
AH	256	32	2x4k	3.5“ HD-Disk. RTOS
BH	1024	10	2x5k	3.5“ HD-Disk. RTOS
CH	512	18	2x4.5k	3.5“ HD-Disk. RTOS+MSDOS
BJ	1024	20	4x5k	3.5” ED-Disk. RTOS
CJ	512	36	4x4.5k	3.5” ED-Disk. RTOS
W5	512	Blocksize 4k		SCSI Harddisc
W6	512	Blocksize 16k		SCSI Harddisk
X5				Platte mit 1024-er Sektoren

Der 2. Parameter:

SS	–	single sided Disketten
DS	–	double sided Disketten
Hz	–	mit z Köpfen der Winchester
B$xxxxx$	–	SCSI-Platte mit xxxxx (dezimal) Blöcken, die eine Größe haben, wie sie durch den ersten Parameter des `FORM`-Befehles festgelegt ist. (z. B. 4 kByte)

Der 3. Parameter: (entfällt bei „B“ als zweitem Parameter!)

1– bis 4–stellige dezimale Anzahl Tracks.

! → Nach dem Formatieren wird jeder Block gelesen und im Fehlerfall für das Filesystem ausgesondert. Bei mehr als 9 unbrauchbaren Blöcken wird die Operation mit „`wrong i/o`“ oder „`ABORTED_COMMAND_ERROR`“ abgebrochen. Zur Zeit werden die Blöcke aber nicht auf Übereinanderfaltung geprüft (z. B. wenn Drive bei `DS` nur eine Seite schreiben kann).

Fehlermeldungen:

```
>> ... :Fx DRIVE_NOT_READY
>> ... :Fx DEVICE_WRITE_PROTECTED
>> ... :Fx TRACK_000_NOT FOUND
>> ... :Fx ID_FIELD_NOT_FOUND (Beim Verify)
```

Beispiele: `FORM D /F0/B5SS80` single sided **RTOS–UH**–Disk
`FORM D /H0/W5B21800` SCSI-Festplatte, ca. 88 Mbyte

Nach dem Absetzen des Kommandos `MSFILES /F0/`:

`FORM D /F0/C5DS80` double sided MS–DOS–Disk

Inspect FREE Storage on Disc **F R E E**

SYNTAX: `FREE` *`devpath`*

Beschreibung: Es wird der noch vorhandene freie Platz auf dem angegebenen Gerät – eine Floppy oder Festplatte – ausgegeben.

devpath: Nur der Geräteteil wird ausgewertet. Es muß sich um ein Gerät mit der Eigenschaft „formatierbar“ (siehe Seite 203) handeln, sonst beendet die Shell die laufende Zeile mit der Fehlermeldung „`... operation failed`“. Wurde ein Working-Directory vereinbart (siehe Befehl `CD`), wirkt der Befehl auf das im Directoy vereinbarte Device.

Beispiele: `FREE /F1/`

Es wurde das Working-Directory `/F0/` vereinbart. Der Befehl

`FREE;`

wirkt dann auf das Floppydevice `/F0/`.

G O — Go and execute

SYNTAX: `GO AD` *hexadr*
`GO.`*sonprocname* `[PRIO` *integ3*`] [SZ` *sizehexnum*`] AD` *hexadr*
`GO.`*sonprocname* `AD` *hexadr*

Beschreibung: Es wird vom System ein Sohnprozeß gebildet, der entweder einen Systemnamen `GO/`*xx* oder den angegebenen Tasknamen (2. Form) erhält. Die Task wird mit dem minimal möglichen Workspace ausgestattet und erhält die angegebene Startadresse. Anschließend wird sie mit der angegebenen Priorität bzw. der Defaultpriorität 50 gestartet.

Beachte: Dies ist eine Hilfskonstruktion für maschinennahe Programmierung. So kann eine kurze Codesequenz schon mal ohne jeden Taskheader zur Ausführung gebracht werden. Auch können im ROM liegende User-Tasks ggf. über diesen Weg exekutiert werden. Privilegierte Befehle führen zur Fehlermeldung „`>>` *xyz* `not privileged`“.

Beispiele:
```
GO.TEST AD 2000

GO AD 5000 SZ 5000

GO.CHECK PRIO 2 AD 8E000
```

Hinweis: Es lassen sich noch `SI` und `CO`–Parameter angeben, die jedoch nur dann Bedeutung haben, wenn der Anwender sich die Informationen aus dem (nur Insidern bekannten) Header des generierten Sohnprozesses selbst herausholt.

Durch Angabe eines `SZ`–Parameters kann die Größe des für den Sohnprozeß angeforderten Task–Workspace vorbestimmt werden (siehe Beispiel).

! → Wenn angegeben, darf der SZ-Parameter die zum Überleben des Systemes notwendige Mindestgröße keinesfalls unterschreiten, da der Kontext in den Taskworkspace passen muß. Ein für alle bisherigen Hardwareplattformen ausreichender Wert ist `SZ=100`.

Help **H E L P / ?**

SYNTAX:

`HELP`	Form a)
`HELP -D`	Form b)
`HELP -E`	Form c)
`XHELP`	Form d)
`?`	identisch zu Form a)
`? -D`	identisch zu Form b)
`? -E`	identisch zu Form c)

Mit diesem Bedienbefehl kann eine Kurzinformation über das aktuelle System angefordert werden:

a) Ohne Parameter: Es wird eine Liste der in der Shell vorhandenen speicherresidenten Bedienbefehle ausgegeben. Die Liste ist in der Reihenfolge geordnet, in der die Befehle vom Scanner erfaßt werden. Auch speicherresident hinzugeladene eigentlich transiente Shellerweiterungen werden aufgelistet, sofern sie aktiv ansprechbar sind. Die Suche im RAM wird allerdings erst nach der Suche im Systembereich gestartet.

b) Mit Parameter `-D` oder `-d`: Es werden die Datenstationen des aktuellen Systemes aufgelistet. Dabei wird auch ihre `LDN` und die Laufwerksnummer `DRIVE` ausgegeben. Vorsicht: Es wird hexadezimale Kodierung verwendet, während der Datenstationscode `/LD/x.y/` bei `x` und `y` Dezimalzahlen erwartet.

c) Mit Parameter `-E` oder `-e`: Es werden die im System vorhandenen fest eingebauten globalen Symbole samt der zugehörigen Adresse ausgegeben. An Hand des Vorspannes „∼“ lassen sich PEARL90-Symbole erkennen. Leider sieht man nicht, um welche Art Objekt es sich dabei genau handelt. Symbole, die mit dem Zeichen `#` beginnen, dienen der Selbstkonfiguration des Betriebssystems und sind für den Nutzer nicht zugänglich.

d) Der Aufruf über `XHELP` entspricht der Form a) – jedoch wird für jeden Bedienbefehl eine komplette Zeile ausgegeben. Die Befehlsnamen werden dadurch ungekürzt ausgegeben. Außerdem kann evtl. vorhandener Beschreibungstext (siehe Seite 68) ausgegeben werden. Wird `XHELP` mit Zusatzparametern aufgerufen, so verhält er sich wie der normale `HELP`-Befehl.

I — Input–device specification

SYNTAX: `I` *pathlist*

Beschreibung: Als Standard-Input (`Stdin`) der Shell, die diesen Befehl ausführt, wird fortan die durch *pathlist* bezeichnete Datenquelle verwendet. Die Wirksamkeit beschränkt sich auf die Kommandos im Rest der Kommandozeile. Der Befehl ist darum im Gegensatz zum „`PI`“-Befehl nicht riskant: die nächste Befehlszeile liest die Shell weiterhin vom bisherigen Gerät.

Hat man ein PEARL-Shellmodul geschrieben, welches von `Stdin` seine Daten liest, so kann man durch Vorschalten dieses Befehles ohne weiteren Kodieraufwand jede beliebige Datenquelle des aktuellen Rechners nutzen.

Beispiel:
```
I /HO/SOURCE/TEXT1.TXT; CONVERT;
```

`CONVERT` könnte eine PEARL-kodierte Shell sein, der man im obigen Beispiel einen Inputfile `.../TEXT1.TXT` anbietet. Eine weitere wichtige Bedeutung hat der „`I`“-Befehl in der Shellsprache, z. B. beim `READ`.

List Task States **L**

SYNTAX: `L [-S[O]|-U[O]] | [-O]`

Beschreibung: Für alle im System existierende Tasks werden die Statusinformationen aufgelistet. Durch die Angabe der Parameter kann eine Selektierung der Ausgabe erfolgen.

- S Es werden nur Systemtasks — alle, die mit einem `#` beginnen — aufgelistet.
- U Es werden nur die Usertasks, die sich im System befinden, aufgelistet.
- O Es werden nur die Tasks aufgelistet, die dem jeweiligen Nutzer zur Zeit „gehören“.

Beispiel: `L` Es werden alle Tasks aufgelistet.

`L -UO` Es werden die eigenen Usertasks gelistet.

Die Statusinformation hat folgenden Aufbau:

`Adr Prio/User (resident) Status TWS=`*xxxx* `PC=`*xxxx* *Name*

Prio: Gibt die Priorität der Task an. Prioritäten von `$1` – `$FFF` sind Anwenderprioritäten. Negative Prioritäten liegen über den Anwenderprioritäten und sind dem Betriebssystem vorbehalten.

User: Gibt an, welchem User die Task zugeordnet ist. User Nummern sind fortlaufend von 1 bis *n*, entsprechend der Zahl der seriellen I/O–Kanäle vergeben.

Resident: Erscheint bei Tasks, deren Taskworkspace auch nach dem Terminieren der Task erhalten bleibt. Bei zyklisch aktivierten Tasks mit kurzen Einplanungsintervallen kann das „Resident„ Attribut zur Verkürzung der Verwaltungszeiten benutzt werden.

TWS= Hier wird die Adresse des Taskworkspace ausgegeben. Ist sie `00000000`, so besitzt die Task noch keinen Workspace (z. B. hat sie noch keine CPU–Zeit bekommen, oder es war noch zu keinem Zeitpunkt genügend Platz).

PC= Hier wird der letzte vom Dispatcher auf Task–Grundebene gültige und festgestellte Wert des Programmzählers ausgegeben.

Name: Gibt den Namen der Task an. Tasknamen, die mit `#` beginnen, sind Systemtasks und lassen sich nicht mit `UNLOAD` entfernen.

Status: Hier werden die folgenden Abkürzungen eingetragen:

- `CWS?` Task wartet auf Zuteilung eines Communication–Elementes, weil sie infolge reger Ausgabetätigkeit auf ein langsameres Gerät ihr Kontingent ausgeschöpft hat.
- `DORM` Die Task ruht zur Zeit, keine Aktivität.
- `I/O?` Task wartet auf Beendigung einer Ein–/Ausgabe.
- `PWS?` die Task wartet darauf, daß irgendwo ein passendes Speicherstück frei wird. In der Regel wird auf Procedurworkspace gewartet, es ist aber auch möglich, daß auf ein „CE“ gewartet wird, wobei das zustehende Kontigent (s. `CWS?` oben) noch nicht ausgeschöpft ist.
- `RUN` Die Task ist lauffähig oder läuft.
- `SCHD` Task ist für eine Aktivierung vorgeplant, die entsprechende Bedingung (Zeit,Ereignis) ist aber noch nicht eingetreten.
- `SEMA` Task wurde durch vergebliches `REQUEST` auf eine Semaphorvariable oder vergebliches `RESERVE` bzw. `ENTER` auf eine Boltvariable blockiert. Es ist auch möglich, daß die Task mit Hilfe des `WFEX`-Traps auf die Beendigung einer anderen Task wartet. Shellprozesse im `WAIT`-Mode warten auf diese Weise auf Sohnprozesse.
- `SUSP` Task wurde suspendiert und wartet auf die `CONTINUE`–Operation. Bei System I/O–Tasks wird auf das Ende des laufenden Records gewartet.
- `????` Die Task ist durch mehrere Bedingungen gleichzeitig blockiert, z. B. weil sie durch eine externe `SUSPEND`–Operation zusätzlich verriegelt wurde. Dieser Zustand wird vom Systemkern auch eingestellt, wenn diese im Supervisormode laufen und dabei einen Fehler verursacht haben.

Tabelle 3.6: Kurznamen der Taskzustände

install Lineedit (optional) **L E / L I N E E D I T**

SYNTAX: `LINEEDIT [options][device]`

Beschreibung: Die Befehle `LE` bzw. `LINEEDIT` installieren, konfigurieren bzw. deinstallieren den Zeileneditor LINEEDIT für das angegebene Eingabe-Gerät. Der LINEEDIT verwaltet die über ein dialogfähiges Datenterminal erfolgten Eingaben.

Der LINEEDIT bietet u.a. die Möglichkeit, über die Cursor-Tasten alte Zeilen zurückzuholen, ggf. zu modifizieren und dann erneut der Shell zur Bearbeitung zu übergeben.

Um die gewünschte Eingabezeile schnell zu finden, kann man auch die ersten Zeichen der gewünschten Zeile eingeben und danach die Cursortasten betätigen. Der LINEEDIT sucht dann nur nach den Zeilen, die mit den angegebenen Zeichen beginnen. Findet er keine solche Zeile, so zeigt er weiterhin alle Zeilen an.

Wird eine Eingabe abgeschlossen, so wird sie im sogenannten History-Buffer abgespeichert. Ist der Puffer des LINEEDIT vollständig gefüllt, so werden bei neuen Eingaben so viele alte Zeilen gelöscht, bis genügend Platz vorhanden ist. Alte Zeilen werden auch bei mehrfacher Auswahl nur einmal im Puffer abgelegt.

Der History-Buffer des LINEEDIT läßt sich auch in eine Textdatei abspeichern und zu einem späteren Zeitpunkt wieder einlesen. Man kann also z.B. nach dem Einschalten des Rechners direkt die Befehle noch einmal benutzen, die man tags zuvor eingetippt hat. Natürlich kann man auch von Hand eine Textdatei mit allen für ein spezielles Projekt benötigten Befehlen mit einem Editor erstellen. Wann immer man an diesem Projekt arbeiten möchte, lädt man die Datei in den LINEEDIT und kann die Befehle komfortabel aufrufen.

Der LINEEDIT bietet noch einige weitere zum Editor **UH–WORD** kompatible Tastaturkommandos zum Editieren einer Zeile.

Im Umgang mit älteren Eingaben stehen folgende Kommandos zur Verfügung:

Taste	Funktion	Erklärung
↑	one line up	eine Zeile zurück (ältere Zeilen), eventuell suchen
↓	one line down	eine Zeile vorwärts (neuere Zeilen), evtl. suchen
^Y	delete line	Zeile aus History-Buffer löschen
^XR;	to first entry	zur ältesten Zeile im History-Buffer springen
^XC;	to last entry	ans Ende des History-Buffer springen, Eingabezeile löschen
^O	push line to Buf.	Vom Nutzer eingegebene Zeile in den History-Buffer schreiben (ist sinnvoll, wenn man nach der Eingabe eines langen Befehl merkt, das doch zunächst noch ein anderer Befehl auszuführen ist)

Zur Bearbeitung einer Zeile kennt der LINEEDIT zusätzlich folgende Kommandos:

Taste	Funktion	Erklärung
CR	exit	Eingabe beenden
EOT	exit	Eingabe beenden [a]
ESC CR	truncate and exit	Eingabezeile hinter dem Cursor löschen und und Zeile an Task schicken, die die Eingabe erwartet
(backspace)	backspace	Zeichen links vom Cursor löschen
←	cursor left	Cursor ein Zeichen nach links
→	cursor right	Cursor ein Zeichen nach rechts
^N	clear line	Eingabezeile löschen
DEL	delete char	Zeichen unter dem Cursor löschen
INS	insert char	Zeichen einfügen
^XD	cursor to linebegin	Cursor zum Zeilenanfang
^XS	cursor to lineend	Cursor zum Zeilenende
^XY	truncate line	Eingabezeile hinter dem Cursor löschen
^T	clear end of word	Wort rechts vom Cursor löschen
^F	cursor word right	Cursor ein Wort nach rechts
^A	cursor word left	Cursor ein Wort nach links

[a] beenden, falls Autostop für EOT aktiv

Parameter: Sofern *device* angegeben ist, wird der LINEEDIT für das angegebene Gerät installiert, konfiguriert oder deinstalliert. Ist dieser Parameter nicht angegeben, so wird das `Stdin`-Device der dem Nutzer zugeordneten Shell verwendet.

Optionen: Bei einigen Optionen kann direkt hinter dem Optionsbuchstaben optional eine `1` bzw. `0` folgen. Die `1` bedeutet Aktivierung und die `0` Deaktivierung der Option. Diese Optionen sind standardmäßig inaktiv.

`-?` Anzeige aller Parameter und Optionen (Online-Hilfe).

`-X` LINEEDIT deinstallieren.

`-B[|0|1]` Diese Option legt fest, ob das Zeichen Backspace ($08) als Backspace oder als Cursor links zu interpretieren ist (`-B` → als Cursor links interpretieren).

`-I[|0|1]` Einfüge-Mode aktivieren.

`-O[|0|1]` Klingel bei Fehleingaben deaktivieren.

`-P[|0|1]` Normalerweise bearbeitet der LINEEDIT alle Eingabe-CEs. Bei aktiver `P`-Option werden jedoch die CEs, deren Pfad nicht mit dem Installationspfad des LINEEDIT übereinstimmen, direkt an die I/O-Task geschickt.

`-S=xxx` Diese Option legt die Größe des History-Buffers fest. Sie darf nur bei der Installation des LINEEDIT angegeben werden. Fehlt sie, so hat der History-Buffer eine Größe von 1KByte. Als Wert sind nur Potenzen von zwei zugelassen. Die Eingabe erfolgt als Hexadezimalzahl.

`-D=xxx` Es ist i.a. nicht sinnvoll, sehr kurze Befehle (z.B. 'S' oder 'LU') abzuspeichern. Mit dieser Option läßt sich eine minimale Zeilenlänge angeben, ab der der LINEEDIT die Zeile in seinen Puffer aufnimmt. Kürzere Zeilen werden nicht im History-Buffer gespeichert. Standardmäßig speichert der LINEEDIT alle Zeilen.

`-F=xxx` Diese Option definiert eine Default-Datei zum Speichern und Lesen des History-Buffers.

`-R=xxx` Mithilfe dieser Option läßt sich eine Textdatei in den LINEEDIT einlesen. Soll aus der mittels `F`-Option definierten Default-Datei gelesen werden, so ist `-R=&` anzugeben.

`-W=xxx` Mithilfe dieser Option läßt sich der History-Buffer in eine Textdatei schreiben. Existiert die Datei bereits, so wird sie überschrieben. Soll in die mittels `F`-Option definierte Datei geschrieben werden, so ist `-W=&` anzugeben.

Sichtgerät: Eine spezielle Konfiguration des LINEEDIT für das verwendete Terminal oder den Windowmanager ist nicht erforderlich. Der LINEEDIT kennt die Steuerzeichen der meisten Terminals (Televideo, VT52,VT100, usw.). Zur Ansteuerung benutzt er nur die druckbaren Zeichen und das allgemein verfügbare Steuerzeichen Backspace ($08). Mehrzeilige Eingaben sind nur möglich, wenn das Sichtgerät einen Autowraparound durchführt.

Es folgt eine vollständige Aufstellung aller dem LINEEDIT bekannten Steuersequenzen:

exit	CR; $04(Autostop bei EOT aktiv);
truncate and exit	ESC CR;
backspace	$07; $08(Option B nicht aktiv);
cursor left	$08(Option B aktiv); ESC D; ESC [D;
cursor right	$0C; ESC C; ESC [C;
delete char	$7F; ESC $08; ESC P; ESC W; ESC ESC D; ESC ESC [D;
insert char	ESC $0C; ESC Q; ESC ESC C; ESC ESC [C;
cursor to linebegin	^XD; ^QD;
cursor to lineend	^XS; ^QS;
clear line	^N; ESC $0A; ESC ESC B; ESC ESC [B;
truncate line	^XY; ^QY;

clear end of word	^T;
cursor word right	^F;
cursor word left	^A;
toggle B-Option	^B;
toggle I-Option	^I; ^_;
one line up	0B; Esc A; Esc [A;
one line down	0A; Esc B; Esc [B; 16;
delete line	^Y Esc 0B; Esc Esc A; Esc Esc [A;
to first entry	^XR; ^QR;
to last entry	^XC; ^QC;
push line to Buf.	^O;

Einbindung in **RTOS–UH**: Der Bedienbefehl `LE` bzw. `LINEEDIT` erzeugt eine Task, die sich vor die Betreuungstask der Eingabeschnittstelle setzt. Diese Filtertask empfängt von **RTOS–UH** alle CEs für die jeweilige Schnittstelle. Alle CEs, die keine Eingabe erwarten, schickt sie direkt weiter an die I/O-Task. Nur Eingabe-CEs ohne binären Transfer mit einer Datenlänge (RECLEN) größer eins, eingeschaltetem Echo und aktivem Autostop bei Cr bearbeitet die Filtertask weiter.

Beispiele:

```
LE  -D=4 -IB -F=/HO/AUTO/MYCOMAND -R=&
```

Für die eigene Eingabeschnittstelle wird der Zeileneditor Lineedit eingerichtet. Alle Eingaben, die kürzer als vier Zeichen sind, werden nicht abgespeichert. Der Lineedit wird im Einfüge-Modus betrieben, das Zeichen $08 wird als Cursor links interpretiert. Die Datei `/HO/AUTO/MYCOMAND` ist standardmäßig zum Speichern und Lesen des History-Buffers zu erwenden. Abschließend werden die in der Datei `/HO/AUTO/MYCOMAND` (`-R=&`) gespeicherten Befehle eingelesen. Dies ist ein typischer Aufruf, wie er in der Startup-Datei eines jeden Benutzers stehen könnte.

```
LE -W=& -X
```

So könnte die letzte Aktivität vor dem Ausschalten des Rechners aussehen: Der Inhalt des History-Buffers wird in die voreingestellte Datei gesichert und der Lineedit deinstalliert.

L I B S E T — Library einrichten

SYNTAX: `LIBSET [[+][file1+file2+...] | [-R [filex]]]`

Beschreibung: Die typische Anweisung sieht wie folgt aus:

`LIBSET` *+file1+file2+ ...*

Sie legt eine Library mit den Bezügen zu den Dateien *file1*, *file2* etc. an, in denen globale Symbole bereits geladener S-Records abgelegt sind, wenn beim Laden derselben die Files *file1* und *file2* mit Hilfe der „Code-Output"-Option des Laders erzeugt wurden, d.h. dem Lader als `CO`-Parameter übergeben wurden.
Zum Beispiel:

`LOAD SREC1 CO` *file1*

`LOAD SREC2 >` *file2*

`CO` und `>` sind wie üblich gleichwertig. Der Vorteil dieser Library ist, daß beim Laden eines S-Records, in dem Querbezüge durch globale Symbole zu den S-Records `SREC1`, `SREC2` ... bestehen, nicht immer wieder alle S-Records in der Ladeliste anzugeben sind. Stellt der Lader beim Laden eines S-Records offene Querbezüge fest, wird die Library nach diesen durchsucht. Bei der Programmentwicklung ist also immer nur noch das sich gerade in Bearbeitung befindliche Modul zu laden und zu entladen.

Die Anweisung

`LIBSET -R`*filex*

entfernt den Bezug auf die Datei *filex* aus der Library. Wird *filex* nicht angegeben, werden alle Bezüge der Library gelöscht.

! → Entladen Sie kein Modul, wenn noch der Eintrag der in diesem Modul vorhandenen globalen Symbole in der Library besteht! Dieser ist mit der Option „`-R`" zuvor zu entfernen. Werden die

Dateien mit den Symboladressen gelöscht, ohne den Bezug in der Library zu entfernen, meldet das System beim Laden mit offenen Bezügen das Nichtvorhandensein dieser Datei, und der Ladevorgang wird abgebrochen.

Die Anweisung

```
LIBSET
```

ohne Parameter listet alle Dateien auf, die in der Library enthalten sind.

Beispiel:	`LIBSET +ACO+BCO`	Fügt `ACO` und `BCO` der Libery hinzu.
	`LIBSET ACO+BCO`	Setzt die Libery auf `ACO` und `BCO`

L I N K — Link Filenames

SYNTAX: `LINK` *filename* `>` *newfilename*

Beschreibung: Mit dieser Anweisung wird ein alternativer neuer Filename (*newfilename*) in das Fileverzeichnis eingetragen, in dem sich das existierende File *filename* bereits befindet.

Beispiel: Mit der Anweisung:

```
LINK /H0/ABCD/GROSS > gross;
```

erreicht man, daß der Inhalt des Files `/H0/ABCD/GROSS` zukünftig auch über den Zugriffspfad `/H0/ABCD/gross` erreichbar ist. Dabei werden keine Inhalte kopiert, sondern es wird in das Directory (Hier:`/H0/ABCD/`) nur ein weiterer Zeiger unter dem neuen Namen eingerichtet. Beim `FIND`-Befehl erkennt man dies daran, daß „gelinkte“ Files denselben Startblock besitzen.

Besonderheit: Weil auch der dazugelinkte Filename einen vollständigen Verwaltungsblock erhält, kann auf diese Weise gleichzeitiges multiples Lesen ein- und derselben Datei durch mehrere Tasks ermöglicht werden. (Natürlich darf nicht gleichzeitig irgendein Schreiber den File benutzen!)

! → Gelinkte Files unterliegen gewissen Restriktionen, die vom jeweiligen File-Handler abhängen. Im MSDOS-kompatiblen Disc-Filehandler ist die Anwendung von `LINK` nicht möglich. Allgemein gilt, daß man vor der Entfernung des Files mit Hilfe des `RM`–Befehles zunächst die dazugelinkten alternativen Zugriffsnamen beseitigen muß. Links bei RTOS-formatierten Disketten und Festplatten sind an der Dateilänge 0 erkennbar.

Link S-Records (optionaler Bedienbefehl) **L N K**

SYNTAX: `LNK`

`LNK.`*`sonprocname`* `[PRIO` *`integer3`*`]` `[`*`linkspeclist`*`]`

`LNK [`*`integer3`*`]` `[`*`linkspeclist`*`]`

Beschreibung: Es wird ein Prozeß zum Linken mehrerer S-Records zu einem einzigen Ladefile generiert. Der Name kann entweder durch *sonprocname* vorgegeben werden, oder er wird mit `LNK/`*`xx`* vom System gewählt. In beiden Fällen kann die Priorität dieser Linker–Task durch eine max. 3–stellige Ganzzahl vorgegeben werden. Bei nicht angegebener Priorität wird ein Wert von 20 eingesetzt.

Fehlt der Zusatz *linkspeclist*, so wird ein Linkvorgang mit den Defaultwerten des Aufrufers für `SI`, `CO` und `LO` eingeleitet.

linkspeclist ist eine Liste von Geräte/File–Namen und eine evtl. Arbeitsspeichergrößenangabe. Die Elemente dieser Liste werden durch Leerzeichen oder Kommata getrennt.

Programmgröße: `SZ` *`hexnum6`* oder `SZ=`*`hexnum6`*

Mit *hexnum6* wird die Größe des verfügbaren Arbeitsspeichers vorgegeben. Die neueren Linkerversionen kennen 2 Betriebsarten: Der *Small*-Mode ist der herkömmliche Linkermode. Dabei ist der Speicherraum für die globalen Symbole merkbar begrenzt. Oberhalb von `SZ=1D000` schaltet der Linker in den *Large*-Mode, bei dem die globalen Symbole durch ein Hash-Verfahren im nun größeren Speicher effizienter abgelegt werden. Der maximal ausnutzbare Wert ist `SZ=FE0000` und ermöglicht praktisch unbegrenzt viele globale Symbole.

Geräte/Filenamen: Es werden die Parameter `SI` (S–Rekord–Input), `CO` (Code–Output) und `LO` (List–Output) akzeptiert.

Linkerbefehle: Die im folgenden erläuterten Linkeranweisungen können direkt in die zu linkenden S-Records eingefügt werden. Komfortabler und übersichtlicher ist aber die Verwendung von Steuerdateien, die dem Linker als Source-Input anzugeben sind und in denen die erforderlichen Anweisungen aufgeführt sind.

`#WD` *`Pathlist;`* Vereinbarung eines Working-Directory für folgende `INCLUDE`-Anweisungen. Bei folgenden `INCLUDE`-Anweisungen wird das

„$“–Zeichen durch den unter Pathlist angegebenen String ersetzt.

`#WDIR` *Pathlist*; Identisch mit `#WD`.

`#INCLUDE` *filespecifier*; Bei der Bearbeitung eines `INCLUDE` setzt der Linker seine Arbeit mit der Bearbeitung des angegebenen Files fort und kehrt nach der Bearbeitung dieses Files an die Stelle hinter der Anweisung zurück. Der *filespecifier* muß auf dem aktuellen System ein gültiges File selektieren. Wird nur ein Filename angegeben, wird das aktuelle Working-Directory des Users nach diesem File durchsucht.

Innerhalb der Include-Datei gilt zunächst das beim Aufruf gültige „WD“, es kann aber dort auch ein lokales „WD“ vereinbart werden, das dann nur in diesem File und eventuell tieferen Include-Leveln gilt. Nach der Rückkehr aus einem Includefile ist das beim Aufruf gültige „WD“ wieder gesetzt.

`#INCLUDE $`*file*; Das „$“-Zeichen wird durch den unter „WD“ angegebenen String ersetzt. Sonst wie oben.

`MODNAME` *Name*; Bei Verwendung dieser Anweisung erhält das Gesamtmodul einen vorgeschalteten Modulkopf mit dem angegebenen Namen. Dieser darf maximal 6 Zeichen lang sein. Fehlt die `MODNAME`-Anweisung, erhält das Gesamtmodul den Namen des ersten Moduls, auf das der Linker trifft. Soll das Gesamtmodul einen anderen Namen erhalten, der länger als 6 Zeichen lang ist, kann ein entsprechendes Modul mit dem PEARL-Compiler erzeugt und dem Linker als erstes File angeboten werden.

Sinnvoll ist die `MODNAME`-Anweisung insbesondere dann, wenn bei der ROM–Code–Erzeugung das Gesamtmodul erneut in den Speicher geladen werden soll, um z. B. den `DUMP`-Befehl zu verwenden. Die Verschiebung der Adresse durch den vorgeschalteten Modulkopf wird dann vom Linker automatisch berücksichtigt (siehe vom Linker ausgegebene Speicherliste).

`CONDLNK;` „Conditional Linkmode On“ für Library Linking. „Im Conditional Linkmode“ wird ein eingegebenes `S`-Record-File nur dann ins Ausgabefile gelinkt, wenn in diesem File offene globale Referenzen vorher bearbeiteter Files definiert sind (Bibliotheksfunktion), ansonsten wird das Eingabefile ignoriert.

`UNCONDLNK;` „Conditional Linkmode Off“.

`ROMCODE;` Einschalten der ROM-Code-Erzeugung. Alle einzugebenden

(PEARL-) Files müssen mit eingeschalteter „CODE=“ und „VAR=“ Option des Compilers übersetzt sein. Der Linker erzeugt als Ausgabe einen S–Recordfile, der nur hexadezimale Werte enthält, zwar ladbar aber nicht ausführbar ist, weil er nur auf den angegebenen ROM– und RAM–Adressen im Zielrechner lauffähig ist.

`ROMCODE+;` Einschalten der ROM-Code-Erzeugung wie oben. An die Ausgabedatei wird nun jedoch ein zusätzlicher – mit `S0` und `S9` eingerahmter Block – angehängt, der alle nicht versteckten globalen Symbole exportiert. Damit kann man nachträglich an E-Prom-residente Programme noch weitere Module anlinken. Siehe dazu auch das `HIDE`- und `UNHIDE`-Kommando.

`CODE= ...` Durch diese Anweisungen lassen sich die Codeadresse und die Variablenadresse bei der ROM-Code-Erzeugung bliebig vorgeben. Damit können die bei der PEARL-Compilation gemachten Vereinbarungen völlig legal übersteuert werden. Ist das Schlüsselwort `ROMCODE` nicht vorhanden, werden die Eingabefiles vorgelinkt, d. h. alle lokalen Label innerhalb der `S`-Records entfernt und auf die in der Linker-Speicherliste angegebenen Adressen verschoben, um dann z. B. erneut gelinkt oder mittels des `PROM`-Befehls bearbeitet zu werden.

`VAR= ...`

`HIDE;` Durch diese Anweisungen lassen sich die in den folgenden einzulesenden S-Rekords definierten globalen Symbole verstecken bzw. wieder aufdecken. Versteckte globale Symbole werden im erzeugten S-Rekord normal benutzt aber nicht global exportiert. Sie sind in der ggf. ausgegebenen Liste durch ein der hexadezimalen Adresse nachgestelltes `i` (für *internal*) zu erkennen. Standardmäßig befindet sich der Linker im `UNHIDE`-Mode.

`UNHIDE;`

`DEVICE` *Name Hex4*; Durch dieses Kommando wird eine `DATION` mit der Bezeichnung *Name* definiert. Damit können die vom Compiler als Extra-Devices adressierten Geräte des Targetsystemes eingebunden werden. Die vierstellige Hexadezimalzahl *Hex4* ist die Device– und Drive–Nummer (z. B. 0302 für /H0).

Bei der „WD“- und der `INCLUDE`-Anweisung kann das vorstehende `#`–Zeichen auch weggelassen werden.

Beispiele: Die Steuerdatei (z. B. `/H0/LK/LINKES`):

```
! Mit dem Ausrufezeichen beginnen Kommentare.
WD /HO/REGELUNG/
MODNAME REGLER
INCLUDE $MESSENSR
INCLUDE $PIDSR
INCLUDE $GRAFIKSR
HIDE ! Grafik-package verstecken
INCLUDE /HO/GRAFIK/GRAFDRV
UNHIDE ! Globale Symbole der folgenden Dateien exportieren.
INCLUDE AUSGABSR
INCLUDE ../TESTSR
```

Durch die Anweisung

```
LNK SI /HO/LK/LINKES CO /HO/PIDREGSR;
```

erzeugt der Linker ein Modul mit dem Namen „`REGLER`“. Abgelegt wird es in der Datei „`/HO/PIDREGSR`“. Die effektiven Zugriffspfade für die sechs gelinkten Teilmodule ergeben sich in diesem Fall wie folgt:

```
/HO/REGELUNG/MESSENSR    /HO/REGELUNG/PIDSR
/HO/REGELUNG/GRAFIKSR    /HO/GRAFIK/GRAFDRV
/HO/LK/AUSGABSR          /HO/TESTSR
```

Fatale Fehler, die zum Abbruch führen:

`NO SRECORD FILE`
das bearbeitete File ist kein S-Record oder Steuerfile.

`CHECKSUM ERROR (FATAL)`
die Prüfsumme eines S-Records ist falsch.

`WRONG LINKER INPUT`
es liegt ein gravierender Fehler in dem S-Record vor.

`MISSING /S0/S9/DATAREC`
die Struktur eines S-Records ist falsch.

`ODD NO (FATAL)`
globales Symbol mit ungerader Adresse oder Kopf eines Files steht auf ungerader Adresse.

`TABLE ERROR (FATAL)`
Fehler bei Organisation der Listen des Linkers (sollte bei S-Records fehlerfrei übersetzter Sources nicht auftreten).

`FILE READ ERROR`
Fehler beim Zugriff auf ein File.

`HIGHNIBBLE INCONST ERROR`
Overflow über einen 3 Byte Wert mit Vorzeichen.

`LINKER PASS 1 ERROR TERMINATED`
leichter Fehler in Pass 1 (Pass 2 wird nicht gestartet).

`NO SKEW IN FILE NO: 0`
Linker wurde erst nach vollständiger Abarbeitung von Files in den romable Mode geschaltet.

`SYNTAX ERROR (CONTINUE)`
Fehler bei Bearbeitung einer Linkeranweisung, führt zum Abbruch nach Pass 1.

Überbrückbare Fehler sowie Warnungen des Linkers:

`WARNING: RTOS - SHELLEXTENSIONS LOST`
ein Modul wird überzeigert, das RTOS-Shellextensions enthält, die später vom System (nur beim Laden in das RAM!) nicht mehr gefunden werden können. Tritt auf, wenn für ein Shellmodul `MODNAME` verwendet wird oder ein Shellmodul nicht das erste File der Includeliste ist.

`*DOUBLE* : ...`
eine doppelte Definition eines Symbols wird ignoriert.

`UNDEF SYMBOL: ... AT ADRS: ...`
bei der ROM-Code-Erzeugung wurde ein Symbol „requested“, das nicht definiert ist; der Fehler kann nachträglich nicht vom Linker korrigiert werden, die Benutzung der erzeugten Codes ist riskant.

`WARNING : RTOS - FILEHEAD MISSED`
im erzeugten Output File konnte kein gültiger RTOS-Filehead gefunden werden; Laden des Files ist riskant.

`LINKER PASS 1 OVERFLOW ENTER P2`
lokaler Overflow; Linker beschäftigt sich in diesem Linklauf nur noch mit lokalen Symbolen und bindet die Files nicht.

`IGNORING FILE WITH ABSOLUT ADRESSDEFINITIONS`
ein Länge-Null File, das absolute Adressdefinitionen enthält, wird vom Linker im non-romable Mode nicht bearbeitet, um keinen rechnerabhängigen Code zu erzeugen.

`WARNING: NO SKEW IN FILE NO: ...`
im romable Mode wurde ein File gefunden, das keine Skews enthielt. Skews plaziert der Compiler, um den Versatz zwischen logischer und physikalischer Adresse zu fixieren. Höchstwahrscheinlich wurde beim Übersetzerlauf das Einschalten der Compiler–Option für ROM-Code vergessen.

`LINKER COMMAND ERROR`
Die `MODNAME`-Anweisung wurde im ROM-Code-Mode verwendet, obwohl die Adressverwaltung nicht vom Linker durchgeführt werden soll.

Load and Link Programm **L O A D**

SYNTAX: `LOAD`
`LOAD.`*`sonprocname`* `[PRIO` *`integer3`*`] [`*`loadspeclist`*`]`
`LOAD [PRIO` *`integer3`*`] [`*`loadspeclist`*`]`

Beschreibung: Es wird ein Sohnprozeß generiert, dessen Name entweder durch *sonprocname* vorgegeben wird oder mit Namen `LOAD/`*`xx`* vom System gewählt werden soll. In beiden Fällen kann die Priorität dieser Lader–Task durch eine max. 3–stellige Ganzzahl vorgegeben werden. Bei nicht angegebener Priorität wird ein Wert von 20 eingesetzt.

Fehlt der Zusatz *loadspeclist*, so wird ein Ladevorgang vom Standard–Ladefile des Systems und des Nutzers eingeleitet. Es wird auf den von unten nach oben gesuchten ersten freien passenden Speicherbereich geladen.

Bleiben noch globale Symbole offen, so durchsucht der Lader den Scan–Bereich des Systems nach 17er–Scheiben, um Referenzen ggf. von dort zu befriedigen. Sind dann immer noch globale Bezüge offen, wird der Lader mit einer entsprechenden Meldung suspendiert. Soll vorher auch das RAM noch nach 17–er–Scheiben durchsucht werden, so ist der `LOADX`-Befehl (Seite 173) zu verwenden.

Der Ladevorgang kann durch die Angabe der *loadspeclist* parametriert bzw. als bindendes Laden formuliert werden.

loadspeclist ist eine Liste von Geräte/File–Namen und Adreß– oder Größenangaben. Die Elemente dieser Liste werden durch Leerzeichen oder Kommata getrennt.

Adreßangabe: `AD` *`hexnum8`* oder `AD=`*`hexnum8`*

Dabei steht *hexnum8* für die maximal 8–stellige Hexadezimalzahl, bei der der Lade/Bindevorgang beginnen soll. Erst der hier generierte Lader prüft später, ob der Adreßbereich überhaupt zum Laden verfügbar ist. Diese Adreßangabe ist außer für Testzwecke (glatte Adresse) aus der Sicht von **RTOS–UH** ein unerwünschter Eingriff.

Programmgröße: `SZ` *`hexnum6`* oder `SZ=`*`hexnum6`*

Mit *hexnum6* wird die Größe des ersten zu ladenden Modules vorgegeben und damit die Größenangabe — falls vorhanden —

im Ladereingabetext übersteuert. Dies darf natürlich nur so geschehen, daß der SZ–Wert größer als die tatsächliche Modulgröße ist.

Geräte/Filenamen: Es werden die Parameter SI (S–Rekord–Input), SC (Source after Continuation), LO (List–Output) und CO (Code Output) akzeptiert.

LO

Wenn LO nicht angegeben wird, so werden nur fehlende Globalsymbole auf dem Defaultgerät aufgelistet. LO /ED/LABEL z. B. bewirkt, daß beim Ladeschluß die Zuordnungstabelle Symbol/Adresse in die Edit–Datei LABEL geschrieben wird. Dieser forcierte List-Output enthält auch jene Symbole, die der Lader aus dem eigenen System-Eprom (bzw. dem in das RAM gebooteten Gesamtsystem) als 17er–Scheiben gefunden hat.

SI

Mit Hilfe des Parameters SI kann eine Liste von Modulquellen angegeben werden. Die einzelnen Quellen werden durch das Zeichen + getrennt. Soweit Querbezüge durch globale Symbole zwischen den einzelnen Modulen existieren, werden sie vom Lader durch einen integrierten Bindevorgang realisiert.

```
LOAD SI=/ED/LIB1+/ED/LIB2 (SI= kann entfallen)
```

SC

Der Parameter SC (Source after Continuation) dient zur Angabe einer ggf. benutzten „Reservedatei“, die der Lader immer wieder als Quelle zur Fortsetzung des Bindevorgangs anläuft, solange noch unbefriedigte Globalreferenzen existieren. Nach der Meldung „*xyz* `suspended loader input`“ wird die Auffüllung dieser Datei bzw. der Defaultdatei /ED/LB bei fehlendem SC erwartet und anschließend ein CONTINUE–Kommando für den Lader–Sohnprozeß. Will man die Referenzen nicht nachreichen, weil man absolut sicher ist, daß sie nicht benutzt werden, so sollte aus Platzgründen der Lader–Sohnprozeß mit UNLOAD eliminiert werden.

```
LOAD.X PRIO 5 AD 6000 SZ 2000 /B2+/B2+/F0/QUELLE
```

Name des Sohnprozesses ist X, Priorität des Ladevorgangs ist 5. Das erste von Port 2 kommende Modul wird ab Adresse 6000 geladen und auf die Größe von 2000 vergrößert. An dieses Modul wird das nächste von Port 2 stammende Modul angebunden. Die Ablageadresse dieses Moduls ist nicht bekannt. Das gleiche gilt für das dritte von Floppy–Laufwerk 0, File QUELLE stammende Modul, welches ebenfalls hinsichtlich der globalen Symbole angebunden wird.

```
LOAD SZ 5000 /F1/TEST LO /A1 SC /B2
```

Der File `TEST` von Floppy Laufwerk 1 wird geladen und die Liste der globalen Symbole samt Adressen über Port 1 ausgegeben. Der Lader hat den Namen `LOAD/xx` und läuft mit Priorität `20`. Unbefriedigte Globalreferenzen können, falls vorhanden, über das Port 2 ergänzt werden.

CO

! →

Mit Hilfe eines optionalen `CO`-Parameters kann der Lader einen S-Rekord-File ausgeben, der als sogenanntes „Null-size-Modul“ linkbare Adressinformation aller beim Laden abgelegten globalen Symbole enthält. Symbole aus den eigenen 17-er–Scheiben werden jedoch nicht in den `CO`-File geschrieben. Der so erzeugte File kann zusammen mit dem `LIBSET`-Befehl eine enorme Verkürzung der Turnaround-Zeit bei Multi-Modul-Bearbeitung bewirken. Näheres siehe `LIBSET` auf Seite 160.

```
LOAD SI=PR1+PR2+PR3 CO=lkfile123
```

legt ein Null-size-Modul mit Filenamen `lkfile123` unter dem aktuellen Working-Directory an. Das könnte ein Unterprogrammpaket sein, daß man in Zukunft öfter benutzten möchte, ohne daß jedes Mal der Programmcode geladen werden muß.

Fehlermeldungen: Beim Laden von Dateien, die durch den PEARL–Compiler erzeugt wurden, können bei fehlenden Programmmarken oder anderen Fehlern unbefriedigte Vorwärtsbezüge zurückbleiben, die von der Ladertask moniert werden. Daher dürfen nur als fehlerfrei vom Compiler ausgewiesene Module geladen werden! Doppelt definierte Globalsymbole werden während des Ladevorganges aufgelistet.

Zeichen, mit denen der Lader nichts anfangen kann, führen zum Abbruch des Ladevorgangs mit der Meldung „`wrong loader input`“.

Werden mehrere Module gleichzeitig geladen, muß die Shell bei relativen Pfadnamen das Working-Directory voranstellen. Reicht der Expansionspuffer der (implementationsabhängigen) Shell nicht aus, erfolgt die Fehlermeldung `capacity overflow`. Der Lader läuft in diesem Fall gar nicht erst an.

Ist der Abstand zwischen Aufruf einer PC-relativen Adresse und ihrer Definition weiter als 32 KByte entfernt (z.B. bei der Verwendung des `CASE/ALT/FIN` Konstruktes), kommt es trotz fehlerfreier Übersetzung beim Laden zur Fehlermeldung `module`

`overflow label`.

Hinweise: Die Anzahl vorgebbarer Modulquellen ist nur durch die Puffergrenze der aufrufenden Shell und nicht durch den Lader selbst begrenzt.

Mit dem PEARL–Einphasencompiler ohne `S= ...` erzeugte Dateien enthalten in ihrem Kopf nicht die Programmgröße des Modules, sondern einen Pauschalwert (2000). Bei Programmen, in denen nur ein PEARL–codiertes Modul vorkommt, sollte dieses als erstes geladen werden und die tatsächliche Programmgröße (s. Compilerbilanz) durch `SZ` vorbesetzt werden.

Es kann eine nahezu unbegrenzte Zahl solcher `LOAD`–Kommandos abgesetzt werden, die im Multitasking parallel bearbeitet werden. Dabei dürfen diese Ladevorgänge keine gemeinsamen Quellfiles benutzen, da sonst die Eingabedatei entweder zerfleddert gelesen wird oder es aber wegen des exklusiven Lesens nur zum Überleben des ersten Ladeprozesses reicht.

Der von einem solchen Ladeprozeß gerade bearbeitete Speicherbereich wird während des Ladevorganges als Speichersektion vom Typ `PWSP` in der Verwaltung von **RTOS–UH** geführt. Wird der Laderprozeß terminiert (und zweckmäßigerweise auch mit `UNLOAD` entfernt), so verschwindet auch das zuletzt angefangene Modul aus der Verwaltung.

Linker: Werden dem Lader mehrere (S0–S9) `S`–Record–Blöcke in einer Datei angeboten, so werden diese beim Laden ebenso gelinkt, als wenn sie aus mehreren Dateien stammen würden, die mit `+` verbunden wurden. Damit ist es möglich, bei größeren Projekten die S-Records aller schon getesteten Module in einer Datei zu vereinigen und nur noch die neuen S-Records aus einer eigenen Datei zu laden. Allerdings ist hier zu prüfen, ob nicht das Vorlinken mit Hilfe des Linkerbefehles `LNK` günstiger ist. Lesen Sie dazu bitte auf den Seiten 163–168 nach.

Load extended **L O A D X**

SYNTAX: `LOADX`
`LOADX.`*`sonprocname`* `[PRIO` *`integer3`*`] [`*`loadspeclist`*`]`
`LOADX [PRIO` *`integer3`*`] [`*`loadspeclist`*`]`

Beschreibung: Es wird ein Sohnprozeß generiert, dessen Name entweder durch *sonprocname* vorgegeben wird oder mit Namen `LOADX/`*`xx`* vom System gewählt werden soll. In beiden Fällen kann die Priorität dieser Lader–Task durch eine max. 3–stellige Ganzzahl vorgegeben werden. Bei nicht angegebener Priorität wird ein Wert von 20 eingesetzt.

Der Befehl ist funktionell völlig identisch zum normalen `LOAD`, der ab Seite 169 genau beschrieben ist. Der einzige Unterschied besteht darin, daß bei unbefriedigten Bezügen auf globale Objekte noch in allen geladenen Modulen nach 17er Scheiben gesucht wird. Erst wenn auch das fehlschlägt, folgt die Selbstsuspendierung. Die 17er Scheiben kann ein Assemblerprogrammierer generiert haben, oder aber der PEARL-Compiler hat sie mit der `/*+G */`-Option innerhalb einer Prozedurdefinition erzeugt.

Sinn dieser erweiterten Ladeanweisung ist es, daß man sich ähnlich wie beim `LIBSET`-Befehl (Seite 160) eine private Unterprogrammbibliothek im RAM halten kann. Andererseits ist die Suche je nach Systemladezustand eventuell zeitaufwendig; aus diesem Grund, ist die Operation nicht als Standardoperation im normalen `LOAD` enthalten.

Man beachte, daß bei der Benutzung des `CO`-Parameters keine Einträge für globale Symbole aus den 17er Scheiben in die erzeugten S-Records geschrieben werden.

L U — List User Task

SYNTAX: `LU`

Beschreibung: Die Anweisung wirkt wie das `L`–Kommando, jedoch werden die beim Kaltstart des Systems bereits vorhandenen Systemtasks nicht mit aufgelistet. Es werden alle im System befindlichen User–Tasks gelistet, ohne Berücksichtigung der User-Nummer. Das Kommando ist eine Kurzform für `L -U`, näheres ab Seite 153.

Beispiel: `LU`

Make Directory **M K D I R**

SYNTAX: `MKDIR` *pathlist–list*

Beschreibung: Der Befehl `MKDIR` erlaubt die Einrichtung von Subdirectories und ermöglicht damit eine hierarchische Dateiverwaltung. Mit dem Befehl `RMDIR` lassen sich vereinbarte Directories wieder löschen.

pathlist–list: Es sind alle Geräte erlaubt, bei denen das entsprechende Bit für `MKDIR` im Device–Wort gesetzt ist (siehe `SD`–Befehl). Die einzelnen Elemente der Liste werden durch Kommata oder Zwischenräume getrennt.

pathlist: 1 bis 7 Buchstaben oder Ziffern pro Pfadelement sind erlaubt (unter MS-Verwaltung bis zu 8). Es gilt die übliche Syntax der Pathlist: die Pfadelemente sind durch „/“ zu trennen. Wird mit der MS–DOS kompatiblen Dateiverwaltung gearbeitet, so kann ein Element der Pathlist aus max. 8 Buchstaben oder Ziffern, gefolgt von einem Punkt und weiteren 3 Zeichen bestehen.

Beispiele: `MKDIR /F0/USER1`

Auf der Diskette in Laufwerk `/F0/` wird das Subdirectory `USER1` angelegt. Dort können jetzt Dateien abgelegt werden, z. B. mit

`COPY /ED/SI>/F0/USER1/DATEI1`

`MKDIR /F0/USER1/DATEN1`

In dem Subdirectory `USER1` wird ein weiteres Subdirectory `DATEN1` angelegt.

`MKDIR /F0/PROG.PRL`

Auf der Diskette im Laufwerk 0 wird das Subdirectory `PROG.PRL` angelegt.

Hinweise: Mit `DIR /F0/` bekommt man jetzt alle Dateien und Directories der Root–Ebene. Ein Directory ist am nachgestellten Schrägstrich zu erkennen. Mit `DIR /F0/USER1` erhält man die Dateien und weiteren Subdirectories des Directorys `USER1`.

M S F I L E S — DOS–filesystem definition

SYNTAX: `MSFILES` *`device, device, ...`*

Beschreibung: Das mit *device* bezeichnete Gerät — typischerweise eine Floppy oder ein Plattenspeicher — wird unter die Verwaltung des MS–DOS-Filemanagers gestellt. Wurde z. B. bisher die Floppy `/F0/` als **RTOS–UH**–Diskette behandelt, so wird nach Absetzen des Befehles `MSFILES /F0/` jetzt eine MS–DOS–Diskette im Laufwerk `F0` erwartet.

Beispiel: Eine Atari– oder MS–DOS–Diskette mit File `PA.TXT` soll auf eine **RTOS–UH**–Diskette kopiert werden, ungeachtet der evtl. späteren Nachbehandlung wegen anderer Sonderzeichen und Zeilenendekennung. Hinterher wird nur noch mit **RTOS–UH**–Disketten gearbeitet.

```
MSFILES /F1                 | unter MS–DOS–Verw.
RTOSFILES /F0               | unter RTOS–UH–Verw.
CP /F1/PA.TXT>/F0/PAPER     | Kopiere
RTOSFILES /F1               | Wieder RTOS–UH–Verw.
```

Hinweise: Ob Ihr Filesystem nach dem Einschalten unter der **RTOS–UH**- oder unter der MS–DOS-Verwaltung anläuft, hängt von Ihrer Implementierung ab. Mit den neueren Filemanagern ist das normalerweise nicht mehr wichtig, weil sie automatisch den jeweils anderen auf den Plan rufen, wenn sie beim Zugriff auf das Medium erkennen, daß das Medium unter der anderen Verwaltung angelegt ist. Soll das Medium neu formatiert werden, so muß logischerweise die gewünschte Verwaltung explizit eingestellt werden.

Wenn noch Files auf dem Gerät geöffnet sind, so wird der Befehl nach Meldung „`... directory active ...`“ zurückgewiesen. Man kann aber das „Vergessen“ des nicht mehr benötigten Filesystems genau wie beim `CF` (Change Floppy)–Befehl erzwingen. Dies erfolgt durch eine spezielle Pseudo–pathlist:

```
SYNC /F0/          (siehe SYNC, zum Retten)
MSFILES /F0/FORGET (Vergiß alte Floppy)
```

Welches Filesystem auf der Floppy gerade gültig ist, kann man jederzeit über den Befehl `FILES` (z. B. `FILES /F0/`) erfragen,

da bei dessen Ausgabe die Verwaltungsstruktur mit erscheint — auch dann wenn kein File geöffnet ist.

Wenn nicht für alle Laufwerke einer Warteschlange der gleiche Filemanager zuständig ist (wie im Bsp. oben), legt die Betreuungstask im Speicher eine Transfertabelle für die Zuordnung Laufwerk<->Filemanager an, die über das S-Kommando sichtbar ist, auch wenn kein File offen ist. Die Tabelle verschwindet wieder, sobald der zuständige Filemanager eine Verbindung herstellt und danach alle Verbindungen auf ihn selbst zeigen. Im obigen Bsp. ist der Block hinterher noch existent, weil das letzte RTOSFILES vom MS–DOS–Filemanager ausgeführt wird. Mit RTOSFILES /F1,/F1 als letztem Befehl i. o. Bsp. spart man folglich einige Bytes Speicher ein.

Hinweis: Neuere Treiber erledigen die Anpassung an das zuständige Format automatisch. Die Anweisung ist dennoch nötig, wenn eine Diskette neu formatiert werden soll! Beim FORM-Befehl würde sonst die zufällig letzte benutzte Diskettenverwaltungsform auf der neuen Diskette eingerichtet.

N O T R A C E — No Tracing for specified Task

SYNTAX: `NOTRACE` *taskname*`,`*taskname* `...`

Beschreibung: Die angegebene Task wird aus dem Trace–Mode entlassen (siehe `TRACE`). Dabei wird die Breakpointadresse gelöscht.

Befand sich die angegebene Task nicht im Trace–Mode, so ist die Anweisung ohne Wirkung.

Wie bei der `TRACE`–Anweisung wird der aktuelle Laufzustand der Task durch diese Anweisung nicht geändert.

Beispiele:
```
NOTRACE TEST XYZ

NOTRACE INIT
```

Output–device specification **O**

SYNTAX: `O` *pathlist*

Beschreibung: Die Ausgabe der Shell wird auf das angegebene Gerät bzw. in den angegebenen File umgeleitet. Die momentan gültige lokale Kopie von „`Stdout`“ wird verändert. Alle Ausgaben des ausführenden Shellprozesses, (außer Fehlermeldungen), die durch die folgenden Kommandos dieser Zeile veranlaßt werden, erfolgen auf dem angegebenen Gerät bzw. in den File.

pathlist: Es werden alle der Shell Ihres **RTOS–UH** bekannten Devices akzeptiert, z. B. `/A1/`, `/A2/`, `/ED/dir` usw. Als Trennsymbol zum nachfolgenden Kommando sind ein Semikolon und ein Leerzeichen erlaubt.

Beispiele:

```
O /A2/;L;S
```

Es werden die Taskliste und die Speicherbelegung über den Port 2 ausgegeben.

```
O /ED/TEST DIR /F0/;O /F1/X S
```

Directory von Laufwerk `/F0/` in die Datei `/ED/TEST` schreiben, danach Speicherbelegung auf Floppyfile `/F1/X`.

P / P E A R L — Compile PEARL–Programm

SYNTAX: `PEARL` oder `P`
`PEARL.`*`sonprocname`* `[PRIO` *`integer3`*`] [parameterlist ]`
`PEARL [PRIO` *`integer3`*`] [parameterlist]`

Beschreibung: Es wird ein unabhängiger Sohnprozeß generiert, dessen Name entweder durch *sonprocname* vorgegeben oder vom System mit `P/`*`xx`* bestimmt wird. *xx* ist eine zweistellige Hexzahl mit automatischer Weiterschaltung. Ebenso kann mit dem Kommando die Priorität der Bearbeitung durch eine 3–stellige Ganzzahl festgelegt werden. Fehlt die `PRIO`–Angabe, wird ein Standardwert von 20 eingesetzt. Fehlt die *parameterlist*, so wird ein Übersetzungslauf gestartet, der von dem Standard–Inputfile des Nutzers liest und auf dem entsprechenden Outputfile die S-Records ablegt. Ein Listing wird auf das entsprechende Terminal ausgegeben.

parameterlist: ist eine Liste von Geräte/Filenamen und einer Größenangabe. Die Elemente der Liste werden durch Leerzeichen oder Kommata getrennt.

Arbeitsspeicher: `SZ` *`hexnum6`* oder `SZ=`*`hexnum6`*

Mit *hexnum6* kann der dynamische Arbeitsspeicher des Compilers bestimmt werden. Der Mindestwert beträgt `500`, vom System wird ein Wert von `2800` (entspricht ca. 10 KByte) eingesetzt. Maximal ist ein Wert von `10100` sinnvoll. Die gewählte Speichergröße beeinflußt die Übersetzungsgeschwindigkeit praktisch nicht. Faustregel für sehr lange PEARL–Programme (mehr als 4000 Zeilen):

`SZ`=(Max. Zahl lebender P-Symbole)*14 + 2000

Geräte/File: Es werden die Parameter `SI` (Source Input), `LO` (List Output) und `CO` (Code Output) ausgewertet. Für fehlende Angaben werden die Default–Werte des Systems und Nutzers eingesetzt. Bei `CO` und `LO` ist zum Abschalten auch der Gerätebezeichner `NO` bzw. `/NO` zulässig. Für `LO` kann nebeneinander ein echter Geräte/File- Bezeichner **und** `LO=NO` angegeben sein: der Compiler wird dann die Kopfzeile, Fehlermeldungen, eventuelle lokale Teillistings und die Schlußbilanz zum angegebenen Gerät bzw. File senden. Zwischen den Parametern und den zugewiesenen Objekten setzt man typischerweise einen Zwischenraum. Auch das Zeichen „`=`“ ist möglich, dann ist allerdings kein Zwischenraum vor und hinter dem „`=`“ mehr zulässig.

Beispiele:

```
PEARL.X PRIO 40 SZ 5000 /F0/TEST>/F1/BCOD LO /A1/
```

Das Programm auf Floppylaufwerk 0 in der Datei `TEST` wird übersetzt. Der Code wird in die Datei `BCOD` auf Floppylaufwerk 1 geschrieben. Das Übersetzungsprotokoll wird über die Schnittstelle `/A1/` ausgegeben. Name des Sohnprozesses ist `X`.

```
P
```

Mit den Defaultwerten für `SI` (`/ED/SI` für USER1), `LO` (`/A1/` für `USER1`) und `CO` (`/ED/SR` für `USER1`) wird ein Compiler (Name des Sohnprozesses `P/`*xx*) gestartet.

```
P.PEARL /A2/>NO LO /ED/ERROR LO NO
```

Der Quelltext wird über Port `/A2/` erwartet, es wird kein Code erzeugt und die Ausgabe fehlerhafter Zeilen erfolgt in die Datei `/ED/ERROR`. Name des Sohnprozesses ist `PEARL`.

Hinweise: Der Compilercode ist wiedereintrittsfest, so daß beliebig viele — sofern Platz für den dynamischen Speicher ist — verschiedene Übersetzungsvorgänge gleichzeitig im Multitasking ablaufen können.

Fehlermeldungen, weitere Eigenschaften und Steuermöglichkeiten des PEARL–Compilers sind ab Seite 277 ausführlich beschrieben.

P E R — Permanent Error Redirect

SYNTAX: `PER` *pathlist*

Beschreibung: Als Standard-Error Datenstation (`Stderr`) der primären Shell, unter dessen Nutzer dieser Befehl zur Ausführung kommt, wird fortan die durch *pathlist* bezeichnete Datensenke verwendet. Die Wirksamkeit erfaßt nicht die Kommandos im Rest der Kommandozeile. Der Befehl ist im Gegensatz zum „`ER`“-Befehl mit gewissen Risiken verbunden: die Fehler der nächsten Befehlszeilen schreibt die Shell in die neue Datenstation – wenn es geht. Kann in die Station nicht geschrieben werden, entstehen neue Fehlermeldungen, die wiederum nicht geschrieben werden können etc.

Beispiel:
```
PER /HO/NIL;
```

Nach dieser Zeile bleiben einem fortan alle Fehlermeldungen erspart! Zu empfehlen ist das natürlich nicht.

Man beachte, daß die Shell vor dem Hineinschreiben in die Datenstation „`Stderr`“ den File nicht öffnet, das macht der Handler der Datenstation notfalls automatisch. Auch wird der File am Ende nicht geschlossen. Auf diese Weise ist das akkumulierende Sammeln von Fehlermeldungen in einem File möglich, man muß allerdings dafür Sorge tragen, daß der File irgendwann geschlossen wird oder häufig genug `SYNC`-Befehle einstreuen.

! → Eine wichtige Bedeutung hat der `PER`-Befehl bei unbedienten Systemen: eine besondere Task kann sich um Unregelmäßigkeiten im System kümmern, diese ggf. auch archivieren. Die Task kann dabei am Ende einer Pipe (Station `/VI` bzw. `/VO`) sprungbereit alle Fehlertexte entgegennehmen.

Im Gegensatz zum `ER`-Befehl werden hier alle Fehlermeldungen, die dem Nutzer zugeordnet sind, umgelenkt. Auch die Ausgabe von Laufzeitfehlern irgendwelcher Tasks des Nutzers erfolgt auf das vereinbarte Gerät. Wird der Befehl von der Console gegeben (User No.1), so werden auch Irregularitäten bei Interrupts etc., die keinem Nutzer zugeordnet sind, umgelenkt.

Permanent Input–device specification **P I**

SYNTAX: `PI` *pathlist*

Beschreibung: Als Standard-Input (`Stdin`) der Shell, die diesen Befehl ausführt, wird fortan die durch *pathlist* bezeichnete Datenquelle verwendet. Die neue Vereinbarung gilt ab der nächsten Kommandozeile.

Warnung:

> Der Befehl ist im Gegensatz zum „`I`"-Befehl sehr riskant: man kann damit die primäre Shell seines Arbeitsplatzes irreparabel unbrauchbar machen! Wird der Input auf eine Station wegdirigiert, die man nicht unter Kontrolle hat, so hilft auch kein Systemabort aus der Klemme!

Beispiel:
```
PI /WINA0/SHELL1;
```

Der Befehl ist für solche Systeme gedacht, bei denen eine primäre Shell in ein Fenster umgelegt wird, oder bei denen im Hochlauf dynamisch Nutzerarbeitsplätze (primäre Shells) entstehen. „`PI`" sendet zusätzlich einen speziellen I/O-Befehl ab, der auch den „Ruf" der Shell mit `Ctrl A` auf das neue Eingabegerät legt. (Nur die in der Praxis in Frage kommenden I/O-Dämonen verstehen diesen Befehl!).

P O — Permanent Output Redirect

SYNTAX: `PO` *pathlist*

Beschreibung: Als Standard-Output Datenstation (`Stdout`) der primären Shell, unter dessen Nutzer der Befehl zur Ausführung kommt, wird fortan die durch *pathlist* bezeichnete Datensenke verwendet. Die Wirksamkeit erfaßt jedoch nicht die Kommandos im Rest der Kommandozeile. Der Befehl ist im Gegensatz zum „`O`"-Befehl mit gewissen Risiken verbunden: die Ausgabetexte der Bedienbefehle aus den nächsten Befehlszeilen schreibt die Shell in die neue Datenstation – wenn es geht. Kann in die Station nicht geschrieben werden, entstehen Fehlermeldungen auf dem Error-Kanal.

Beispiel:

```
PO /NIL;
```

Nach dieser Zeile bleiben einem fortan alle regulären Textausgaben der Shell erspart. Sinn macht das `PO`-Kommando nur in Ausnahmefällen, etwa bei der Einrichtung eines Nutzerarbeitsplatzes in einem Fenster. Im allgemeinen ist man mit dem `O`-Befehl besser bedient.

Man beachte, daß die Shell vor dem Hineinschreiben in die Datenstation „`Stdout`" den File nicht öffnet, das macht der Handler der Datenstation notfalls automatisch. Auch wird der File am Ende nicht geschlossen. Auf diese Weise ist das akkumulierende Sammeln von Ausgabetext in einem File möglich, man muß allerdings dafür Sorge tragen, daß der File irgendwann geschlossen wird oder häufig genug `SYNC`-Befehle einstreuen.

Prevent activation of Task **P R E V E N T**

SYNTAX: `PREVENT` *`taskname-list`*

Beschreibung: Alle Einplanungen auf Zeitpunkte oder Interrupts der einzelnen Tasks werden sofort gelöscht. Außerdem werden eventuell bereits im Aktivierungspuffer stehende Neuaktivierungen durch Ausräumen des Puffers verhindert. Die Tasknamen können durch Kommata oder durch leere Zwischenräume getrennt werden.

Beispiele:

```
PREVENT ABCD,TT

PREVENT init

prevent overflowtask reglerprozess
```

Hinweis: Trotz gelöschter Zeitplanung bleibt die Aufmerksamkeit der Planungsuhr für den in Aussicht genommenen Zeitpunkt erhalten. Bei `CLOCK` wird also unter `NEXT SCHED` eventuell der Termin noch aufgeführt, bleibt aber wirkungslos.

P R O M — Prepare for Read Only Memory

SYNTAX: `PROM` *name*
`PROM` *name**

Beschreibung: Wenn kein Linker (siehe Seiten 163 ff.) zur Verfügung steht oder aus irgendwelchen Gründen nicht eingesetzt werden soll, kann mit Hilfe des `PROM`-Befehles alternativ auch durch die Shell ein S-Rekord-Paket für ROM-residente Anwenderprogramme erzeugt werden.

Aus in den RAM–Bereich geladenen Programmelementen (Modul oder Task) werden Scheibendaten für das **RTOS–UH**–Autolink, die Systemkonfigurierung in der Kaltstartphase, erzeugt. Diese Scheibendaten werden als S-Records auf das Ausgabemedium der aufrufenden Shell („`Stdout`“) ausgegeben; für jeden zu „prommenden“ Speicherbereich werden ein „S0-Record“ (enthält Längenangabe), mehrere „S2-Records“ und ein „S9-Record“ erzeugt. „S2-Records“ werden von **RTOS–UH** stets relativiert, d. h. die Adreßangabe im „S2-Record“ zählt stets relativ zur Ladeadresse. Mit den S-Rekords kann ein EPROM–Gerät (etwa MODIPROG) direkt angesteuert werden, um die Scheibe(n) im Scanbereich im ROM abzulegen (s. Seite 625: Scanbereich ändern, oder Seite 637 neue Tabelle anschließen).

Beim Einsatz des `PROM`–Befehls sind zwei Einsatzfälle zu unterscheiden:

1. Prommen normal compilierter und geladener PEARL– oder Assembler–Programme. *name* kann ein Modul– oder ein Taskname sein. Folgt einem Modulnamen ein Stern `*` (keine Wildcard), so werden alle auf dieses Modul im Speicher unmittelbar folgenden Tasks neben dem Modul ebenfalls bearbeitet. Es werden Scheibendaten für 13–er Scheiben erzeugt, d. h. es wird ein Speicherbereich quasi als Dump im EPROM abgelegt. Beim Erzeugen des Scheibentextes werden Nulldatenblöcke durch die im 13–Code vorgesehenen Datablöcke weggekürzt. Es empfiehlt sich daher unbedingt, nicht benötigte Initialdaten von Modulvariablen vor dem Absetzen des Befehles zu „Nullen“, etwa durch eine Hilfstask, die man vorher laufen läßt. Nach Einbau der so erzeugten Scheibe generiert **RTOS–UH** beim Kaltstart die entsprechenden Module bzw. Tasks an genau die gleichen Adressen, die die Objekte im Moment des

PROM–Befehles hatten. Es muß vom Anwender dafür gesorgt werden, daß RAM–Plätze nicht doppelt belegt werden! (**RTOS–UH** richtet dann den Block, der später vom Scanner erfaßt wird, einfach nicht ein). Sollen Tasks beim Warmstart automatisch loslaufen, so benutzen Sie vor dem PROM–Befehl den Befehl AUTOSTART *modulname,taskname*.

2. Prommen von mit der CODE=$ ...,VAR=$ ...–Option erzeugten PEARL–Programmen. *name* kann nur ein Modulname sein; *name** wird impliziert und muß nicht angegeben werden. Nach dem Laden des kompilierten Programmes erscheint das Modul incl. aller eingeschlossenen Tasks als Speicherblock mit der Kennung PMDL beim S–Kommando. Es können keine Tasks aus dem Modul gestartet werden. Der PROM–Befehl erzeugt nun nur für den Bereich der Modulvariablen eine 13–er Scheibe, die beim Kaltstart des Systems den Modulvariablenblock auf der mit VAR=$ ... angegebenen Adresse einrichtet. Nulldatenblöcke werden, wie im Fall 1, weggekürzt; daher sollte das Programm mit einer Size–Angabe über die SC=–Option (Size and Clear) übersetzt werden (spart EPROM–Platz). Tasks werden in eine 1–er Scheibe umgesetzt, Prozeduren ohne Scheibenkennung abgelegt. Es werden zwei S0-... S9-Record–Blöcke erzeugt; der erste Block umfaßt die 13–er Modulvariablenscheibe und kann auf beliebiger EPROM–Adresse abgelegt werden, der zweite Block umfaßt den Code–Bereich und muß im EPROM genau beginnend mit der in der CODE=$ ...–Option angegebenen Adresse abgelegt werden. Hierzu kann es ggf. sinnvoll sein, die Ausgabe des PROM–Befehls zunächst in einen ED–File zu lenken, um beide Blöcke mit Hilfe von Editor und COPY zu trennen. Damit wird die Eingabe des Adreß–Offset beim EPROMmer erleichtert. Bei Systemen, die das Betriebssystem aus dem RAM exekutieren, muß bei der CODE=$ ... –Option die spätere tatsächliche Laufzeit–Adresse angegeben werden. Dies betrifft alle Platten-Boot–Systeme und solche, die beim Anlauf das EPROM in das RAM umkopieren (wie es viele der 68020/30/40/60- und PowerPC- Systeme tun). Für EPROM– und RAM–Layout orientiert man sich an den Adreß– und Längenangaben der Compiler–Bilanz. Sollen Tasks beim Warmstart automatisch loslaufen, so benutzen Sie auch hier vor dem PROM–Befehl den Befehl AUTOSTART *modulname, taskname*.

Die Operation erfolgt auf Ebene des Shellprozesses (!), ggf. sollte mit `O /ED/xyz` der S-Recordtext daher aus Zeitgründen zunächst in eine `ED`–Datei geschrieben werden. Drücken Sie in der Zeit nicht die „BREAK“–Taste, die Operation würde sonst abgebrochen! Sauberer ist es, einen sekundären Shellprozeß, z. B. mit Hilfe von „`DEFINE`“ mit der Aufgabe zu betrauen, insbesondere wenn im Hintergrund noch andere Echtzeitaktivitäten ungestört bleiben sollen.

Beispiele: `O /A2/;PROM Lager*;O /A1/;CLOCK`

Modul `Lager` und folgende Tasks an das Programmiergerät an `/A2/` senden, Uhrzeit erscheint wenn fertig.

```
DEFINE.xy PRIO 100--O /ED/Buffer--prom Maus1--UNLOAD xy
```

Ein sekundärer Shellprozeß (Name:`xy`) übernimmt die Aufgabe und vernichtet sich anschließend selbst. Die Aktion läuft auf niedriger Priorität ab: Modul `Maus1` prommen, S-Records in `ED`–File schreiben (für Fall 2, so lassen sich die Blöcke für Modulvariablen und Tasks einfacher trennen).

Print Working Directory **P W D**

SYNTAX: `PWD`

Beschreibung: Das mit dem Befehl `CD` vereinbarte Working-Directory kann mit dem Befehl `PWD` („Print Working Directory“) angezeigt werden. Weiterhin werden die z. Z. gültigen Execution-Directories mit ausgegeben.

Beispiele: `PWD` wenn die Shell darauf mit

```
WD=/-
XD=/-
```

antwortet, sind weder Working- noch Execution-Directory vereinbart.

`PWD` wenn die Shell darauf mit

```
WD=/ED/-
XD=/F0/cmmd
```

antwortet, ist das Working-Directory `/ED/` und das Executing-Directory `/F0/cmmd` vereinbart.

`PWD` wenn die Shell darauf mit

```
WD=/H0/TEX/DOCUS
XD=/H0/XD
+ /H1/XD2
```

antwortet, ist auf der Festplatte `/H0` das Working-Directory `TEX/DOCUS` vereinbart. Transiente Befehle und Skripte sucht die Shell zunächst unter `/H0/XD/...` . Bei Mißerfolg wird anschließend noch unter `/H1/XD2/...` gesucht.

Q A S — Quick Assembling (optional)

SYNTAX: `QAS` oder
`QAS.`*`sonprocname`* `[PRIO` *`integer3`*`] [parameterlist ]`
`QAS [PRIO` *`integer3`*`] [parameterlist]`

Beschreibung: Es wird der schnelle „native coded“ (68K-) Assembler aufgerufen. Dieser kann transient (Zeitverlust, wenn er öfter gebraucht wird!) oder aus dem Speicher (vorher laden) benutzt werden.

Es gelten alle Angaben des normalen Assemblers, siehe dazu Seite 103. Einziger Nachteil dieses ca. 2 bis 3 mal schnelleren Übersetzers ist sein sehr viel längerer Code. Wenn man max. ca. 150 kByte verschmerzen kann, so sollte man ihn bevorzugen.

Speichern Sie den S-Rekord-File des `QAS` sinnvollerweise im üblichen Execution-Directory (z. B. unter `/HO/XD`). (Die Lizenz für den normalen 68K-Assembler schließt die Verwendungsrechte des „QAS“ mit ein).

Quick Link S-Records (optional) **Q L N K**

SYNTAX: `QLNK` oder
`QLNK.`*`sonprocname`* `[PRIO` *`integer3`*`] [parameterlist ]`
`QLNK [PRIO` *`integer3`*`] [parameterlist]`

Beschreibung: Es wird der schnelle „native coded“ Linker aufgerufen. Dieser kann transient (Zeitverlust, wenn er öfter gebraucht wird!) oder aus dem Speicher (vorher laden) benutzt werden.

Es gelten alle Angaben des normalen Linkers, siehe dazu Seite 163. Einziger Nachteil dieses ca. 2 bis 3 mal schnelleren Linkers ist sein sehr viel längerer Code. Wenn man max. ca. 150 kByte verschmerzen kann, so sollte man ihn bevorzugen.

Speichern Sie den S-Rekord-File des `QLNK` sinnvollerweise im üblichen Execution-Directory (z. B. unter `/HO/XD`). (Die Lizenz für den normalen Linker schließt die Verwendungsrechte des „QLNK“ mit ein).

Q P — Quick PEARL Compilation (optional)

SYNTAX: `QP` oder
`QP.`*`sonprocname`* `[PRIO` *`integer3`*`] [parameterlist ]`
`QP [PRIO` *`integer3`*`] [parameterlist]`

Beschreibung: Es wird der schnelle „native coded“ PEARL-Compiler aufgerufen. Dieser kann transient (Zeitverlust, wenn er öfter gebraucht wird!) oder aus dem Speicher (vorher laden) benutzt werden.

Es gelten alle Angaben des normalen Compilers, siehe dazu Seite 180. Einziger Nachteil dieses ca. 2 bis 3 mal schnelleren Übersetzers ist sein sehr viel längerer Code. Wenn man max. ca. 300 kByte verschmerzen kann, so sollte man ihn bevorzugen.

Speichern Sie den S-Rekord-File des `QP` sinnvollerweise im üblichen Execution-Directory (z. B. unter `/HO/XD`). (Die Lizenz für den normalen Maxi-PEARL-Compiler schließt die Verwendungsrechte des „QP“ mit ein).

Release Semaphorvariable **R E L E A S E**

SYNTAX: `RELEASE` *hexnum8*`,`*hexnum8*`,...`
`RELEASE` *taskname*`,`*taskname*`,...`

Beschreibung: Durch *hexnum8* werden Speicheradressen angegeben, die als Semaphorvariablen in Benutzung sind. Wird dabei der Zustand „Requested“ (und task waiting) verlassen, so wird — falls noch nicht gestorben — die erste (= höchstpriorisierte) wartende Task freigegeben, die auf diese Semaphore wartete. Die Semavariable bleibt in diesem Fall im Zustand „Requested“. Wird keine wartende Task ermittelt, so wird der Wert der Semavariablen um eins erhöht.

Steht eine Task im Zustand „Waiting for SEMA“, so kann die Semaphore, auf der die Task hängt, „released“ werden. Ist die Semaphore danach frei, läuft die Task weiter. Damit entfällt das unten beschriebene Ermitteln der Adresse der Semaphore.

Die Adressen müssen natürlich zunächst ermittelt werden, dazu empfiehlt sich die Benutzung globaler Symbole und Inspektion der Lader–Liste (`LO`–Option).

Beispiele: `RELEASE 4020,10112`

`RELEASE TEST`

Wenn die Task `TEST` im Rechner vorhanden ist und auf einer Semaphore „hängt“, wird diese Semaphore „released“.

RENAME — Rename File

SYNTAX: `RENAME` *`/device/pathlist/old_filename>new_filename`*

Beschreibung: Es wird der Name der angegebenen Datei in den neuen Namen geändert.

/device/old_name: Der Devicebezeichner kann z. B. die Form `Fx` oder `Hx` haben. Ein Working-Directory wird ggf. berücksichtigt. *old_name* sollte ein gültiger File–Name sein.

new_name: Es wird nur ein Filename ohne Pathlist akzeptiert.

Hinweis: Existiert bereits eine Datei mit *new_name*, so erfolgt eine Fehlermeldung „`File in system`“ und die Umbenennung unterbleibt.

Beispiele: `RENAME /F0/MIST>HALLO`

Die Datei `MIST` auf dem Laufwerk `F0` wird in `HALLO` umbenannt.

Es existiert ein Working-Directory `/H0/sub1/sub2`:

`RENAME DAT1>DAT2`

Die Datei `/H0/sub1/sub2/DAT1` wird in `DAT2` umbenannt.

Return Files **R E T U R N**

SYNTAX: `RETURN` *pathlist–list*

Beschreibung: Eine Floppy-, Platten– oder ED-Datei wird aus der Verwaltung von **RTOS–UH** entlassen. Vorher wird die Datei geschlossen. Die Anweisung wird als Kommandofehler behandelt, wenn ein angegebenes Objekt keine Floppy–, Platten– oder ED–Datei ist.

pathlist–list: Es handelt sich um eine Liste, deren Elemente durch Leerzeichen oder Kommata getrennt werden. Ein Element der Liste besteht aus einem Device–Bezeichner (z. B. `/F0/`) und einem Filenamen. Ein eingestelltes Working–Directory wird entsprechend berücksichtigt.

Optionen: Es ist die Option `-A` (oder `-a`) zugelassen. Damit können auch Files, die in exklusiver Belegung (auch anderer Nutzer!) sind, bedingungslos zurückgegeben werden. Die Option ist nicht erforderlich, wenn das `RETURN` von einer primären Shell auf eine dem Nutzer zugeordnete Datei ausgeführt wird.

Beispiel: `RETURN /F1/QUELLE,/F0/XYZ`

`RETURN mist` bei eingestelltem `WD:/F0/-`

`RETURN -A /ED/SOURCE1;`

! → Wenn der angesprochene File nicht geöffnet ist, so antwortet das System mit einer „`... not found`“-Meldung. Die Ursache: der Filehandler sucht bei diesem Befehl nur unter den von ihm geöffneten Files und Directories.

REWIND — Rewind Files

SYNTAX: `REWIND` *pathlist–list*

Beschreibung: Die angegebenen Dateien werden zurückgespult. Die Anweisung führt zum Kommandofehler, wenn das Device nicht rückspulbar oder die Datei nicht vorhanden ist.

pathlist–list: Es handelt sich um eine Liste von Pfadlisten, deren Elemente durch Leerzeichen oder Kommata getrennt werden. Jedes Element der Liste beginnt wie üblich entweder auf der „Root-Ebene“ mit „`/`“ oder bezieht sich auf das aktuelle Working-Directory.

Beispiele:
```
REWIND /F1/QUELLE,/F0/XYZ

REWIND /ED/myfile

REWIND /SN7/ST29/H0/TEX/DOCU1
```

Im letzten Fall wird der Rechner `/SN7` (im **RTOS–UH** typischerweise eine Ethernetkopplung) als Gateway zur Station `/ST29` benutzt. Dort wird auf der Festplatte `/H0` der File `TEX/DOCU1` auf seinen Anfang gesetzt.

! → Es können nur existierende Files zurückgespult werden. Neue Files können mit `REWIND` nicht angelegt werden. Beachten Sie auch, daß der File nach dieser Operation geöffnet ist: die Plattenverwaltung wird „sprungbereit“ gehalten und schreibt nicht mehr alle Änderungen an anderen Files sofort auf das Medium zurück.

Remove File **R M**

SYNTAX: `RM` *pathlist–list*

Beschreibung: Die in der Parameterliste angegebenen Dateien werden unwiderruflich aufgegeben. Die Dateien werden aus der Systemverwaltung entfernt, so daß auf sie nicht mehr zugegriffen werden kann. Bei `ED`-Dateien wird der zur Ablage der Datei benötigte Speicherplatz (Typ `EDTF`) wieder frei verfügbar, auf Massenspeichern wird der entsprechende Platz frei. In der *pathlist–list* werden vereinbarte Working-Directories mit berücksichtigt.

Beispiele: `RM /ED/quelle, /ED/test, /F0/mist`

`RM /ED/SI`

oder mit Working-Directory `/ED/xyz`:

`RM mein` löscht `/ED/xyz/mein`

Durch Angabe einer vollständigen Pathlist können Files außerhalb eines vereinbarten Working-Directories gelöscht werden.

vereinbartes Working-Directory: `/ED/xyz`

`RM /ED/nutz1/abc` löscht angegebenen File

Hinweis: Die ausführende Shell übergibt die Kommandos an den jeweils zuständigen File–Handler, nachdem überprüft wurde, ob das angegebene Gerät überhaupt in löschbare Files untergliedert ist (siehe `SD`, `DD`-Befehl). Ist letzteres nicht erfüllt, so wird die Anweisung mitsamt dem Rest der Kommandozeile zurückgewiesen. Wenn möglich erfolgt die Meldung von genaueren Fehlern durch den File–Handler über dessen evtl. vorhandene „Report-Error"-Funktion.

! → Statt des Kommandos `RM` kann mit gleicher Wirkung auch `ERASE` eingegeben werden.

RMDIR — Remove Directory

SYNTAX: `RMDIR` *pathlist–list*

Beschreibung: Die mit dem Befehl `MKDIR` eingerichteten Directorys können mit dem Befehl `RMDIR` wieder entfernt werden. Falls das Directory noch ein Directory oder File enthält, erscheint die Fehlermeldung

`... directory active`.

In diesem Fall sind zunächst die in der Hierarchie weiter unten stehenden Directories und Files zu entfernen.

pathlist–list: Es sind alle Geräte zulässig, bei denen das entsprechende Bit im Device–Wort (`RMDIR` erlaubt, siehe `SD`–Befehl) gesetzt ist. Die einzelnen Pfadlisten werden durch Kommata oder Leerzeichen getrennt.

pathlist: Die *pathlist* hat die übliche Syntax, insbesondere sind Beschränkungen des verwendeten Filehandlers zu beachten. Siehe dazu auch den Befehl `MKDIR` auf Seite 175.

Beispiele: Mit dem Befehl `MKDIR` wurde das Directory

```
/F0/USER1/PROJEKT1
```

vereinbart. Es besteht die Möglichkeit mit

```
RMDIR /F0/USER1/PROJEKT1
```

das Directory `PROJEKT1` zu entfernen. Mit dem Befehl

```
RMDIR /F0/USER1
```

läßt sich danach auch das Directory `USER1` entfernen.

RTOS–filesystem definition **R T O S F I L E S**

SYNTAX: `RTOSFILES` *`device, device, ..`*

Beschreibung: Das mit *device* bezeichnete Gerät — typischerweise eine Floppy oder Winchester — wird unter die Verwaltung des **RTOS–UH**–Filemanagers gestellt. Wurde z. B. bisher die Floppy `/F0/` als Fremddiskette, z. B. von einem MS–DOS–Rechner stammend, behandelt, so wird nach Absetzen des Befehles `RTOSFILES /F0/` jetzt wieder eine **RTOS–UH**–Diskette im Laufwerk `F0` erwartet.

Beispiele: Eine Atari– oder MS–DOS–Diskette mit File `PAPER.TXT` soll auf eine **RTOS–UH**–Diskette kopiert werden, ungeachtet der evtl. späteren Nachbehandlung wegen anderer Sonderzeichen und Zeilenendekennung. Hinterher wird nur noch mit **RTOS–UH**–Disketten gearbeitet.

```
MSFILES /F1/                    F1 unter MS–DOS–Verwaltung
RTOSFILES /F0/                  F0 unter RTOS–UH–Verwaltung
COPY /F1/PAPER.TXT>/F0/                              Kopieren
RTOSFILES /F1/   Beide wieder unter RTOS–UH–Verwaltung
```

Hinweis: Wenn noch Files auf dem Gerät geöffnet sind, so wird der Befehl nach Meldung `... directory active ...` zurückgewiesen. Man kann aber das „Vergessen“ des nicht mehr benötigten Filesystems genau wie beim `CF` (Change Floppy)–Befehl erzwingen. Dies erfolgt durch eine spezielle Pseudo–Pathlist:

```
SYNC /F0/ (siehe SYNC, zum Retten)
MSFILES /F0/FORGET (Vergiß alte Floppy)
```

Welches Filesystem auf der Floppy gerade gültig ist, kann man jederzeit über den Befehl `FILES` (z. B. `FILES /F0/`) erfragen, da bei dessen Ausgabe die Verwaltungsstruktur mit erscheint — auch dann wenn kein File geöffnet ist.

Hinweis: Wenn nicht für alle Laufwerke einer Warteschlange der gleiche Filemanager zuständig ist (wie oben im Bsp.), so legt die Betreuungstask im Speicher eine Transfertabelle für die Zuordnung Laufwerk<->Filemanager an, die über das `S`–Kommando sichtbar ist, auch wenn kein File offen ist. Die Tabelle verschwindet wieder, sobald der zuständige Filemanager eine Verbindung herstellt und danach alle Verbindungen auf ihn selbst zeigen. Im obigen Bsp. ist der Block hinterher noch existent, weil das letzte `RTOSFILES` vom MS–DOS–Filemanager ausgeführt wird. Mit

`RTOSFILES /F1,/F1` als letztem Befehl i. o. Bsp. spart man folglich einige Bytes Speicher ein.

! → Mit welcher Fileverwaltung Ihr System startet, ist implementierungsabhängig. Das Kommando hat mit den modernen Filehandlern seine frühere Bedeutung verloren: nach Inspektion des Bootsektors schalten die neueren Filehandler erforderlichenfalls automatisch auf den jeweils anderen um. Will man jedoch eine Festplatte oder Diskette für **RTOS–UH** formatieren, so sorgt dieser Befehl, abgesetzt vor dem `FORM`-Befehl, für das gewünschte Ergebnis.

Storage **S**

SYNTAX: `S [-A[O]|-C[O]|-E[O]|-F[O]|-M[O]|-T[O]][-O]`

Beschreibung: Es wird die Speicherbelegung des gesamten Systems aufgelistet. Damit ist es jederzeit möglich, sich einen Überblick über die Systemauslastung und Belegung der einzelnen Speicherbereiche anzusehen. Aufeinanderfolgende gleiche `ED`–Blöcke werden zusammengefaßt und bei *adr2* mit einem `+` gekennzeichnet, um mehr Übersichtlichkeit zu gewährleisten.

Parameter: Durch die Angabe von verschiedenen Parametern kann die Speicherbelegung selektiv angezeigt werden, um z. B. bei größeren Mehrnutzersystem einen kurzen Überblick zu bekommen.

- A Kein Zusammenfassen von gleichen aufeinanderfolgenden `ED`–Blöcken
- C Nur Anzeige von `CWSP`–Segmenten
- E Nur Anzeige von `EDTF`–Segmenten
- F Nur Anzeige von `FREE`–Segmenten
- M Nur Anzeige von `MDLE`– und `PMDL`–Segmenten
- T Nur Anzeige von `TASK`– und `ATSK`–Segmenten
- O mit einem angehängten `O` wird die Usernummer berücksichtigt, d. h. daß nur die „eigenen“ Sektionen angezeigt werden.

Die einzelnen Ausgabezeilen haben folgendes Format:

adr1 `-` *adr2* `type RESIDENT` *taskname filename*

adr1/*adr2*: 8–stellige hexadezimale Adresse

type: Siehe Tabelle auf der nächsten Seite.

MARK	Am Anfang und am Ende des von **RTOS–UH** verwalteten Bereiches steht je eine Speichersektion dieses Typs.
FREE	Diese Speichersektion ist nicht belegt.
TASK	Es handelt sich um den Code–Körper (oder Scheinkörper bei Tasks im ROM) einer Task mit dem nachfolgend angegebenen Namen.
ATSK	Es handelt sich um eine Auto–Start–Task, die sofort beim Systemstart lauffähig ist.
TWSP	Task–Workspace der angegebenen Task.
CWSP	Communication–Element im Besitz der angegebenen Task. Falls vorhanden, wird der File–Name des Elementes ausgegeben.
PWSP	Prozedur–Workspace der angegebenen Task.
MDLE	Es handelt sich um den Kopf eines Modules mit dem nachfolgend angegebenen Namen. Dieser Name kann z. B. der bei PEARL–Programmen mögliche Modulname sein.
EDTF	Dieses Segment ist als Editor–Textfile mit dem nachfolgend angegebenen Namen im Speicher abgelegt.
PMDL	Das Segment ist ein PEARL–Modul, das mit den Optionen `CODE=$... ,VAR=$...` des Compilers übersetzt wurde. Das Modul ist nur zur Bearbeitung mit Hilfe des `PROM`–Befehls geeignet.
SMDL	Das Segment ist ein PEARL–Modul, das in der Sonderform `SHELLMODULE` übersetzt wurde und mindestens einen in PEARL codierten Bedienbefehl enthält.
????	Die Sektion ist nicht identifizierbar. Entweder liegt eine mehrfach Blockierung einer Task vor oder das Betriebssystem ist durch eine illegale Operation eines Nutzers praktisch unmittelbar vor dem Zusammenbruch. Sie sollten — soweit möglich — Ihre Dateien retten und einen `RESET` durchführen.

Tabelle 3.7: Kurznamen der Speichersektionen.

Set Device–Parameters **S D**

SYNTAX: `SD` *`device [+ hexadd--expression] value`*

Beschreibung: Die Parametrierung der durch *device* bezeichneten Datenstation wird durch das (die) Byte(s) in *value* ersetzt. Bereits in der Warteschlange (von PEARL–Programmen, sonst nur in aktueller Bearbeitung) des Gerätes stehende Ein– oder Ausgaben werden dadurch nicht mehr verändert. Die Wirkung erfolgt nur durch Information gewisser Softwarepakete über den DVDSC–Trap.

device: Ein dem System bekannter Stationsname. Dabei wird nur eine `LDN` generiert; `/A1/` läßt sich also z. B. nicht anders als `/B1/` oder `/C1/` parametrieren.

hexadd–expression: kann wie beim `SM`–Befehl benutzt werden, um z. B. das erste Byte unverändert zu lassen etc.

value: Zur Zeit 2 Bytes mit funktionellen Bits. In der folgenden Tabelle ist in Klammern exemplarisch vermerkt, welche Systemfunktionen das entsprechende Bit berücksichtigen.

1.Byte:

- $80 Die Station ist „rückspulbar“ z. B. `ED`, `F`*`x`*, `H`*`x`*.
- $40 Die Station muß vor der ersten Benutzung explizit mit „open“ und nach der letzten mit „close“ angesprochen werden, etwa `F0/F1`.
- $20 Jedem endenden Cr soll ein Lf angefügt werden, etwa `A1`(=`B1`, `C1`).
- $10 Die Station ist ein dialogfähiges Datenterminal, etwa `A1`, `A2`.
- $08 Das Echo soll explizit unterdrückt werden, nur bei den `A`*`x`*, `B`*`x`*, `C`*`x`* sinnvoll.
- $04 Die Station erlaubt das Löschen bezeichneter Files, nur bei `F`*`x`*/`H`*`x`* oder `ED` sinnvoll.
- $02 Die Station erlaubt die Ausgabe von Daten.
- $01 Die Station erlaubt die Eingabe von Daten.

2. Byte:

$80 Die Station akzeptiert ein explizites `DIR` oder `FILES`–Kommando, nur `Fx`/`Hx` und `ED`.

$40 Die Station akzeptiert ein explizites `FORM`-Kommando, z. B. `Fx`, `Hx`.

$20 Das `CF`–Kommando ist zugelassen.

$10 `RMDIR`, `MKDIR` möglich.

$08 `SYNC`, `SEEK`, `SAVEP`, `TOUCH` möglich.

$04 Error-Report kann angefordert werden.

$02 Editor `ED`: Das angeschlossene Terminal macht keinen automatischen Wrap am Zeilenende.

$01 Editor `ED`: Der Cursor soll über ESC–Sequenzen gesteuert werden (VT–52).

Beispiele:

```
SD /A2/ 0B
```

`A2` ist für Ein–/Ausgabe zugelassen, es wird kein Echo gemacht. Diese Einstellung ist z. B. für eine Rechnerverbindung sinnvoll.

```
SD /A2/ 33
```

`A2` ist ein dialogfähiges Gerät, das jedem CR ein LF anhängt und für Ein–/Ausgabe zugelassen ist. Einstellung für ein Terminal.

```
SD /PP/ 02
```

Umparametrierung des Printer–Ports für einen Drucker, der selbst ein LF nach jedem CR generiert oder Ausgabe einer MS–DOS–Datei, in der jede Zeile mit CR/LF endet.

```
SD /A1/+1 01
```

Anpassung des Editors an ein VT–52 Terminal (Cursorsteuerung über ESC–Sequenzen).

Hinweis: Im Umfeld dieses elementaren Bedienbefehles sind Skripte gebräuchlich, die einem die mühselige Kodierung des Bitmusters abnehmen.

Time–Sharing für Task's **S H A R E**

Syntax: `SHARE [PRIO integer3]`

Beschreibung: Der Befehl `SHARE` dient dazu, ein gleichzeitiges Abarbeiten von gleich priorisierten Tasks zu ermöglichen.

Sind z. B. zwei Compiler vom Nutzer gestartet worden, so werden diese normalerweise nacheinander bearbeitet, da sie die gleiche Priorität haben. Damit sie gleichzeitig bearbeitet werden, wird `SHARE` mit der `PRIO Taskprio - 1` aufgerufen. Nachdem dieses erfolgt ist, wird die Prozessorkapazität auf beide Compiler aufgeteilt.

Ab dem erstmaligen Aufruf bleibt für die angegebene Task–Priorität die Time–Sharing–Funktion aktiv, bis die `SHARE/xx`–Task entladen oder terminiert wird. Auch ein Abort unterbricht nicht das Time–Sharing.

Beispiele:
```
P xxxx > yyyy /* läuft mit Prio 20 */
P zzzz > uuuu /* läuft mit Prio 20 */
SHARE /* jetzt werden beide bearb.*/
```

Der Aufruf von `SHARE` richtet einen Sohnprozeß `SHARE/xx` ein, der das eigentliche Scheduling ausführt, indem jedem Prozeß dieser Priorität der Prozessor für ca. 50 msec zugeteilt wird. Haben Sie in Ihrem System noch Tasks auf höheren Prioritäten laufen, so entziehen diese natürlich den Prozessor. Sind die höher priorisierten Tasks zyklisch eingeplant, kann der Eindruck entstehen, das Timesharing funktioniert nicht, weil eine höher priorisierte Task immer dann läuft, wenn eine bestimmte Task im Timesharing an der Reihe ist.

Der Aufruf von `SHARE` muß mit einer um 1 höheren Priorität erfolgen, als der Level, der beeinflußt werden soll, damit die Share–Task sofort aktiv wird.

WICHTIG: Um z. B. die Prioritätsstufe 30 zu beeinflussen muß

`SHARE PRIO 29` eingegeben werden.

Standard Priorität für `SHARE`–Task ist 20 (also ein implizites `PRIO 19` beim Aufruf).

Um verschiedene Prioritäts–Level im Time–Sharing Betrieb zu nutzen, können beliebig viele `SHARE`–Kommandos abgesetzt werden.

S H E L L — Install Shell

Syntax: `SHELL` [`PRIO` *integer3*] [`SZ` *hexnum6*] [*path*] [*positpara*]

Beschreibung: Der Befehl `SHELL` dient dazu, eine skriptgesteuerte Shell einzurichten und diese an die Stelle der bisherigen primären Shell zu setzen. Mit *path* wird normalerweise das mitgelieferte Skript adressiert, mit dem man sich eine Unix-ähnliche Shell erzeugen kann. (siehe Seite 74)

Beim Aufruf dieses Skriptes über den Befehl `SHELL` gelten außer beim Sohnprozeßnamen die gleichen Aufrufparameter wie beim Aufruf über `EX`. Als Name erhält der Sohnprozeß „`#BSH`*xx*“, wobei *xx* die Usernummer des Aufrufers ist. Zusätzlich wird der Sohnprozeß als sekundäre Shell in das Userenvironment eingetragen und damit beim Anschlag der Taste „`CTRL A`“ des Users fortgesetzt. Die primäre Shell ist dann nur noch über die „`BREAK`“-Taste erreichbar. Das SHELL-Skript sollte in einer Endlosschleife Befehle einlesen, ausführen und sich dann für den nächsten Anschlag der Taste „`CTRL A`“ suspendieren. Damit kann man sich eine Shell mit eigenem Environment und geringerer Priorität einrichten. Die sekundäre Shell kann mit dem `EXIT`-Befehl beendet werden. Nach einem Warmstart läuft das SHELL-Skript neu an und bleibt als sekundäre Shell aktiv. Pro User ist nur eine sekundäre Shell einrichtbar, der SHELL-Befehl darf nicht gestapelt abgesetzt werden.

Beispiel:

```
SHELL /HO/XD/SHELL;
```

Show state of specified Task **S H O W / S H**

SYNTAX: `SHOW` *`taskname,taskname,...`*
`SH` *`taskname,taskname,...`* (Kurzform)

Beschreibung: Es wird eine Zustandszeile für jede der angegebenen Tasks ausgegeben. Der Aufbau dieser Zustandszeile entspricht genau denen beim `L`–Kommando. Wenn einzelne Tasks der Liste nicht vorhanden sind, so erfolgt die Meldung „`... not loaded`“.

Beispiele:
```
SHOW abcd ASMB12

SH PCOM45
```

S M — Set Memory

SYNTAX: `SM` *adr value value ...*

Beschreibung: Es kann der Inhalt einer oder mehrerer Speicherzellen verändert werden. Ist *value* eine n–stellige Hexadezimalzahl, so werden (n+1)//2 Bytes beginnend bei *adr* abgelegt, wobei der Ablagezeiger für den nächsten Wert *value* anschließend um diesen Betrag weiterrückt.

adr: 1...8–stellige Hexadezimalzahl, mit der die Adresse der Speicherzelle angegeben wird. Es können auch mehrere Hexadezimalzahlen angegeben werden, die durch ein `+/-` Zeichen verbunden sein müssen und vom System addiert/subtrahiert werden.

value: 1...8–stellige Hexadezimalzahl, mit der der neue Inhalt der Speicherzelle angegeben wird.

Hinweis: Der Prozessor greift im Usermode auf den Speicher zu. Damit sind je nach aktueller Hardware nicht alle Adressen erreichbar. Man erkennt dies an der Shellreaktion „`... bus error`“. Aus diesem Grund existiert noch eine erweiterte Form als `SMX`-Befehl, der typischerweise transient ausgeführt wird. Mit dem `SMX` sind dann folgende Zusatzparameter (unmittelbar hinter dem Befehl, vor der Adresse) möglich:

- `-Sx` Es wird festgelegt, daß der Zugriff im Supervisor–Mode des Prozessors ausgeführt wird.
- `-x` Zugriff im User–Mode durchführen.
- `-xB` Bytezugriff mittels Befehl `MOVE.B`.
- `-xW` Wortzugriff mittels Befehl `MOVE.W`.
- `-xL` Langwortzugriff mittels Befehl `MOVE.L`.
- `-xM` Zugriff mittels Befehl `MOVEP.W`.
- `-xP` Zugriff über POT–Trap — für Pbus Ausgabe.

V O R S I C H T !

Diese Anweisung darf nur mit größter Sorgfalt benutzt werden, da der ausführende Shellprozeß nicht prüft, ob evtl. lebenswichtige Daten oder Zeiger des Systems zerstört werden. Diese Zerstörungen können zunächst verborgen bleiben und sich erst später bemerkbar machen.

Beispiele: `SM 14EAF 02 AFFE12CD 3 4`

Ab der Adresse `$14EAF` wird `$02AFFE12CD0304` abgelegt.

`SM 1000` (keine Operation)

`SM 6000+10 4`

Auf Adresse `$6010` wird das Byte `$04` abgelegt.

`SMX -SB 400 45`

Auf Adresse `$400` wird mit einem Supervisor Byte–Zugriff das Byte `$45` abgelegt.

`SMX -W 3000 5 6`

Ab Adresse `$3000` wird mit Wortzugriffen `$00050006` abgelegt.

`SMX-L 80000 1 2 30`

Ab Adresse `$80000` wird mit Langwortzugriffen `$0000000100000002000000030` abgelegt.

S U S P E N D / S U — Suspend Task

SYNTAX: `SUSPEND` *`taskname,taskname,...`*
`SU` *`taskname taskname ...`* (Kurzform)

Beschreibung: Die angegebenen Tasks werden in den Zustand „blockiert“, wartend auf `CONTINUE`, gebracht.

Die ausführende Shell prüft, ob die angegebenen Tasks vorhanden sind, ggf. erfolgt die Meldung „`... not loaded`“. Eine weitere Analyse des aktuellen Taskzustandes findet nicht statt, d. h. die Tasks werden im aktuellen Zustand eingefroren. Wartete die angesprochene Task z. B. auf die Zuteilung einer Semaphore, so hat sie anschließend eine Doppelblockierung. Ihr Zustand wird dann vom `L` bzw. `SHOW` Kommando mit `????` gezeigt.

Beispiele:
```
SUSPEND ABCD TEST

SU init
```

Synchronize File–System **S Y N C**

SYNTAX: `SYNC` *`Floppy-/Winch-devicelist`*

Beschreibung: Verwaltungsdaten im Speicher und Inhalt auf den bezeichneten Geräten werden aktuell abgeglichen, so daß bei einem Netzausfall die Daten gesichert sind. Die Zustände der Files werden dabei nicht verändert, die Files also auch nicht geschlossen.

Solange auf einem Massenspeicher (Diskette, Festplatte) noch geöffnete Files existieren, ist nicht gesichert, daß die zuletzt geschriebenen Daten auch wirklich bereits auf dem Medium abgelegt wurden. Die Filehandler sparen auf diese Weise erhebliche Zeit für die Kopfpositionierung. Dieser Befehl erzwingt das Hinausschreiben aller Daten, die bisher nur im Speicher des Rechners („Disc-cache“) angelegt oder verändert wurden.

Der Befehl darf gegeben werden, wann immer man das für sinnvoll hält. Eventuell passiert auf den Befehl hin überhaupt nichts, etwa wenn man nur von dem Gerät gelesen hat.

Beispiele: `SYNC /H0/ /H1/ /H2/ /F0/`

Hier werden alle ggf. offenen Files auf den Winchester– und Floppylaufwerken synchronisiert, d. h. für diesen Zeitpunkt besteht Übereinstimmung zwischen der File–Verwaltung im Speicher und dem Inhalt auf dem Medium.

Hinweis: Der Befehl eignet sich auch, um eine Diskette in großer Eile entnehmen zu können. Dabei wird nach dem `SYNC` die Verwaltungsinformation gelöscht, und man kann danach die Floppy herausnehmen.

```
SYNC /F0/;CF /F0/FORGET;CLOCK;
```

Der anschließende Clock–Befehl dient nur dazu, um den Abschluß der Aktion erkennbar zu machen.

Achtung: Sie können mittels eines `DEFINE`–Befehles einen `SYNC` zyklisch einplanen, um im Falle eines Rechnerabsturzes oder Stromausfalls die Gefährdung ihrer Daten auf der Festplatte zu verringern:

```
DEFINE.asyn PRIO 100--SYNC /H0/ /H1/;ALL 30 SEC asyn
```

Alle 30 Sec werden die Directories auf den Winchesterlaufwerken auf den aktuellen Stand gebracht. Mit der Angabe der niedrigen

Priorität wird erreicht, daß ein evtl. laufender Datenstrom in Richtung Platte nicht gestört wird.

Terminate Task **T E R M I N A T E / T**

SYNTAX: `TERMINATE` *`taskname taskname ...`*
`T` *`taskname,taskname ...`* (Kurzform)

Beschreibung: Die angegebenen Tasks werden beendet. Der von ihnen belegte Prozedur–Workspace wird an das System zurückgegeben. Communication–Elemente, die nicht in einer Ausgabeschlange oder in Bearbeitung einer Inputtask sind, werden ebenfalls sofort an das System zurückgegeben. Die automatische Rückgabe aller anderen Communication–Elemente nach Abschluß der Ein– oder Ausgabe wird vorbereitet. Ist die Task nicht vom Typ `RESIDENT`, so wird auch der von ihr belegte Task–Workspace an das System zurückgegeben.

! → Semaphore, die von der Task belegt wurden, bleiben wie bei der PEARL–Anweisung `TERMINATE` unberührt und können somit andere Tasks dauerhaft blockieren.

Die ausführende Shell prüft nicht, ob die angegebene Task existiert oder ihr Laufzustand eine Terminierung ermöglicht, also sie nicht im Zustand `DORM` ist. Eine Fehlermeldung wird ggf. vom Betriebssystemkern erzeugt. Meldung: „`... not loaded (terminate)`“.

Beispiele:
```
TERMINATE TEST INIT

T RUN
```

T O U C H — Touch a File

SYNTAX: `TOUCH` *pathlist–list* (Form A)
`TOUCH` *timepara datepara pathlist–list* (Form B)
`TOUCH` *datepara timepara pathlist–list* (Form B)
`TOUCH -R` *pathlist–list* (Form C)

Beschreibung: Mit diesem Befehl kann man den (scheinbaren) Erstellungszeitpunkt von selektierten Files aktualisieren, willkürlich setzen oder sich ausgeben lassen.

Form A: Die in der *pathlist–list* bezeichneten Files erhalten das aktuelle Datum und die aktuelle Zeit aus dem Betriebssystem als neues "Erstellungsdatum". Inhaltliche Änderungen an den Files werden nicht vorgenommen.

Form B: Die in der *pathlist–list* bezeichneten Files erhalten als neues „Erstellungsdatum" die mit *timepara* und *datepara* angegebenen Werte. Die Syntax dieser Parameter entspricht derjenigen bei `CLOCKSET` und `DATESET`. Bei der Uhrzeit dürfen die Sekunden fehlen, da sie ohnehin nicht im Filesystem abgelegt werden können. Fehlt einer der beiden Parameter, so wird er durch den aktuellen Wert aus dem Betriebssystem ersetzt. Will man z. B. nur die Uhrzeit verändern, muß neben der neuen Uhrzeit das alte Datum explizit angegeben werden.

Form C: Es wird für jeden File der *pathlist–list* Datum und Uhrzeit der letzten Änderung bzw. die letzten mit `TOUCH` abgelegten Daten ausgegeben.

Hinweis: Die Bedeutung des Befehles in der „Form A" liegt bei den sogenannten „Make"-Skripten: durch einen „Touch" kann man die erneute Einbeziehung des Files in Übersetzerläufe erzwingen.

Die „Form B" kann für Archivierungszwecke interessant sein, wenn auf Grund irgendwelcher Kopierwege das Erstellungsdatum nicht mehr mit Protokollangaben übereinstimmt. Auch läßt sich damit natürlich ein „ewig neuer" File simulieren, wenn man den Zeitpunkt in die Zukunft verlagert.

In der „Form C" schließlich ist der Befehl eine große Hilfe, wenn man schnell prüfen möchte, ob ein bestimmtes Duplikat erneuert werden muß, weil es inzwischen veraltet ist. Innerhalb von Shellskripten ist diese Form des Befehles noch in anderem Zu-

! → sammenhang interessant: es ist die schnellste Möglichkeit, um

festzustellen, ob es einen bestimmten File überhaupt gibt. Der File wird nämlich nicht geöffnet oder sonstwie in Status oder Inhalt verändert. Im Gegensatz dazu wird bei einem versuchsweisen „`REWIND`“ der erste Datenblock eingelesen, und der File bleibt geöffnet.

Bei der Anwendung des Befehles in der „Form C“ über das Netz wird im Fehlerfall, wenn der File im fernen Rechner nicht existiert oder der Zugriff nicht erlaubt ist, Datum und Uhrzeit des eigenen Rechners eingesetzt. Man kann daher über das Netz mit `TOUCH` die Existenz des Files und die Zugriffsberechtigung nicht sicher feststellen. Der Grund dafür liegt darin, daß ältere Netzsoftware den zugehörigen I/O-Befehl nicht beherrschte und nur durch diese Strategie der `COPY`-Befehl (er benutzt intern den `TOUCH`-I/O-Befehl) auch mit solchen Zielrechnern möglich ist.

Beispiel:

```
TOUCH /H0/TEX/DOCU2.TEX
TOUCH 12-01-1994 13:45:00 /F0/testfile /F0/backup
touch 12:00 /ed/test3      (Datum von heute einsetzen)
TOUCH -R /ED/GRAFF5        (hier antwortet die Shell)
```

T R A C E — Switch Task to Trace Mode

SYNTAX: `TRACE` *`taskname adr`* (Form A)
`TRACE` *`taskname`* `L` *`linenr`* (Form B)
`TRACE` *`taskname`* `L` *`linenr,linenr`* (Form B)

Beschreibung: Die angegebene Task wird in den Hardware–Trace–Mode überführt (Form A) bzw. die Programmzeilenüberwachung wird eingeschaltet (Form B). Sobald die durch *adr* (Form A) angegebene Adresse, bzw. eine der max. zwei angebbaren Zeilennummern (Form B) überlaufen wird, wird die Task durch das Betriebssystem suspendiert und die Meldung

`taskname``:`*`adr`* `BREAKPOINT` (Form A) oder
`taskname``:L=`*`linenr`* `BREAKPOINT SUSPENDED` (Form B)

ausgegeben. Das Erreichen des Haltepunktes löscht diesen nicht. Nach einer `CONTINUE`–Anweisung kann also der gleiche Haltepunkt erneut angelaufen werden.

Der Befehl ändert nichts am Taskzustand einer Task, d. h. auch inaktive oder eingeplante Tasks dürfen angesprochen werden.

War die Task bereits im Trace–Mode, so werden alle alten Haltepunkte gelöscht. Eine vollständige Aufhebung der Programmzähler– (Form A) oder Zeilenüberwachung ist nur mit Hilfe des `NOTRACE`–Kommandos möglich. Auch nach einem Warmstart des Systems sind bei allen Tasks die Trace–Modi gelöscht.

Form A: Die Task läuft im TRACE–Mode wie in „Zeitlupe“ ab. Jeder Maschinenbefehl löst eine Trace–Exception aus, in der der Adreßvergleich erfolgt.

Instruktionen nach TRAPs können nicht als Haltepunkte erkannt werden.

Form B: Das Diese Variante eignet sich für PEARL-Programme und Shellskripte. Ein PEARL-Programm muß zumindest in der Zeile, bei der die Task anhalten soll, mit der Markierungsoption `/*+M*/` ohne den Compiler-Mode `NOLSTOP` übersetzt sein.

Die Task wird suspendiert, sobald die erste in der angegebenen Zeile (bzw. einer der beiden, falls zwei angegeben waren) beginnende Anweisung erreicht wird. Die Anweisung wird jedoch noch nicht ausgeführt, sondern als erste nach der `CONTINUE`–Anweisung ausgeführt.

Die Laufgeschwindigkeit der Task wird durch das Zu- oder Abschalten des Line–Trace–Modus praktisch nicht geändert. Allerdings ist das Compilat an sich bereits durch Benutzung der `+M`–Option um ca. 5 bis 500% verlangsamt.

adr: 1...8–stellige Hexadezimalzahl, mit der die Adresse des Breakpoints angegeben wird.

linenr: 1...5–stellige Dezimalzahl, die die Programmzeilennummer des Haltepunktes bezeichnet. Höhere Zeilennummern als 34575 können nicht mit Erfolg adressiert werden, da die interne, zu früheren Systemversionen abwärtskompatible Codierung der Zeilennummer durch den Compiler dies nicht zulässt. Beide Haltepunkte sind völlig gleichwertig. Wird nur einer angegeben, so existiert ein ggf. früher eingegebener zweiter nicht mehr.

Beispiele:

```
TRACE TEST 2346
```

Die Task `TEST` wird in den Hardware–Trace überführt und soll bei Adresse `$2346` suspendiert werden.

```
TRACE ABCD L 55 76
```

Die Hochsprachtask `ABCD` soll beim Erreichen der Zeilen `55` oder `76` suspendiert werden.

Hinweise: Mit dem Hardware–Tracer können durchaus auch Hochsprach–Codierte Tasks überwacht werden, wenn folgendes beachtet wird:

- Virtuelle Befehle können nicht erkannt werden.
- Der erste Befehl nach einem virtuellen Befehl kann nicht als Haltepunkt dienen.
- Der To–Virtual–Befehl kann benutzt werden.

Werden Adressen oder Zeilennummern in Prozeduren angewählt, so beeinflußt die `TRACE`–Anweisung natürlich die anderen Tasks nicht, die diesen Haltepunkt überlaufen.

Es ist möglich — bei unabhängig compilierten Modulen —, daß Programmzeilennummern mehrfach vorhanden sind. In diesem Fall — den man möglichst mit Hilfe der `SETLINE`-Option des Compilers vermeiden sollte — ist nach Erreichen des Breakpoints zusätzlich der PC (oder ersatzweise A6) zu inspizieren, um festzustellen, welche der gleichnumerierten Zeilen getroffen wurde. Gängige und bewährte Praxis ist hier, mit Hilfe von `SETLINE` die

Tausenderstelle der Startzeilennummer als Modulidentifikator zu verwenden.

Mit der `+M`–Option übersetzte Programme können nur durch Neucompilation ohne diese Option wieder auf maximale Laufgeschwindigkeit und minimalen Code gebracht werden.

Im Gegensatz zu anderen Konstruktionen wird der Code einer Task oder Prozedur durch die Anwendung der `TRACE`–Anweisung im Speicher nicht verändert.

Trigger Interupt **T R I G G E R**

SYNTAX: `TRIGGER EV` *hexnum8*

Beschreibung: Sämtliche Interrupts, die durch das Bitmuster von *hexnum8* mit einer „`1`“ selektiert werden, passieren die durch `ENABLE/DISABLE` eingestellte Interruptmaske des Systems. Ist dort keiner der durch *hexnum8* ausgewählten Interrupts freigegeben, so ist die Anweisung ohne Wirkung. Das System unterscheidet nicht, ob ein Interrupt durch `TRIGGER` oder durch die Hardware ausgelöst wurde. Diese Anweisung eignet sich also hervorragend zum Austesten eines interruptgesteuerten Programms.

hexnum8: 1...8–stellige Hexadezimalzahl, die das 32 Bit Ereignismuster beschreibt. Bei weniger als 8 Stellen werden links Nullen ergänzt.

Beispiele: Vorgeschichte: `WHEN EV 3 C TEST; ENABLE EV 7; WHEN EV 6 XYZ;`

`TRIGGER EV 1` die Task `TEST` wird fortgesetzt.

`TRIGGER EV 4` die Task `XYZ` wird aktiviert.

`TRIGGER EV 2` `TEST` wird fortgesetzt, `XYZ` aktiviert.

`TRIGGER EV FFFFFFFF` alle „enabled“ Interrupts werden gefeuert.

Hinweis: Die `TRIGGER`-Anweisung entspricht der gleichnamigen PEARL–Anweisung.

T Y P E — Type a File

SYNATX: `TYPE` *device-file-spec*

Beschreibung: Es wird der durch die *device-file-spec* angegebene File auf dem USER–Terminal ausgelistet. Das USER–Terminal ist immer die Schnittstelle, von der der Befehl aufgerufen wurde.

Beispiele: `TYPE /H0/mein/mist`

Es wird der File `mist` von der Festplatte `/H0/` im Subdirectory `mein` auf dem Terminal ausgegeben.

Bemerkung: Mittels `O`–Kommando kann die Ausgabe auch umgelenkt werden. In Wirklichkeit ist das Kommando ein „COPY“-Befehl, nur mit anderen Default-I/O-Parametern. Mit den Parametern `SI`, `CO`, `SC` kann der Befehl alle Möglichkeiten von `COPY` nutzen. Siehe dazu Seite 115, mit der Beschreibung von `COPY`.

Unload Tasks or Modules **U N L O A D**

SYNTAX: `UNLOAD [-A] taskname modulname ...modulname*`

Beschreibung: Es werden die in der Liste angegebenen Namen in der Systemverwaltung der Tasks und Module gesucht. Existiert ein Name nicht, so wird die Meldung „`... not loaded`“ abgesetzt und das Kommando mit dem nächsten Element fortgesetzt. Wurde eine Task oder ein Modul von einem anderen Nutzer geladen oder zuletzt aktiviert, wird der Befehl nicht ausgeführt!

Parameter: Soll ein Programmpaket, das von einem anderen Terminal (anderer User) geladen wurde, entladen werden, so ist der Parameter `-A` anzugeben. Eine Überprüfung auf gleiche User findet dann nicht mehr statt, so daß auch Tasks anderer User entladen werden können. Ebenso sollte verfahren werden, wenn ein Programm von mehreren Usern gemeinsam genutzt wird.

taskname: Die Task wird ausgeplant, gepufferte Aktivierungen werden gelöscht und anschließend wird, falls erforderlich, die Task terminiert. Dann verschwindet sie endgültig aus der Verwaltung von **RTOS–UH**.

modulname: Folgt dem Modulnamen kein `*`, so wird nur die dem Modulnamen zugeordnete Speichersektion entfernt. Vorsicht bei PEARL–Modulen!! Dies kann zum Zusammenbruch des Systems führen, falls anschließend noch Tasks auf Modulvariablen oder Datenstationen in dieser Sektion zugreifen.

Folgt dem Modulnamen ein `*`, so werden sowohl die Modulsektion als auch sämtliche (!) in unmittelbarer Speicherfolge liegende Tasks entfernt. Dieser Fall ist hauptsächlich für das großzügige Aufräumen im System gedacht und verlangt eine gewisse Vorsicht, da man vorher feststellen muß, an welcher Stelle die erste Nichttasksektion in der Kette steht. Eine logische Zugehörigkeit der mitentladenen Folgetasks zu dem Modulnamen kann die Shell nicht überprüfen! Allerdings kann nur durch das unglückliche Zuladen von assemblerkodierten reinen Tasksektionen hier in der Praxis Ungemach entstehen.

Beispiele: `UNLOAD TEST KALA* BAUER`

Hinweise: Systemtasks können weder versehentlich noch absichtlich beeinflußt werden. Bereits in der Ausgabeschlange stehende Communication–Elemente einer Task verbleiben dort und werden

weiter bearbeitet, allerdings ist der Task–ID gelöscht, so daß als Besitzer `(RTOS)` beim `S`–Befehl ausgegeben wird. Hängende Eingaben müssen abgeschlossen sein, damit das „CE“ frei wird.

Wait for following statements **W A I T**

SYNTAX: `WAIT`

Beschreibung: Der Shellprozeß setzt sein individuelles „Waitflag“. Alle folgenden Befehle bis zum Ende der Anweisungszeile werden nun sequentiell abgearbeitet, solange sie den Fehlerstatus „o.k.“ zurückgeben. Das Kommando zwingt den ausführenden Shellprozeß bei Befehlen, die Sohnprozesse generieren (z. B. `P`, `COPY`, `AS`), in einen Wartezustand („`SEMA`“) bis zum regulären oder irregulären Ende des Sohnprozesses. Zum Unterschied zwischen `WAIT` und „`--`“ (Doppelminus) siehe Seite 66.

Beispiel:
```
WAIT; P /ed/test lo liste; load;
```

Erst wenn der Compiler fertig ist und keinen Fehler festgestellt hat kommt es zur Ausführung des „`load`“-Befehles.

! → Gibt der Sohn, auf den der Shellprozeß wartet, den Fehlerstatus „fehler“ zurück, so meldet die Shell sich mit „`.... operation failed`“ über `Stderr` und unterläßt die Ausführung der restlichen Anweisungen in der Zeile. Auch wenn der Sohn durch „`TERMINATE`“ von irgendwoher gewaltsam beendet wurde, gibt dieser ebenfalls den Status „fehler“ an die Shell zurück, und die Shell ist ebenfalls wieder entblockiert. Gleiches gilt im Falle sekundärer Shellprozesse für beliebige Vater-Sohn-Ketten: die ganze Kette wird rückwärts freigegeben.

! → Auch wenn die Shell scheinbar auf eine Semaphore („`SEMA`“) wartet, so gibt es dennoch nirgendwo eine entsprechende Speicherzelle: Der Laufzustand des Sohnes selbst ist die blockierende Größe. Nach System-Abort und auch nach dem „Notruf“ der Shell über die `BREAK`-Taste ist daher auch der Systemzustand wieder ganz normal.

! → Auf die Beendigung freier Tasks, die nicht erst durch ein nachfolgendes Kommando entstehen, kann mit diesem Befehl **nicht** gewartet werden. Zwar benutzt `WAIT` intern den `WFEX`-Trap, wendet ihn aus Sicherheitsgründen jedoch nur eingeschränkt an.

W H E N — When event activate or continue given Task

SYNTAX:
```
WHEN EV hexnum8 ACTIVATE taskname [PRIO integer3]
WHEN EV hexnum8 CONTINUE taskname
WHEN EV hexnum8 taskname [PRIO integer3]
WHEN EV hexnum8 C taskname
```

Beschreibung: Bereits bestehende Einplanungen für eine Aktivierung (bei ... ACTIVATE) bzw. zur Fortsetzung (bei ... CONTINUE) werden gelöscht und die angegebene Einplanung für den mit *hexnum8* bezeichneten Prozeßinterrupt wird eingetragen. Wird bei ... ACTIVATE keine Priorität angegeben, so wird die taskeigene Priorität eingesetzt. Die aktuelle Priorität einer gerade laufenden Task wird dadurch jedoch nicht verändert, sondern erst, wenn die Einplanung zur Aktivierung führt.

hexnum8: 1...8–stellige Hexadezimalzahl, die das 32 Bit Ereignismuster beschreibt. Die Aktivierung bzw. Fortsetzung erfolgt, wenn das Bitmuster eines Interruptereignisses (bzw. einer TRIGGER–Anweisung) mindestens ein gesetztes Bit gemeinsam mit dem durch *hexnum8* angegebenen Ereignismuster besitzt (d. h. wenn die „UND“–Verknüpfung nicht 00000000 ergibt. Bei weniger als 8 Stellen werden links Nullen ergänzt.

Beispiele:
```
WHEN EV 2 XYZ PRIO 20
```

```
WHEN EV FFFFFFFF CONTINUE ABCD
```

ABCD wird bei einem beliebigen Prozeßinterrupt, sofern er „enabled“ ist, fortgesetzt.

Who has shell access

W H O

Syntax: `WHO`

Beschreibung: Es werden alle Prozesse mit ihrem aktuellen Laufzustand und ihrer User-ID aufgelistet, die dem System als primäre Shellprozesse bekannt sind. Sekundäre Shellprozesse kann man mit diesem Befehl nicht erkennen. Wenn sich ein oder mehrere Nutzer über ein Netz eingelogged haben, so erscheinen die entsprechenden temporären primären Shellprozesse ebenfalls in der Liste. Das Ausgabeformat entspricht genau dem Zeilenformat beim `L`-Befehl, der auf Seite 153 beschrieben ist.

Beispiel:
```
WHO;
```

! → Wenn der primäre Prozeß mit Hilfe des optionalen `SHELL`-Befehles abgehängt wurde, übernimmt ein sekundärer Shellprozeß seine Funktion. Weil der ursprüngliche (primäre) Shellprozeß weiterhin existiert, erscheint er auch noch in der ausgegebenen Liste. Er kann ja auch durch `BREAK` noch immer gerufen werden. In der Extrazeile mit dem Vorspann „`You:`“ erscheint in solchen Fällen jedoch nicht der primäre Prozeß, der zu dem Terminal gehört.

In der „You:“-Zeile wird in jedem Fall der gerade tatsächlich aktive Prozeß ausgegeben. Hier kann darum ausnahmsweise auch ein sekundärer Shellprozeß beschrieben werden.

(Leere Seite vor neuem Kapitel)

Kapitel 4: Der Editor RTOS-WORD

4.1 Einleitung

Der Editor RTOS-WORD ermöglicht ein komfortables Editieren von beliebig großen und einer sehr hohen Anzahl von Texten. Jeder zu bearbeitende Text muß allerdings komplett im RAM gehalten werden, wobei eine Fragmentierung des Textes erlaubt ist (macht RTOS-UH bei Bedarf automatisch). Liegen die zu bearbeitenden Texte auf einer Floppy bzw. Harddisk oder auf einem über ein Netzwerk angeschlossenen Rechner, legt RTOS-WORD eine Textkopie auf dem Device `/ED` des eigenen Rechners ab. Dadurch ist ein Verlassen ohne Abspeichern möglich. Liegt das File direkt auf `/ED`, arbeitet RTOS-WORD direkt auf der Datei und erspart RTOS-UH die Kopie im eigenen RAM. In diesem Fall ist zwangsläufig ein Verlassen nur mit Abspeichern möglich.

Die maximal erlaubte Spaltenzahl des zu editierenden Textes beträgt 231 (automatische Quelltextanpassung durch Umknicken beim Einlesen), die Zeilenzahl ist nur durch den Speicherplatz begrenzt. Allerdings können nur die ersten 65500 Zeilen verändert werden. Der Text darf alle Textzeichen enthalten, allerdings werden nicht darstellbare Zeichen durch ein „@“ im Fenster dargestellt. Die Standardgröße des Fensters beträgt 80 Spalten mal 25 Zeilen.

Jedem Text ordnet RTOS-WORD einen Zeilenpuffer zu, mit dem sehr schnell gearbeitet werden kann. Weiterhin gibt es einen gemeinsamen Blockpuffer aller Texte, der als Zwischenspeicher für verschiedene Blockoperationen und dem Austausch von Blöcken zwischen den Texten dient.

Eine Besonderheit ist die Fernsteuerung. Über das Device `/VO` läßt sich der Editor fernsteuern, d. h. beliebige Befehlskombinationen behandelt der Editor wie Nutzereingaben. Eine Ausführung von Batch-Dateien ist ebenfalls möglich.

Der Editoraufruf `WE` beinhaltet eine Umschaltung auf ein eigenes Editorfenster, falls ein Window-Manager (WiM) im System vorhanden ist. In diesem Fall (Window-Modus) arbeitet der Editor mausunterstützt (Pull-Down-Menüs, Cursorpositionierung, diverse Parametereinstellungen). Jeder Text erhält sein eigenes in der Größe einstellbares Fenster, weiterhin gibt es ein Gruppenfenster, das eine Übersicht aller gleichzeitig bearbeiteten Texte gibt und auch der Umschaltung zwischen den Texten dient. Der Nutzer hat dann auch eine weitere Entscheidungsfreiheit bei der Bearbeitung mehrerer Texte. Er kann wahlweise

mehrere Editoren über `WE` aufrufen (verschiedene Editor-Tasks, kein gemeinsamer Blockpuffer, kein gemeinsames Gruppenfenster) oder mehrere Texte mit einer Editor-Task bearbeiten.

4.2 Erste Schritte

4.2.1 Öffnen einer Datei

Bereits beim Aufruf kann der Benutzer dem Editor einige Parameter übergeben. An dieser Stelle sollen erst einmal die Aufrufformen beschrieben werden, die maximal den Dateinamen als Übergabeparameter enthalten:

`WE` *`[device/][subdirectories/]filename`* (Form 1)
`WE SC` *`[device/][subdirectories/]filename`* (Form 2)
`WE` (Form 3)

Ist ein WiM im System vorhanden, kann ein eigenes Editorfenster durch die folgenden Aufrufe unterdrückt werden[1].

`WD` *`[device/][subdirectories/]filename`* (Form 1)
`WD SC` *`[device/][subdirectories/]filename`* (Form 2)
`WD` (Form 3)
`WORD` *`[device/][subdirectories/]filename`* (Form 1)
`WORD SC` *`[device/][subdirectories/]filename`* (Form 2)
`WORD` (Form 3)

Die folgenden Hinweise sollen diese drei Formen näher erläutern:

- `WE` unterscheidet sich von den beiden anderen Befehlen dadurch, daß RTOS-WORD ein eigenes Fenster für jeden Text öffnet, wenn ein WiM im System vorhanden ist (Vorsicht, falls Sie sich an einem Rechner eingeloggt haben !!!). Das eigene Textfenster läßt sich gemäß Abschnitt 4.4 unterdrücken.
- Fehlt in Form 1 oder Form 2 die Deviceangabe, setzt die Shell das Working-Directory und ein „/“ vor den Übergabeparameter.
- In der Form 1 sind die Dateinamen SI, LO, SC, AD, SZ und PRIO (auch kleingeschrieben) ohne Device oder Subdirectory verboten, da diese Schlüsselworte zur Spezifikation weiterer Übergabeparameter dienen (Fehlermeldung: „`wrong command`“). Als Dateinamen sind sie erlaubt, wenn ein Device und/oder mindestens ein Subdirectory angegeben ist.
- Alle weiteren Übergabeparameter sind in Abschnitt 4.4 erläutert.

[1] Sobald RTOS-UH die Information bereitstellt, ob RTOS-WORD auf einem Terminal oder auf einer Terminalemulation eines WiMs läuft, verhalten sich die Bedienbefehle `WD`, `WORD` genauso wie `WE`

- Das Device, auf dem die Datei liegt, muß rückspulbar sein. Ist das nicht der Fall, erscheint die Fehlermeldung „`Sorry, can't work on this device. Bye, Bye!`“
- In der Form 3 bearbeitet RTOS-WORD das File `/ED/SI`, falls die aufrufende Shell die User-ID „1“ besitzt, sonst `/ED/SI`*x*, wobei *x* die User-ID darstellt.

Im Normalfall startet der Editor bei Beachtung der o. a. Hinweise ohne weitere Fehlermeldungen. Sieht sich RTOS-WORD gezwungen, weitere Fehlermeldungen auszugeben, finden Sie eine Erklärung ab Seite 272.

Ist die Datei vorhanden, öffnet RTOS-WORD diese, sonst erscheint in etwa folgendes Bild:

```
RTOS-UH  W O R D

Version 2.2-e
(c) 1988-1996 IRT, Hannover

Help with ^XH

[N]  edit a new File
[X]  exit before opening
```

Sie müssen nun wählen, ob Sie eine Programm-Datei editieren oder zum Betriebssystem zurückkehren wollen. Entscheiden Sie sich für eine Bearbeitung einer Datei, öffnet RTOS-WORD diese und schreibt in die erste Zeile „`*File was opened by RTOS-WORD.`“

Sollte wider Erwarten statt dem `[N]` ein `ÖNÄ` erscheinen, ist bei dem Terminal oder dem Fenster der falsche ASCII-Satz eingestellt. Dieses stört RTOS-WORD nicht weiter, die eckigen Klammern werden nur anders angezeigt.

Wird eine bereits vorhandene Datei editiert, sind Zeichen gemäß Tabelle 4.1 in der Datei erlaubt.

$20 ... $FF	Normale ASCII-Zeichen
$0D	Carriage return (CR): Zeilenende für RTOS-UH
$0A	(nur hinter CR): Zeilenende für `MS-DOS`/`MS-Windows`
$04	Dateiende (EOT: End of Text) für RTOS-UH
$1A	Dateiende für `MS-DOS`/`MS-WINDOWS`
$09	Tabulator: Wird beim Öffnen durch 3 Blanks ersetzt

Tabelle 4.1: Erlaubte Textzeichen für den Editor RTOS-WORD

4.2.2 Statuszeile, Tabulatorleiste und Fensteraufbau

Hat RTOS-WORD eine Datei geöffnet, ist ab Bildschirmzeile 3 der Textanfang[2] dargestellt. In Bildschirmzeile 1 befindet sich die Statuszeile, über dem Text eine Tabulatorleiste. In den ersten sieben Spalten der Textzeilen steht im Normalfall eine nicht zur Datei gehörende Zeilennummer. Ihre Bedeutung ist hinter den Hinweisen zur Tabelle 4.2 erklärt.

Die Statuszeile soll an Hand des Befehles `WE /HO/TEST` erklärt werden. Sie sieht nach dem Öffnen der Datei in etwa wie folgt aus:

```
line 1 col 1 ins H+ a+ C w- m- /HO/TEST
```

Die einzelnen Elemente sind in Tabelle 4.2 erklärt. Dort sind ggf. Verweise angegeben. Die Tabelle enthält auch die nicht angezeigten Modi.

Hinweise zu Tabelle 4.2:

- Lautet der Defaultstatus einmal nicht `ins H+ a+ w- m-`, ist ein Konfigurationsmodul geladen. Die Erzeugung und Parametrierung eines solchen Modules ist im Abschnitt 4.8 ab Seite 264 erläutert.
- RTOS-WORD erkennt automatisch, ob die Datei `MS-DOS/MS-Windows` kompatibel ist. Solange der Nutzer keine Änderung erzwingt, wird der Status beibehalten.
- UNIX-kompatible Dateien werden nicht unterstützt.
- Funktioniert die inverse, bei einigen Terminals auch halbhelle, Schrift nicht, muß ein Konfigurationsmodul erstellt werden (siehe Abschnitt 4.8, Seite 264).

Im Normalfall (kein Konfigurationsmodul) sind neben den Textzeilen sieben Spalten für die „logische Zeilennummer“ reserviert. Solange Sie keine Zeilen hinzufügen oder entfernen, stimmt die physikalische Zeilennummer immer mit der logischen überein. Die logische Zeilennummer ist beim Beheben von Übersetzerfehlern sehr wichtig, da das Einfügen von Zeilen die Zuordnung von Text zu logischer Zeilennummer nicht ändert. Dazu ein Beispiel: Drücken Sie „^XR“[3], um den Cursor an den Anfang des Textes zu positionieren, und anschließend „^M“, um eine Zeile einzufügen. Die logische Zeile 1 bleibt dem Text zugeordnet, die physikalische Zeilennummer ändert sich. Dadurch können Sie Zeilen beliebig hinzufügen und entfernen und trotzdem durch den Befehl „logische Zeilennummer anspringen“ immer die vom Übersetzer angezeigten Fehlerzeile anspringen (Solange Sie die Zeile nicht entfernt haben).

[2] Ausnahme: Sie benutzen das Hilfesystem (siehe Seite 250).

[3] Das Zeichen „^“ vor einem anderen bedeutet, daß Sie gleichzeitig die Control-Taste, auch CTRL- und STRG-Taste genannt, und die angegebene Taste drücken sollen.

Status	*Anmerkung*	*Bedeutung*
`line 1`	„line“ invers	Cursor steht in physikalischer Textzeile 1
`col 1`	„col“ invers	Cursor steht in physikalischer Textspalte 1
`ins`	invers	Datei wird im „Einsetzmodus“ bearbeitet (siehe Seite 234). Gegenstück von `rep`.
`rep`	invers	Datei wird im „Überschreibmodus“ bearbeitet (siehe Seite 235). Gegenstück von `ins`.
`H+` / `H-`	„H+“ invers	Blockbefehle sind ein-/ ausgeschaltet (siehe Seite 247).
`a+` / `a-`	„a+“ invers	automatisches Einrücken ist ein-/ausgeschaltet (siehe Seite 233).
`L`	invers	Zeilenende ist CR /LF . (`MS-DOS/-Windows` kompatible Datei) Gegenstück von `C`.
`C`	invers	Zeilenende ist CR . Gegenstück von `L`.
`w+` / `w-`	„w+“ invers	Wortumbruch/kein Wortumbruch am rechten Rand (siehe Seite 234).
`m+` / `m-`	„m+“ invers	Klingelsignal „rechter Rand erreicht“ ein-/ausschalten (siehe Seite 233).
`*`		Benutzer hat Dateiinhalt nach dem Öffnen bzw. dem letzten Abspeichern geändert. Ist der Stern nicht sichtbar, wurde Datei nicht geändert.
`/H0/TEST`	Beispiel	Name der editierten Datei einschließlich Device und Subdirectories. Kann RTOS-WORD den kompletten Pfad nicht darstellen, wird ein Teil aus dem Pfad herausgeschnitten und durch „...“ ersetzt.

Tabelle 4.2: Statuszeilenelemente des Editors RTOS-WORD

Ist eine Zeile länger als der Bildschirm, so wird in der letzten Spalte der Zeile ein invers dargestelltes „+“ gezeigt, um Ihnen zu zeigen, daß diese Zeile über den Bildschirm herausragt.

Unter dem Textfenster wird ggf. der Zeilenpuffer eingeblendet.

Wenn Sie ein Menü aufrufen - z. B. mit „^X“ - so wird der entsprechende Buchstabe in der linken oberen Ecke angezeigt. Wird der zweite Buchstabe sehr schnell nach dem ersten eingegebenen (z. B. Funktionstaste), so unterbleibt die Ausgabe.

Als letztes soll die Tabulatorleiste erklärt werden. Sie sieht wie folgt aus:

```
L------!------!--------------------!--- ... ---------!- ... -R
```

(...) deutet an, daß einige „-“ weggelassen wurden. Es bedeuten:

Zeichen	Erklärung
L	Linker Rand (nicht veränderbar)
-	nicht gesetzter Tabulator (veränderbar: siehe Seite 249)
!	gesetzter Tabulator (veränderbar: siehe Seite 249).
R	Rechter Rand (veränderbar: siehe Seite 249)

Die Veränderung der Tabulator-Leiste bzw. die Benutzung der Tabulator-Taste ist in Unterabschnitt 4.3.8 beschrieben.

4.2.3 Fenster-Elemente im Window-Modus

Im Window-Modus sind alle wichtigen Kommandos über Pull-Down-Menüs ausführbar. Weiterhin können Sie verschiedene Befehle mit der Maustaste absetzen. Folgende Regionen innerhalb eines Fensters unterscheidet RTOS-WORD bei der Benutzung der linken Maustaste:

- Den Text (Cursorpositionierung, Verlassen des Zeilenpuffers).
- Die Statuszeile (Änderung der Betriebsmodi, Zeilen und Spaltenänderung, Textwechsel, Dateinamensänderung (mit rechter Taste)).
- Die Tabulatorleiste (Setzen und Löschen von Tabulatoren sowie Veränderung des rechten Randes).
- Die Spalten links des Textes (Anspringen von Zeilen).
- Die beiden Zeilen unterhalb des Textes (Editieren des Zeilenpuffers, Cursorpositionierung innerhalb des Puffers).

Die Verwaltung des Schließfeldes sowie die Rollbalken einschließlich der Pfeile übernimmt RTOS-WORD.

An dieser Stelle alle Befehle und Mausklicks zu erklären, würde eine doppelte Erläuterung vieler Befehle bedeuten. Probieren Sie einfach aus, was Sie mit der Maus erreichen können.

4.3 Bearbeitung von Texten

4.3.1 Beschreibung der Bedienbefehle

Die Bedienbefehle sind in der Befehlsbeschreibung mittels der folgenden Maske erklärt:

Nr.	*Kurzbeschreibung*	*Taste*

Ausführliche Beschreibung

Die Titelzeile beginnt mit der sogenannten Befehlsnummer (*Nr.*). Sie dient zum schnelleren Auffinden bei Verweisen, da i. a. auf die Befehlsnummer und nicht auf die Seite verwiesen wird. Alle Befehle mit einer ausführlichen Erklärung sind durchnumeriert. Ist bei einem Befehl keine Nummer vergeben worden, ist er an einer anderen Stelle ausführlich erklärt. Der Befehlsnummer folgt die Kurzerklärung des Befehls und anschließend die Tastenkombination, die den Befehl auslöst. Sind mehrere angegeben, kann die in der Erklärung beschriebene Wirkung mit allen Kombinationen erreicht werden. Ein „^“ vor einer Taste bedeutet, daß Sie die CTRL-Taste und die darauf folgende gleichzeitig drücken sollen. Ein ESC bedeutet, daß Sie die ESC-Taste und die darauf folgende hintereinander drücken müssen. Bei angeschlagenen Buchstaben ist es egal, ob Sie Groß- oder Kleinschreibung verwenden.

Bei einigen Befehlen sind drei Tastenkombinationen angegeben, von der die mittlere „...“ lautet. Lesen Sie in diesem Fall die ausführliche Erklärung zu diesem Befehl.

4.3.2 Statusänderungen des Editors

1	**Einsetzmodus ein-/ausschalten**	**^_**	**^-**

Mit diesem Befehl können Sie zwischen dem Einsetz- und dem Überschreibmodus wechseln, wobei „^-“ nur im Window-Modus erlaubt ist. Die Unterschiede zwischen den beiden Modi sind in den Tabellen 4.3 und 4.4 erklärt.
Im Window-Modus können Sie statt Verwendung der Hot-Keys in das Anzeigefeld in der Statuszeile klicken (inverses „inv“ bzw. „rep“), um vom Einsetz- in den Überschreibmodus und umgekehrt zu wechseln.

2	**Randauslösung ein-/ausschalten**	**^OX**

Bei eingeschalteter Randauslösung (inverses „`m+`“ in der Statuszeile) wird beim Erreichen des rechten Randes die Klingel Ihres Terminals ausgelöst, das Zeilenendesignal entfällt allerdings, wenn der Editor im Window-Modus ausgeführt wird. Mit diesem Befehl können bei eingeschalteter Auslösung diese ausschalten („`m-`“ in der Statuszeile) und umgekehrt.
Durch einen Mausklick in das entsprechende Anzeigefeld der Statuszeile können Sie im Window-Modus ebenfalls den Zustand wechseln.

3	**Einrücken ein-/ausschalten**	**^OU**

Bei eingeschaltetem Einsetzmodus und eingeschaltetem Einrückmodus (inverse „`ins`“ und „`a+`“ in der Statuszeile) wird beim Drücken von CR der Cursor unter das erste Nicht-Leerzeichen der vorherigen

Einsetzmodus	
Text	Ein eingegebenes Zeichen wird in den Text eingefügt, solange die zulässige Zeilenlänge nicht überschritten wird.
Cr	Beim Anschlagen des Cr wird die Zeile an der aktuellen Cursorposition beendet. Der eventuelle Rest der Zeile wird in die neu eingefügte nächste Zeile kopiert. Ist automatisches Einrücken aktiviert, werden am Anfang der neuen Zeile so viele Leerzeichen eingefügt, wie in der alten vor dem ersten Nicht-Leerzeichen auch standen.
Tab	Beim Anschlagen der Tab-Taste werden ab der Cursorposition so viele Leerzeichen eingefügt, bis ein Tabulator erreicht wird. Steht rechts des Cursors kein Tabulator, wird am rechten Rand – „R“ in der Tabulatorleiste – ein Cr, in der neu eingerichteten Zeile bis zum Erreichen des ersten Tabulators Leerzeichen eingefügt. Ist überhaupt kein Tabulator gesetzt, werden Leerzeichen bis zum rechten Rand und anschließend ein Cr eingefügt.
Bs	Steht der Cursor in der ersten Spalte und wird die Bs-Taste angeschlagen, wird das Cr aus der vorherigen entfernt und diese beiden Zeilen unter Beachtung der zulässigen Zeilenlänge vereint. Die logische Zeilennummer wird aus der Verwaltung entfernt.
Del	Steht der Cursor beim Drücken der Del-Taste in der letzten Spalte, wird das Cr entfernt und diese Zeile mit der nächsten Zeile unter Beachtung der zulässigen Zeilenlänge vereint. Die evtl. vorhandene logische Zeilennummer der nächsten Zeile wird aus der Verwaltung entfernt.

Tabelle 4.3: Der Einsetzmodus von Rtos-Word

Spalte gesetzt. Mit diesem Befehl wechseln Sie zwischen ein- und ausgeschaltetem „`a-`“ Einrücken.
Durch einen Mausklick in das entsprechende Anzeigefeld der Statuszeile können Sie im Window-Modus ebenfalls den Zustand wechseln.

4	**Wortumbruch ein-/ausschalten**	**^OW**

Bei eingeschaltetem Wortumbruch (inverses „`w+`“ in der Statuszeile) wird beim Eingeben eines Leerzeichens hinter diesem die Zeile beendet, falls es sich hinter dem rechten Rand („R“ in der Tabulatorleiste) befindet. Mit diesem Befehl können Sie bei eingeschaltetem Umbruch diesen ausschalten („`w-`“ in der Statuszeile) und umgekehrt.
Durch einen Mausklick in das entsprechende Anzeigefeld der Statuszeile können Sie im Window-Modus ebenfalls zwischen den beiden Zuständen hin- und herwechseln.
Hinweis: Es ist geplant, in einer der nächsten Versionen den Umbruch

Überschreibmodus	
Text	Ein eingegebenes Zeichen überschreibt das unter dem Cursor stehende, solange das Zeilenende nicht erreicht ist. Steht der Cursor auf der Zeilenendemarkierung, wird das Zeichen vor dieser eingefügt.
Cr	Beim Anschlagen des Cr wird der Cursor auf Spalte 1 der nächsten physikalischen Zeile positioniert. Es wird keine Zeile eingefügt.
Tab	Beim Anschlagen der Tab-Taste wird der Cursor auf den ersten Tabulator rechts der Cursorpostition gesetzt. Steht rechts der aktuellen Position kein Tabulator, wird der Cursor eine Zeile tiefer auf die erste Spalte mit gesetztem Tabulator positioniert. Ist kein Tabulator gesetzt, wird der Cursor in die erste Spalte der nächsten Zeile positioniert.
Bs	Steht der Cursor in der ersten Spalte und wird die Bs-Taste angeschlagen, wird der Cursor hinter das letzte Zeichen der vorherigen Zeile gesetzt.
Del	Steht der Cursor beim Drücken der Del-Taste in der letzten Spalte, wird der Tastenanschlag ignoriert.

Tabelle 4.4: Der Überschreibmodus von Rtos-Word

vor dem Wort durchzuführen. Verlassen Sie sich nicht darauf, daß Rtos-Word den Umbruch hinter das Wort setzt.

4.3.3 Grundlegende Bearbeitung einer Datei

Dieser Abschnitt befaßt sich mit dem Einfügen und Löschen von Text sowie der Cursorbewegung. Zusammen mit Abschnitt 4.3.5 beschreibt er die elementaren Editierfunktionen.

Der Cursor kann am einfachsten mit den Cursortasten über den Text bewegt werden. Rtos-Word läßt es nicht zu, daß der Cursor aus dem Schirm herausscrollt. Im Zweifelsfall wird der bearbeitete Text geblättert. Zur Beschleunigung der täglichen Arbeit gibt es spezielle Tastenkombinationen (und im Window-Mode Menüoptionen), mit denen der Cursor positioniert werden kann.

	Tabulator anlaufen	**Tab**

Der Cursor wird auf den nächsten Tabulator gesetzt. Das genaue Verhalten ist wegen der Unterschiede im Einsetz- und Überschreibmodus in den Tabellen 4.3 und 4.4 ab Seite 234 erklärt.

5	**Zeichen löschen**	**Del**	**Esc←**	**EscW**

Löscht das Zeichen der aktuellen Cusor-Position. Steht der Cursor am Zeilenende und ist der Einsetzmodus eingeschaltet, so wird das Zei-

lenende gelöscht und die beiden Zeilen werden unter Berücksichtigung der zulässigen Zeilenlänge vereinigt.

6	Leerzeile einfügen	EscL	Esc↓	EscE

Dieser Befehl setzt vor die Zeile, in der der Cursor steht, eine Leerzeile ein und positioniert den Cursor auf Spalte 1 (also das Zeilenende) dieser Leerzeile.

7	Leerzeichen einfügen	EscQ	Esc→

Dieser Befehl fügt unter dem Cursor ein Leerzeichen unabhängig vom Überschreib- oder Einfügemodus ein.

8	Sonderzeichen eingeben	^PA	...	^PW

Nach der Eingabe von „^P“ können Sie ein Sonderzeichen in den Text eingeben. Um das Zeichen einzugeben, können Sie entweder Tabelle 4.11 verwenden, um die Zuordnung zwischen Buchstaben und Sonderzeichen zu erhalten oder Sie addieren auf den ASCII-Wert ihres Zeichens $40 und geben den zu diesem Wert korrespondierenden Buchstaben ein. Zulässig sind die Buchstaben „A“ bis „W“ mit Ausnahme des „M“. Beispiele: Mit „^PD“ können Sie also ein EOT eingeben. Steht es in der ersten Spalte, können Sie hier das neue Dateiende (mit allen Konsequenzen) setzen. Mit „^PL“ (Form-Feed) können Sie einen Drucker dazu bewegen, hier einen Seitenvorschub zu forcieren.

9	Cursor nach links	^H	←	EscD

Der Cursor wird um ein Zeichen nach links bewegt. Steht der Cursor am Anfang einer Zeile, so wird er an das Ende der vorherigen Zeile positioniert.

10	Cursor nach rechts	^L	→	EscC

Der Cursor rückt um ein Zeichen nach rechts. Steht er am Ende einer Zeile, springt er an den Anfang der nächsten Zeile.

11	Cursor nach unten	^J	^V	↓	EscB

Der Cursor springt um eine Zeile nach unten. Wenn möglich, bleibt er in derselben Spalte. Ist die nächste Zeile kürzer, so springt der Cursor an das Ende der Zeile.

12	Cursor nach oben	^K	↑	EscA

Der Cursor springt um eine Zeile nach oben. Wenn möglich, bleibt er in derselben Spalte, ansonsten springt er an das Ende der Zeile.

13	Cursor Wort links	^A

Der Cursor wird an den Anfang des Wortes gesetzt, auf dem er steht. Die Positionierung erfolgt über die Suche des ersten Wortendes links vom Cursor, verbunden mit einer Positionierung hinter dem Wortende. Das Wortende ist in Befehl 23 erklärt.

14	Cursor Wort rechts	^F

Der Cursor wird auf den nächsten Wortanfang gesetzt. Die Positionierung erfolgt über die Suche des ersten Wortendes rechts vom Cursor, verbunden mit einer Positionierung hinter dem Wortende. Das Wortende ist in Befehl 23 erklärt.

Die folgenden 4 Befehle dienen der Positionierung des Cursors an die Ränder der gerade aufgeblätterten Seite. Die Buchstaben S, D, X und E bilden eine Art Kreuz auf Ihrer Tastatur. An diesem können Sie sich orientieren, wenn Sie den Cursor positonieren wollen.

15	Cursor an Zeilenanfang	^XS

Der Cursor wird an den Anfang der Zeile positioniert.

16	Cursor an Zeilenende	^XD

Der Cursor wird hinter das letzte eingegebene Zeichen der Zeile gesetzt.

17	Cursor an oberen Bildschirmrand	^XE

Bewegt den Cursor in die zweite Bildschirmzeile. Der Text wird nicht geblättert. Die Spaltenposition wird, wenn möglich, beibehalten.

18	Cursor an unteren Bildschirmrand	^XX

Positioniert den Cursor in die vorletzte Bildschirmzeile. Der Text wird nicht geblättert. Die Spaltenposition wird, wenn möglich, beibehalten.

19	Cursor an den Dateianfang	^XR

Der Textanfang wird aufgeblättert und der Cursor an den linken Rand der ersten physikalischen Zeile gesetzt.

20	Cursor an das Dateiende	^XC

Das Textende wird aufgeblättert und der Cursor auf das Dateiende gesetzt.

21	Zeile umbrechen (Hartes Cr)	^N

Es wird an der aktuellen Cursorposition ein Zeilenende eingefügt. Der eventuelle Zeilenrest wird unter Berücksichtigung der Einrückoption in die neu eingefügte nächste Zeile kopiert. Der Cursor behält seine relative Position zum Zeilenrest: Nach Anschlagen der Taste „Cursor rechts“ steht der Cursor auf dem ersten Zeichen des Zeilenrests.

22	Zeichen links vom Cursor löschen	^G	Bs

Es wird das Zeichen links vom Cursor gelöscht. Steht der Cursor in Spalte 1 und ist der Einsetzmodus eingeschaltet, so wird das Zeilenende der vorherigen Zeile gelöscht und die beiden Zeilen werden unter Berücksichtigung der zulässigen Zeilenlänge vereinigt. Dieser Befehl hat die gleiche Wirkung wie die Bs-Taste.

23	Wort ab Cursor bis Wortende löschen	^T

Dieser Befehl löscht ab der Cursorposition den Rest des Wortes und alle folgenden Leerzeichen. Steht der Cursor am Zeilenende, wird im Einsetzmodus das Zeilenende gelöscht, die Zeilen unter Berücksichtigung der zulässigen Zeilenlänge vereinigt und alle Leerzeichen, die links des ersten „Nicht-Leerzeichens“, liegen, gelöscht. Ein Wortende ist in RTOS-WORD durch eins der folgenden Zeichen definiert: ␣ , ; . : ! ? = - * + / ( ) ’ { } \ < > CR.

24	Zeile, in der Cursor steht, löschen	^Y	Esc↑

Es wird die gesamte Zeile, in der der Cursor steht, gelöscht. War eine logische Zeilennummer für diese Zeile vergeben, wird diese aus der Verwaltung entfernt und ist nicht mehr mit „EscZ“ (siehe Nr. 33) erreichbar. Alle nachfolgenden Zeilen rücken um eine Zeile nach oben. Die Cursorposition innerhalb des Bildschirms bleibt, soweit möglich, erhalten.

25	Zeile ab Cursor bis Zeilenende löschen	^XY

Löscht ab der Cursorposition bis zum Zeilenende alle Zeichen. Die Zeilenendemarkierung wird nicht gelöscht, auch wenn der Cursor auf ihr steht.

26	Zeile links vom Cursor löschen	^XZ

Löscht links vom Cursor bis zur Spalte 1 alle Zeichen. Steht der Cursor in Spalte 1, hat dieser Befehl keinerlei Wirkung.

	Block löschen	**^EY**

Befehl ist unter Nummer 55 erklärt.

27	**Löschen rückgängig machen**	**^U**

Haben Sie eine Löschoperation irrtümlich ausgeführt, können Sie mit sofort danach gedrücktem „^U“ dieses rückgängig machen. Eine Ausnahme bildet der Befehl „Block löschen“: Die „Undo“-Funktion ist hier „^EM“ (siehe Befehl 57). Wichtig ist, daß Sie „^U“ sofort eingeben, sonst kann RTOS-WORD Ihren Wunsch nicht erfüllen und gibt statt dessen eine Fehlermeldung aus. Lediglich der Aufruf eines Submenüs und Abbruch mit einem Leerzeichen erlaubt weiterhin das „Undo“.

28	**Letzte gelöschte Zeile einfügen**	**^XU**

Haben Sie eine gesamte Zeile z. B. mit „^Y“ gelöscht, setzt dieser Befehl die gelöschte Zeile vor die, in der z. Zt. der Cursor steht. Dieser wird in die erste Spalte der eingesetzten Zeile positioniert.

4.3.4 Befehle zum Blättern

	Cursor auf Marke positionieren	**^X*0-9***

Befehl ist im Unterabschnitt 4.3.9 als Nr. 70 erklärt.

29	**Cursor zum Blockanfang bewegen**	**^XB**

Bewegt den Cursor auf den mit „^EB“ markierten Blockanfang. Der Text wird, falls nötig, so geblättert, daß der Cursor sichtbar bleibt. Die Blockbefehle dürfen ausgeschaltet sein. Der Block darf ungültig sein (Blockende vor Blockanfang oder gar nicht definiert).

30	**Cursor zum Blockende bewegen**	**^XK**

Bewegt den Cursor auf das mit „^EK“ markierte Blockende. Der Text wird, falls nötig, so geblättert, daß der Cursor sichtbar bleibt. Die Blockbefehle dürfen ausgeschaltet sein. Der Block darf ungültig sein (Blockanfang hinter Blockende oder gar nicht definiert).

31	Cursor auf Start von Suchen/Ersetzen	^XV

Der Cursor wird an die Stelle gesetzt, wo das letzte „Suchen und/oder Ersetzen“ gestartet wurde. Der Text wird, falls nötig, geblättert, so daß der Cursor sichtbar bleibt.

32	Cursor auf physikalische Zeile	EscY

Der Cursor wird, falls möglich, auf die von Ihnen eingegebene Zeile positioniert. Der Versuch scheitert, wenn die Zeilennummer zu groß ist. Der Text wird, falls nötig, so geblättert, daß der Cursor sichtbar bleibt.
Durch einen Mausklick in die Zeilenanzeige der Statuszeile können Sie diesen Befehl ebenfalls ausführen.

33	Cursor auf logische Zeile	EscZ

Der Cursor wird, falls möglich, auf die von Ihnen eingegebene Zeile positioniert. Der Versuch scheitert, wenn die Zeilennummer zu groß ist oder Sie die Zeile entfernt haben. Der Text wird, falls nötig, so geblättert, daß der Cursor sichtbar bleibt.
Diesen Befehl können Sie auch durch einen Mausklick im Bereich der Zeilennumerierung des Textfensters ausführen.

34	Cursor auf physikalische Spalte setzen	EscG

Der Cursor wird, falls möglich, auf die von Ihnen eingegebene Textspalte positioniert. Ist die von Ihnen eingegebene Zahl zu groß, wird der Cursor an das Zeilenende gesetzt. Der Text wird, falls nötig, seitlich so geblättert, daß der Cursor sichtbar bleibt.
Im Window-Modus können Sie statt Verwendung des Hot-Keys in das Spaltenanzeigefeld der Statuszeile klicken.

Mit den beiden nächsten Befehlen kann der Text „feinpositioniert“ werden.

35	Text abwärts scrollen	^Z

Scrollt den Bildschirm um eine Zeile nach oben. Die oberste Bildschirmzeile verschwindet und eine neue Zeile wird am unteren Bildschirmrand sichtbar. Der Cursor bleibt, sofern möglich, auf derselben Stelle im Text stehen, wird also auch um eine Zeile aufwärts bewegt. Steht er in der zweiten Bildschirmzeile, wird er nicht gescrollt.
Im Window-Modus können Sie statt Verwendung des Hot-Keys den unteren Rollpfeil anklicken.

36	**Text aufwärts scrollen**	**ˆW**

Scrollt den Bildschirm um eine Zeile nach unten. Die unterste Bildschirmzeile verschwindet und eine neue Zeile wird am oberen Bildschirmrand sichtbar. Der Cursor bleibt, sofern möglich, auf derselben Stelle im Text stehen, wird also auch um eine Zeile abwärts bewegt. Steht er auf der vorletzten Zeile, wird er nicht gescrollt.
Im Window-Modus können Sie statt Verwendung des Hot-Keys den oberen Rollpfeil anklicken.

37	**Folgende Textseite aufblättern**	**ˆC**

Der Text wird eine Bildschirmseite weitergeblättert. Nach dem Blättern wird die sich an die z. Zt. unterste Zeile anschließende Zeile die oberste sein. Der Cursor bleibt, falls möglich, in derselben Bildschirmzeile und -spalte.
Im Window-Modus können Sie in den unteren Rollbalken klicken, um eine Seite vorwärts zu blättern.

38	**Textseite zurückblättern**	**ˆR**

Der Text wird eine Bildschirmseite zurückgeblättert. Nach dem Blättern wird die vor der z. Zt. obersten Zeile liegende Zeile die unterste sein. Der Cursor bleibt, falls möglich, in derselben Bildschirmzeile und -spalte.
Im Window-Modus können Sie in den oberen Rollbalken klicken, um eine Seite rückwärts zu blättern.

39	**Halbe Seite vorwärtsblättern**	**EscF**

Der Text wird um eine halbe Seite in Richtung Dateiende weitergeblättert. Der Cursor bleibt, falls möglich, in derselben Bildschirmzeile und -spalte.

4.3.5 Dateibefehle

Bei einigen Dateibefehlen erhalten Sie von RTOS-WORD eine Eingabeaufforderung. Den von Ihnen eingegebenen String können Sie mit den in Tabelle 4.5 angegebenen Tasten editieren.

Betrachten wir folgendes Beispiel: Sie wollen als zweites zu bearbeitendes Dokument „/H0/TEX/WF.TEX“ öffnen. Nach der Eingabe von „ˆEO“ (siehe Befehl 42) fordert Sie RTOS-WORD zur Eingabe des Dateinamens auf. Drücken Sie „ˆY“, um den vorgeschlagenen Pfad zu löschen, und geben Sie anschließend „/H0/TEX/WF.TEX“ ein. Nun stellen Sie fest, daß das File tatsächlich

Zeichen	Bedeutung
Bs	löscht das letzte Zeichen.
^D	restauriert das letzte Zeichen
^Y	löscht die gesamte Eingabe
^R	restauriert die gesamte Eingabe
^U	bricht das Kommando ab

Tabelle 4.5: Korrektur von Dateinamen bei RTOS-WORD

„WE.TEX“ heißt. Sie können die Eingabe korrigieren, indem Sie fünfmal die Bs-Taste drücken – der Cursor steht dann hinter dem „W“ –, anschließend ein „E“ und dann „^R“. Nun müßten Sie den korrekten Pfad vor sich sehen können. Drücken Sie nun „^U“, um das ganze abzubrechen.

40	**Editor unterbrechen**	**^EU**

Im Terminalmodus suspendiert sich der Editor und gibt die Shell frei, damit Sie nach einem „^A“ wie gewohnt Befehle an RTOS-UH absetzen können. Den Tasknamen können Sie der Meldung „*taskname* `suspended waiting`“ entnehmen. Durch ein Fortsetzen der Task mit „C *taskname*“ können sie mit dem Editieren fortfahren.
Im Window-Modus wird das aktuelle Fenster iconisiert[4]. Hier können sie weiterarbeiten, indem Sie ein Textfenster des Editors öffnen.

41	**Speichern und Verlassen**	**^EX**	**EscX**

RTOS-WORD speichert die Datei, falls Sie nach dem letzten Öffnen bzw. Sichern geändert wurde. Anschließend entfernt er sie aus der Editorverwaltung. Bearbeitet die Editor-Subtask keine weitere Datei, terminiert und entlädt sie sich. Anderenfalls wird auf die folgende Datei umgeschaltet.
Speichern bedeutet beim Bearbeiten einer Datei, die nicht als `/ED`-Datei geöffnet wurde, daß die auf der Datenstation `/ED` abgelegte Arbeitskopie mit allen erfolgten Änderungen zurückgeschrieben wird.
Wurde die Datei neu angelegt (`*File was opened by RTOS-WORD.`) und nicht geändert, wird sie gelöscht.
Wenn beim Zurückschreiben der Datei ein Schreibfehler auftritt, z. B. das Zieldevice eine schreibgeschützte Diskette ist, bricht RTOS-WORD den Speichervorgang ab. Die temporäre Arbeitsdatei im RAM Ihres Rechners wird nicht gelöscht. Den Namen dieser Datei teilt Ihnen RTOS-WORD bei der Fehlermeldung mit. Mit dieser Datei können Sie nun machen, was Sie wollen: Erneut editieren, löschen, kopieren ...

[4] In einer späteren Version werden alle Texte iconisiert

42	**Weitere Datei öffnen**	^EO

Geben Sie nach der Eingabeaufforderung den Dateinamen an. Ist die Datei nicht vorhanden, kann sie neu angelegt werden. Danach können Sie diese Datei bearbeiten. Im Window-Modus wird für den Text ein eigenes Fenster eingerichtet. Achtung: Bei Dateien, die im Netzwerk oder auf einem Massenspeicher liegen, kann die gleiche Datei mehrmals editiert werden. Solange nur eine der Arbeitskopien geändert wird, kann dieses manchmal von Vorteil sein. RTOS-WORD kann beim Speichern allerdings nur die Änderungen aus einer Arbeitskopie übernehmen! Nämlich die der zuletzt mit Befehl Nr. 41 verlassenen Arbeitkopie.
Vorsicht: Ein zweifaches Öffnen einer `/ED`-Datei kann die Datei sofort und endgültig zerstören!!

43	**Arbeitstext wechseln**	^EN

RTOS-WORD schaltet auf den folgenden Text um. Im Window-Modus bekommt das entsprechende Fenster den Input-Focus. War es iconisiert, wird das Fenster geöffnet.

44	**Verlassen mit Namensänderung**	^EZ

Mit „^EZ“ können Sie das Originalfile mit Originalnamen unverändert lassen und Ihren bearbeiteten Text unter anderen Namen abspeichern. Eine Namensänderung (unter Beibehaltung des Originalfiles mit Originalnamen) ist bei `/ED`-Dateien nur möglich, wenn Pfad und Name mit „^EL“ (siehe Befehl 48) erzeugt wurde.
Weiterhin besteht die Möglichkeit, Dateien vom RTOS-UH-Format ins MS-DOS-Format und umgekehrt zu konvertieren (siehe auch Tabelle 4.1).
Sie können die Datei auch an eine schon bestehende anhängen. Dabei darf am Dateiende, an die Sie Ihren Text anhängen wollen, kein EOT stehen, da bei RTOS-UH die Dateiendekennung EOT zur Datei gehört und RTOS-WORD den Text hinter die Endekennung hängt! Sie ahnen es schon: Nach dem ersten Kopieren oder Editieren ist der angehängte Text verschwunden. Tip: Entfernen läßt sich das EOT durch Editieren des Files und Entfernen des „@“ am Dateiende.
Weiterhin haben Sie die Option, alle Leerzeichen, denen in einer Zeile nur noch weitere Leerzeichen und das CR folgen, aus Ihrem Text zu entfernen.
Alle optionalen Parameter sind in Tabelle 4.6 erläutert. Sie sind in beliebiger Reihenfolge nach einem „-“ an den Dateinamen anzuhängen.
Beispiel: Ein Verlassen einer Datei mit „/H0/SYS/TEST -adl“ hängt Ihren Text an die Datei „/H0/SYS/TEST“ an. Alle Leerzeichen am

Parameter	Bedeutung
c	Abspeichern im RTOS-UH-Format
l	Abspeichern im MS-DOS-Format
a	An eine bestehende Datei anhängen
d	Endende Leerzeichen löschen

Tabelle 4.6: Parameter von RTOS-WORD beim Verlassen einer Datei

Zeilenende werden entfernt. Die Datei ist MS-DOS kompatibel (siehe auch Anmerkung zu Befehl 58).

45	Verlassen ohne Speichern	^EQ

RTOS-WORD entfernt den Text aus seiner Verwaltung, ohne ihn abzuspeichern. In der Datei steht der Originaltext bzw. der zuletzt mit „^ES“ abgespeicherte Text. Dieses ist bei `/ED`-Dateien nur möglich, wenn Pfad und Name mit „^EL“ (siehe Befehl 48) erzeugt wurde. Sie können den Text nur nach einer Sicherheitsabfrage („`This File will not be saved! (y/n)`“) und Anschlagen des „Y“ verlassen. Wurde die Datei neu angelegt („`*File was openened by RTOS-WORD.`“), wird die Datei gelöscht.

46	Speichern und weiterarbeiten	^ES

Dieser Befehl speichert Ihren Text und löscht alle Marken und die Änderungskennung „`*`“ in der Statuszeile, falls der Text nach dem Öffnen bzw. letzten Speichern geändert wurde. Nach dem Speichern können Sie mit dem Editieren fortfahren. Während des Speichervorganges erscheint die Meldung „`Please wait: Saving!`“ Der Befehl kann benutzt werden, um bei einer längeren Editor-Sitzung die Datei zu sichern, ohne RTOS-WORD verlassen zu müssen. Im Window-Modus können Sie Quelltexte bequem compilieren/assemblieren, ohne RTOS-WORD zu verlassen! Dieser Befehl ist bei `/ED`-Dateien nur möglich, wenn Pfad und Name mit „^EL“ (siehe Befehl 48) erzeugt wurde.

47	Automatisches Sichern	^ED

RTOS-WORD kann Texte automatisch nach einem von Ihnen vorgegebenen Zeitintervall zyklisch sichern. Geben Sie das Sicherungsintervall in Minuten nach der Eingabeaufforderung ein. Die Sicherung erfolgt entsprechend dem Befehl „^ES“ (Nr. 46). Zum Beenden des automatischen Sicherns müssen sie diesen Befehl mit dem Zeitintervall „0“ Minuten verwenden.

48	Neuer Dateiname	^EL

Sie können Device und kompletten Pfad inklusive der optionalen Parameter des Befehles „^EZ“ (Nr. 44), die beim Speichern berücksichtigt werden, eingeben. Eine automatisch erzeugte Sicherung (siehe „^ED“, Befehl Nr. 47) übernimmt ebenfalls den neuen Dateinamen und die Parameter. Der Parameter „a“ bewirkt allerdings bei jedem Speichern ein Anhängen an das File.
Durch einen Mausklick mit der rechten Maustaste in den in der Statuszeile angezeigten Dateinamen können Sie diesen Befehl ebenfalls ausführen.

49	Datei löschen	^EJ

Die angegebene Datei wird, falls im Dateisystem vorhanden, gelöscht. Es findet keine Überprüfung des eingegebenen Dateinamens statt, so daß auch Dateien mit Sonderzeichen, die über die RTOS-UH-Shell nicht eingegeben werden können, gelöscht werden.

4.3.6 Blockbefehle

Ein Block ist durch eine Anfangs- und Endmarke definiert, wobei die Anfangsvor der Endemarke liegen muß. Die Länge ist nicht begrenzt. Alle Blockoperationen werden über den Blockpuffer ausgeführt. Er enthält immer den zuletzt bearbeiteten Block. (Sehr vorteilhaft beim irrtümlichen Löschen eines Blockes!)

Die Größe des Blockpuffers paßt RTOS-WORD während des Editierens automatisch an. Kann ein Block nicht bearbeitet werden, weil der noch verbliebene freie Speicherplatz nicht ausreicht, müssen Sie entweder den freien Speicherplatz erhöhen oder den Block in mehrere kleine aufteilen.

Bei der gleichzeitigen Bearbeitung mehrerer Dateien können Blöcke zwischen den Dateien ausgetauscht werden (siehe Befehle Nr. 56 und 57).

50	Blockanfang markieren	^EB

Der Blockanfang wird auf die aktuelle Cursorposition verlegt. Waren die Blockbefehle ausgeschaltet, werden sie wieder zugelassen und die Statuszeile angepaßt.

51	Blockende markieren	^EK

Das Blockende wird auf die aktuelle Cursorposition verlegt. Waren die Blockbefehle ausgeschaltet, werden sie wieder zugelassen und die Statuszeile angepaßt.

52	**Block kopieren**	^EC

Der Block wird im Blockpuffer abgelegt und anschließend aus dem Blockpuffer an die aktuelle Cursorposition kopiert.

53	**Block verschieben**	^EV

Der Block wird zuerst im Blockpuffer abgelegt, danach aus der Datei entfernt und anschließend aus dem Blockpuffer an der aktuellen Cursorposition wieder eingefügt.

54	**Block einrücken**	^EI

Mit diesem Befehl kann die Einrücktiefe der einzelnen Zeilen des Blockes verändert werden. Geben Sie nach Absetzen des Befehles die Anzahl der Spalten ein, um die die Zeilen des Blockes zu verschieben sind. Bei einer positiven Zahl wird der Block nach rechts geschoben (eingerückt), mit einer negativen können Sie den Block nach links schieben (Es werden aber nur Leerzeichen entfernt: Steht z.B. in der 5. Spalte Text, führt eine Verschiebung um „-10“ nur zum Löschen der vier Leerzeichen). In einem Schritt können Sie maximal eine Verschiebung um 50 Zeichen vornehmen.

55	**Block aus Text entfernen**	^EY

Der Block wird zuerst im Blockpuffer abgelegt, anschließend aus der Datei entfernt. Ein irrtümliches Löschen läßt sich mit „^EM“ (Nr. 57) rückgängig machen. Achten Sie auf die Cursorposition.

56	**Block in Puffer kopieren**	^EG

Der Block wird im Blockpuffer abgelegt und kann zu einem späteren Zeitpunkt, der aber noch innerhalb der Editorsitzung liegen muß, aus dem Blockpuffer an eine beliebige Stelle kopiert werden.

57	**Block aus Blockpuffer kopieren**	^EM

Der Blockpuffer wird an die aktuelle Cursorposition kopiert.

58	**Block in Datei schreiben**	^EW

Der Block wird im Blockpuffer abgelegt und – je nach Parametrierung – anschließend in eine Datei geschrieben oder an eine Datei angehängt. Steht das Zieldevice in der Stellung „Nach endendem Cr ein Line-Feed anfügen“, beendet Rtos-Word jede Zeile zusätzlich mit einem Line-Feed. Der gerade bearbeitete Text bleibt unverändert. Nach Absetzen

des Befehles müssen Sie die Zieldatei angeben, danach, noch in der gleichen Eingabezeile, eventuelle Parameter:

- „-C“ speichert Block im RTOS-UH-Format. Steht Zieldevice im „Line-Feed“ Modus, wird „-C“ ignoriert.
- „-L“ speichert Block im MS-DOS-Format.
- „-A“ hängt Block an Datei an.
- Die Kombinationen „-CA“ und „-LA“ sind erlaubt.

59	**Block aus Datei lesen**	**ˆER**

Geben Sie nach Absetzen des Befehles den Dateinamen ein. Der Dateiinhalt wird an die Cursorpositon kopiert. Die Quelldatei kann beliebiger Herkunft sein, muß also nicht mit „ˆEW“ erzeugt worden sein. Die Blockmarkierungen und der Blockpuffer sind nach dem Einlesen gelöscht. Waren die Blockbefehle ausgeschaltet, sind sie nun wieder zugelassen.

60	**Blockbefehle ein-/ausschalten**	**ˆEH**

Waren die Blockbefehle nicht zugelassen, sind sie nun wieder erlaubt. Ein evtl. vorhandener Block wird invertiert angezeigt. Im anderen Fall werden sie ausgeschaltet und ein evtl. vorhandener Block in Normaltext dargestellt. Alle Blockbefehle, die sich nicht implizit selbst einschalten (alle außer „ˆEB“, „ˆEK“, „ˆER“) werden kommentarlos ignoriert.

4.3.7 Befehle für den Zeilenpuffer

Der Zeilenpuffer erlaubt ein schnelles Kopieren und Verschieben einzelner Zeilen. Eine Suchfunktion ist ebenfalls implementiert. Da der Puffer seitlich nicht gescrollt wird, ist die Anzahl editierbarer Zeichen auf die dargestellte Spaltenzahl beschränkt, auch wenn die Zeile selbst länger ist. Das Editieren erfolgt immer im Überschreibmodus.

61	**Zeilenrest in Zeilenpuffer kopieren**	**EscO**

Ab der aktuellen Cursorposition wird der Zeilenrest linksbündig in den Zeilenpuffer kopiert. Steht der Cursor im Zeilenpuffer, führt dieser Befehl zu einer Fehlermeldung.

62	Zeilenpuffer in Text einfügen	EscI

Über der Zeile mit dem Cursor wird eine Leerzeile eingefügt und der Zeilenpuffer in diese kopiert. Steht der Cursor im Zeilenpuffer, führt dieser Befehl zu einer Fehlermeldung.

63	Nach Pufferinhalt suchen	EscS

Steht der Cursor im Text, wird ab der Cursorposition nach dem Inhalt des Zeilenpuffers gesucht. Steht der Cursor im Zeilenpuffer, wird ab der Stelle gesucht, auf der der Cusor vor dem Befehl „Zeilenpuffer editieren“ (Nr. 64) stand.

64	Zeilenpuffer editieren	EscK

Dieser Befehl positioniert den Cursor in Spalte 1 des Zeilenpuffers, der ggf. neu dargestellt wird (falls er unsichtbar war). Anschließend können Sie den Zeilenpuffer im Überschreibmodus editieren. Die erlaubten Befehle sind in der hinter diesem Befehl folgenden Tabelle zusammengefaßt.
Um den Puffer zu verlassen, können Sie „↑“ (setzt Cursor an Position vor dem Editieren), „^XR“ (19), „^XC“ (20), „Esc S“ (63), „Esc Y“ (32) und „Esc Z“ (33) verwenden. „^C“ (37) und „^R“ (38) blättern zwar den Text vor/zurück, der Cursor bleibt allerdings im Zeilenpuffer.
Durch einen Mausklick in den Bereich unterhalb des Textes können Sie diesen Befehl ebenfalls ausführen. Bearbeiten Sie den Zeilenpuffer schon, bewirkt der Klick eine Änderung der Cursorspalte.

Bef.	Nr.	Bef.	Nr.	Bef.	Nr.	Bef.	Nr.
←	9	→	10	^A	13	^F	14
^XS	15	^XD	16	^G	22	Del	5
^Y	24	^XY	25	^XZ	26	Esc ←	5
Esc →	7	Esc ↑	24	EscG	34		

4.3.8 Tabulatorbefehle

Beim Öffnen einer Datei expandiert Rtos-Word jeweils ein Tabulatorzeichen durch drei Leerzeichen. Beim Drücken der Tabulatortaste werden im Einsetzmodus Leerzeichen eingefügt. Beim Speichern eines Textes werden beim Öffnen expandierte Tabulatoren nicht zurückverwandelt. Die Wirkung der Tabulatortaste ist, getrennt nach Einsetz- und Überschreibmodus, in den Tabellen 4.3 und 4.4 erklärt.

Eine Besonderheit besteht bei der Bearbeitung von `/ED`-Dateien: Die Tabulatorleiste wird beim erneuten Editieren nicht neu aufgebaut. Die Leiste, die beim Verlassen gültig war, finden Sie wieder vor.

65	**Tabulator anlaufen**	**^I**

Dieser Befehl ist mit dem Drücken der Tabulatortaste identisch. Diese ist, getrennt nach Einsetz- und Überschreibmodus, in den Tabellen 4.3 und 4.4 erklärt.

66	**Tabulator setzen**	**^OI**

Geben Sie nach der Eingabeaufforderung die Spalte zwischen linkem und rechtem Rand an, in der der Tabulator gesetzt werden soll. Der gesetzte Tabulator wird durch ein „!“ in der Tabulatorleiste angezeigt. In der dem Befehl folgenden Tabelle sind die zulässigen Sonderzeichen erläutert.
Durch einen Mausklick in die Tabulatorleiste können Sie im Window-Modus ebenfalls einen Tabulator setzen, wenn in der ausgewählten Spalte noch kein Tabulator gesetzt ist.

Zeichen	Bedeutung
Esc	Der Tabulator wird in die aktuelle Cursorspalte gesetzt
A	Neue Tabulatorleiste für Assembler-Quelltexte
P	Neue Tabulatorleiste für PEARL- und C-Quelltexte

67	**Tabulator löschen**	**^ON**

Geben Sie nach der Eingabeaufforderung die Spalte zwischen linkem und rechtem Rand an, in der der Tabulator gelöscht werden soll. Der gelöschte Tabulator wird durch ein „-“ in der Tabulatorleiste angezeigt. In der nachfolgenden Tabelle sind die zulässigen Sonderzeichen aufgeführt.
Durch einen Mausklick in die Tabulatorleiste können Sie im Window-Modus ebenfalls einen Tabulator löschen, wenn in der angeklickten Spalte ein Tabulator gesetzt ist.

Zeichen	Bedeutung
Esc	Der Tabulator der aktuellen Cursorspalte wird gelöscht
A	Alle Tabulatoren werden gelöscht

68	**Rechten Rand setzen**	**^OR**

Geben Sie nach der Eingabeaufforderung eine Spalte zwischen 2 und einschließlich 231 an, die der neue rechte Rand werden soll. Wird der

neue rechte Rand kleiner als der linke, werden alle Tabulatoren rechts davon gelöscht und bleiben es auch beim nächsten Vegrößern des rechten Randes. Beim Drücken der Esc-Taste wird die aktuelle Cursorposition gelöscht.
Im Window-Modus können Sie einfach das „R“ in der Tabulatorleiste anklicken, um den rechten Rand zu verändern.

4.3.9 Marken

Die 10 Marken dienen dem schnellen Anspringen von Textstellen. Der Nutzer kann sie frei im Text mit „ˆE*x*“(Befehl Nr. 69) setzen, wobei *x* eine Ziffer ist. Nach dem Verlassen des Textes sind alle Marken gelöscht und müssen beim erneuten Aufrufen der Datei auch neu gesetzt werden.

69	**Marke setzen**	**ˆE0**	**...**	**ˆE9**

An der aktuellen Cursorposition wird eine Marke gesetzt. Steht der Cursor direkt hinter einer Marke und wird das Kommando „ˆE*x*“ erneut aufgerufen, wird die Marke versteckt, d. h. sie ist nicht mehr sichtbar, kann aber immer noch angesprungen werden. Beispiel: „ˆE3 ˆE3“ setzt und versteckt die Marke 3 links neben der aktuellen Cursorposition.

70	**Cursor auf Marke setzen**	**ˆX0**	**...**	**ˆX9**

Der Cursor wird auf die entsprechende mit „ˆE*0-9*“ (siehe Nr. 69) definierte Marke gesetzt. Der Text wird, falls so nötig, geblättert, daß der Cursor sichtbar bleibt. Die Marke wird nach dem Anspringen sichtbar. Beispiele:

- „ˆX5“ setzt den Cursor auf die mit „ˆE5“ gesetzte Marke. Die Marke wird durch die Zeichen „`<5>`“ links neben dem Cursor dargestellt.
- „ˆX5 ˆE5“ setzt den Cursor auf die zuvor mit einem „ˆE5“ gesetzte Marke. Die Marke wird nicht angezeigt.

4.3.10 Das Hilfesystem

Rtos-Word stellt Ihnen ein kontext-sensitives Hilfesystem zur Verfügung. Wenn Sie die Hilfestufe 2 gewählt haben und das Fenster mindestens elf Textzeilen enthält, wird im oberen Teil des Fensters ein Hilfemenü eingeblendet, das wichtige Kommandos anzeigt. Wenn Sie ein Untermenü anwählen und nicht mehr genau wissen, wie das Kommando hieß, warten Sie einen kurzen Moment, dann blendet Rtos-Word die Untermenübefehle ein. Nun können Sie

das gewünschte Kommando heraussuchen und ausführen. Wenn Sie ein Kommando zügig eintippen, wird das Hilfemenü für das Untermenü aus Zeitersparnisgründen nicht eingeblendet.

71	Hilfemenü ein/aus	^XH

Geben Sie nach der Eingabeaufforderung eine „0“ ein, um das Hilfesystem zu beenden bzw. ausgeschaltet zu lassen. Bei Eingabe einer „2“ wird bzw. bleibt das Hilfesystem eingeschaltet.

4.3.11 Befehle zum Aufräumen

72	Bildschirm restaurieren	EscV

Der Bildschirm wird komplett neu aufgebaut. Dieses Kommando sollte benutzt werden, wenn nicht klar ist, ob der Bildschirm noch den aktuellen Ausschnitt des Textes zeigt. Diese Situtation kann z. B. auftreten, wenn eine andere Task Meldungen auf Ihren Bildschirm (im Window-Modus: in Ihr Fenster) ausgibt.

73	Datei komprimieren	EscH

Die Datei wird verdichtet, d. h. mit einer minimalen Anzahl von `/ED`-Blöcken abgelegt, um Speicher an RTOS-UH zurückzugeben. Der Cursor wird auf den Dateianfang gesetzt. Die Operation kann nach längerem Arbeiten wieder etwas Platz im Rechner schaffen.

74	Blockpuffer löschen	^ET

Der Blockpuffer wird gelöscht und der reservierte Platz RTOS-UH zurückgegeben.

75	Neue logische Zeilennummern	EscN

Die logischen Zeilennummern werden neu vergeben. Nach dieser Operation stimmen die physikalischen wieder mit den logischen Nummern überein.

4.3.12 Zusätzliche Befehle im Window-Modus

In diesem Unterabschnitt sind alle Fensterbefehle aufgeführt. Sie sind nur im Window-Mode erlaubt und führen im Terminalmode zu einer Fehlermeldung. Mit verschiedenen Befehlen dieses Abschnittes können Sie die Farben von RTOS-WORD ändern. Die Farben sind folgenden Zahlenwerten zugeordnet:

Zahl	Farbe	Zahl	Farbe	Zahl	Farbe	Zahl	Farbe
0	schwarz	1	rot	2	grün	3	braun
4	marine	5	lila	6	türkis	7	grau
8	anthrazit	9	hellrot	10	hellgrün	11	gelb
12	blau	13	pink	14	hellblau	15	weiss

Tabelle 4.7: Farbzuordnungstabelle von RTOS-WORD

76	**Farbe der Statuszeile ändern**	**^BI**

Mit diesem Befehl können Sie die Vorder- und Hintergrundfarbe der Statuszeile ändern. Geben Sie nach der Eingabeaufforderung die zu der gewünschten Vordergrundfarbe korrespondierende Zahl an und bestätigen Sie mit CR. Danach können Sie die Hintergrundfarbe auswählen. Sind beide Farben identisch, ignoriert RTOS-WORD die Eingabe. Haben Sie keine Zahl angegeben und nur CR gedrückt, entspricht dieses der „0“ bzw. schwarz. Die ausgewählten Farben wirken sich auf alle Textfenster aus, die Sie mit dieser Subtask bearbeiten. Aus Geschwindigkeitsgründen wird die Änderung in anderen Fenstern erst sichtbar, wenn Text neu aufgebaut werden muß oder Sie dieses mit „ESCV“ (Nr. 72), im jeweiligen Fenster ausgeführt, erzwingen.

77	**Farbe des markierten Blockes ändern**	**^BM**

Dieser Befehl ändert Vorder- und Hintergrundfarbe eines markierten Blockes. Die Eingabe ist mit „^BI“ (Nr. 76) identisch.

78	**Textfarbe ändern**	**^BT**

Mit diesem Befehl können Sie die Vorder- und Hintergrundfarbe des Textes ändern. Die Eingabe ist mit „^BI“ (Nr. 76) identisch.

79	**Farbe der Kommandozeile ändern**	**^BK**

Dieser Befehl ändert Vorder- und Hintergrundfarbe der Kommandozeile. Die Eingabe ist mit „^BI“ (Nr. 76) identisch.

80	**Textfensterbreite ändern**	**^BS**

Wollen Sie die dargestellte Textbreite ändern, können Sie neben der Maus diesen Befehl verwenden. Geben Sie nach der Eingabeaufforderung die Spaltenzahl des gesamten Fensters ein. Sind links die logischen Zeilennummern dargestellt, verringert sich die dargestellte Textbreite um acht Spalten. Die minimale Spaltenzahl ist z. Zt. auf 17 begrenzt. Ist Ihr Wert kleiner, wird die Spaltenzahl auf 17 gesetzt. Die Maximal-

breite ist nach dem ersten Öffnen eines Textes 96 Spalten. Kann Ihre Grafikkarte eine höhere Spaltenzahl darstellen, läßt sich die maximale Textfensterbreite durch Aufziehen des Textfensters erhöhen.

81	Textfensterhöhe ändern	^BZ

Mit diesem Befehl können Sie die dargestellte Texthöhe ändern. Geben Sie nach der Eingabeaufforderung die Zeilenzahl des dargestellten Textes an. Die minimale Zeilenzahl ist auf drei begrenzt. Ist Ihr Wert kleiner, wird die Zeilenzahl auf drei gesetzt. Die Maximalzeilenzahl ist nach dem ersten Öffnen eines Textes 27. Kann Ihre Grafikkarte eine höhere Zeilenzahl darstellen, können Sie die maximale Textfensterhöhe durch Aufziehen des Textfensters erhöhen.

82	Dateiauswahlfenster anzeigen	^BL

Dieser Befehl ist mit einem Klick in den Dateinamen identisch. Es wird ein Fenster erzeugt, das die Namen aller bearbeiteten Texte einschließlich der Änderungskennung „*“ anzeigt.
Mit diesem können Sie zu einem anderen Text wechseln. Die Selektion erfolgt durch einen Mausklick in den Textnamen, durch Anschlagen der dem Text vorangestellten Ziffer oder durch ein Cr , welches den markierten Text auswählt. Die Markierung wecheslt durch die Cursortasten ↑ und ↓. Nach der Dateiauswahl, durch einen Mausklick in das Schließfeld oder ein Esc verschwindet das Fenster.
War der selektierte Text iconisiert, wird das Fenster geöffnet.

83	Dateiübersichtsfenster anzeigen	^BW

Das Dateiübersichtsfenster hat fast dieselbe Wirkung wie das Dateiauswahlfenster (siehe „^BL“, Nr. 82). Allerdings bleibt das Fenster nach der Selektion erhalten, so daß es für weitere Textwechsel verwendet werden kann. Dieses ist sehr vorteilhaft, wenn man gleichzeitig mehrere Texte editiert und alle momentan nicht bearbeiteten iconisiert. Mit Hilfe dieser Box können Sie sofort den Richtigen auswählen. Um das Dateiübersichtsfenster aus der Fensterverwaltung zu entfernen, müssen Sie das Schließfeld anklicken.

84	Position des nächsten Textfensters	^BA

Wollen Sie vor dem Öffnen eines weiteren Textes (siehe „^EO“, Nr. 42) Fensterposition und -größe festlegen, können Sie diesen Befehl verwenden. Geben Sie zuerst die Fensterposition in der Reihenfolge Spalte/Zeile an und bestätigen Sie jeweils mit Cr. Die linke obere Ecke ist mit Spalte 0, Zeile 0 zu erreichen. Anschließend müssen Sie Fen-

sterbreite und -höhe eingeben und jeweils bestätigen. Dieser Befehl ist vor allem für die Fernsteuerung gedacht, um das nächste zu öffnende Fenster an der gewünschten Stelle mit der richtigen Größe zu öffnen.

4.3.13 Suchen und Ersetzen

RTOS-WORD erlaubt ein komfortables „Suchen“ sowie „Suchen und Ersetzen“. Die einzugebenden Strings und Suchoptionen können Sie mit den Befehlen gemäß Tabelle 4.5 editieren. Beim Suchstring sind die folgenden „Wildcards“ erlaubt:

Zeichen	Bedeutung
^A	Deckt sich mit jedem Zeichen
^T	Deckt sich mit jedem Zeichen, das weder Buchstabe noch Ziffer ist
^O*x*	Deckt sich mit jedem Zeichen außer „*x*“

Weiterhin gibt es die folgenden Suchoptionen:

Zeichen	Bedeutung
B	Es wird ab der Cursorposition rückwärts gesucht
G	Beginnt am Dateianfang, beim Ersetzen: „Alles“
N	Unterdrückt beim Ersetzen die Frage nach einem Austausch
U	Ignoriert im Suchstring Groß- und Kleinschreibung
W	Gleichheit wird nur bei ganzen Worten gefunden
xx	Suchen: Es wird das *xx*-te Auftreten gesucht Ersetzen: Es werden *xx* Ersetzungen durchgeführt

Sowohl bei der Eingabe als auch während der Suche können Sie ein Abbruch mittels „^U“ erzwingen.

85	**Text suchen**	**^XF**

Geben Sie nach der Eingabeaufforderung den zu suchenden Text (max. 30 Zeichen) ein. Ein Abschluß mit ESC beginnt die Suche sofort (nächstes Auftreten ab Cursorposition vorwärts), ein Abschluß mit CR ermöglicht die Eingabe von Optionen gemäß obiger Tabelle.

86	**Text suchen und ersetzen**	**^XA**

Geben Sie nach der Eingabeaufforderung den zu suchenden Text (max. 30 Zeichen) ein und bestätigen Sie diesen mit ESC oder CR. Nach der nächsten Eingabeaufforderung können Sie den Text, der den Suchtext ersetzen soll, eingeben (bis 30 Zeichen). Ein Abschluß mit ESC beginnt das Suchen und Ersetzen sofort (nächstes Auftreten ab Cursorposition

vorwärts mit Abfrage), ein Abschluß mit Cr ermöglicht die Eingabe der Optionen gemäß obiger Tabelle. Achten Sie darauf, daß die Angabe der Option „g“ alle Zahlenangaben übersteuert, also beliebig oft gesucht/ersetzt wird.

87	**Suchen und ggf. Ersetzen wiederholen**	^^

Das letzte Such- oder Austauschkommando wird wiederholt.

88	**Ersetzungstext einfügen**	**^XI**

Der Text, der bei „^XA“ (Nr. 86) den Suchtext ersetzen soll, wird an der aktuellen Cursorposition eingefügt.

4.3.14 Ausführen von Batchdateien

Sie können Kommandos für Rtos-Word auch aus einer sogenannten „Batchdatei“ einlesen lassen. Dies ist kann nützlich sein, wenn Sie z. B. die gleichen Änderungen in mehreren Dateien durchführen wollen. Sie können die Arbeit dann von einem Batchprozeß durchführen lassen und sich eine kreative Pause gönnen.

Soll Rtos-Word gleich mit der Abarbeitung einer Batchdatei beginnen, müssen sie einen weiteren Aufrufparameter verwenden (siehe Abschnitt 4.4). Rtos-Word liest diese Datei, in der Rtos-Word-Kommandos stehen müssen, als ob Sie alle Eingaben über die Tastatur machen würden. Die Ausgaben werden ganz normal exekutiert. Sie können also den Fortgang der Arbeiten am Bildschirm beobachten. Rtos-Word beendet den Automatikbetrieb beim Dateiende. Zum vorzeitigen Abbrechen des Automatikbetriebes müssen Sie „^U“ eingeben. Ein „^EX“ bzw. „EscX“ in der Batchdatei beendet den Automatikbetrieb nur, wenn lediglich ein Text bearbeitet wird! Bei mehreren Texten wird mit der nächsten Datei weiter gearbeitet.

Um eine solche Batchdatei zu erhalten, können Sie entweder den Protokoll-Mode („^EE“, Nr. 89) benutzen oder ein kleines Programm schreiben, daß so eine Datei erzeugt. Während des Editierens können Sie die Batchdateien mit „^EA“ (Nr. 90) abarbeiten.

89	**Eingabeprotokoll ein-/ausschalten**	**^EE**

Ist das Mitprotokollieren ausgeschaltet, können Sie nach der Eingabeaufforderung den Dateinamen der Protokolldatei angeben. In ihr werden alle Tastenanschläge mitprotokolliert, bis der Editor sich selbst terminiert (siehe „EscX“, Nr. 41) oder erneut „^EE“ angeschlagen

wird: War vor der Ausführung dieses Befehles der Protokollmodus eingeschaltet, wird er beendet.

90	Batchdatei exekutieren	^EA

Nach der Eingabe des Dateinamens liest RTOS-WORD seine Eingaben aus dieser Datei so, als ob Sie sie über die Tastatur eingegeben hätten. Sie können am Bildschirm beoabachten, was passiert. Hat RTOS-WORD das Dateiende erreicht, beendet der Editor den „Batchdateimodus“. Einen vorzeitigen Abbruch können Sie mit „^U“ erzwingen.

91	Makro zyklisch ausführen	^XQ

Geben Sie nach der Eingabeaufforderung ein Makro aus maximal 30 Zeichen ein, welches zyklisch abgearbeitet werden soll. Bestätigen Sie dieses mit CR und geben Sie anschließend eine Ziffer ein. Diese wird mit 0,5 sec multipliziert, um die Zykluszeit zu erhalten, wobei die „0“ eine Abarbeitung ohne Pause ermöglicht. Während der Abarbeitung können Sie die Zykluszeit durch Anschlagen einer Zifferntaste variieren. Ein Abbruch ist mit „^U“ möglich. Ein Makro wird allerdings immer zu Ende ausgeführt.

4.4 Übergabeparameter des Bedienbefehles

In Abschnitt 4.2.1 haben Sie die Möglichkeit kennengelernt, beim Aufruf des Editors gleich einen Dateinamen zu übergeben. RTOS-WORD verfügt noch über andere Parameter, die mit angegeben werden können. Die genaue Syntax lautet:

`WE`[[␣].*taskname*][␣`PRIO`␣*prio*][␣`AD`␣*cursorzeile*][[␣`SC`]␣*textname*]
[␣`LO`␣*batchdateiname*][␣`SI`␣*Device mit Terminal*]

Die einzelnen Übergabeparameter haben folgende Bedeutung:

- „. *taskname*“ legt den Namen der Editortask fest. RTOS-WORD benutzt im Window-Modus die ersten 6 Bytes von *taskname* für die Fensteradressierung. Achten Sie also darauf, daß weder zwei Editortasks mit gleichem Namen noch zwei Tasks existieren, bei denen die ersten 6 Buchstaben identisch sind.
- „*prio*“ gibt die Priorität des Editors vor. Sie darf zwischen 1 und 9999 liegen.
- „*textname*“ ist die Datei, die Sie bearbeiten wollen. Beachten Sie bitte auch die Anmerkungen in Abschnitt 4.2.1.

- „*batchdateiname*“ ist eine Batchdatei, die sofort nach dem Öffnen exekutiert wird.
- „*cursorzeile*“ gibt die Zeile an, auf der der Cursor nach dem Aufruf steht. Die Zeile 1 hat eine Sonderbedeutung: Auch beim Aufruf mit `WE` wird kein eigenes Window für den Editor eingerichtet.
- Weiterhin können Sie das Device „*Device mit Terminal*“ vorgeben, an dem RTOS-WORD den Nutzer erwartet. Bei dem Device muß das Bit „dialogfähiges Datenterminal gesetzt sein.“

4.5 Die Fernsteuerung

Soll RTOS-WORD innerhalb eines größeren Programmpaketes benutzt werden, kann das Programmpaket den Editor aufrufen und auch fernsteuern. Dadurch sind alle RTOS-WORD-Befehle auch von anderen Programmen nutzbar. Beispielsweise können beliebige Stellen im Text angelaufen, markiert und auch entfernt werden. Auch die Beendigung der Editortask ist möglich.

Eine Fernsteuerung ist über eine Ausgabe an die Datenstation `/VO` möglich. Die Pipe, in die der Text hineinzuschreiben ist, lautet „`/VO/`*`taskname`*“, wobei *taskname* der Name der Editortask ist. Beispiel: Haben Sie den Bedienbefehl „`WE.FERNSTEUER /H0/TEST`“ ausgeführt, bearbeiten Sie den Text „/H0/TEST“ und können diesen über die Pipe „`/VO/FERNSTEUER`“ ferngesteuert editieren.

Da RTOS-UH eine direkte Taskadressierung — in Assembler mittels des Traps `MSGSND`, in `PEARL` demnächst auch in Hochsprache — erlaubt, wurde diese Datenübertragung ebenfalls implementiert, zumal diese Ansteuerung eleganter und schneller als der Umweg über `/VO` ist.

Die Ausführung von Befehlen ist von Hochsprachen aus nicht so einfach, da die „Control-“ und „Escape-Sequenzen“ nur umständlich auszugeben sind. Daher gilt bei der Fernsteuerung folgende vereinfachte Regelung: Geben Sie das Zeichen „^“ und den Buchstaben, der normalerweise gleichzeitig mit der Control-Taste anzuschlagen ist, einfach hintereinander aus. RTOS-WORD faßt die beiden Zeichen zusammen. Sie können natürlich auch den richtigen Wert ausgeben (z. B. den ASCII-Wert „$05“ für „^E“). Die Escape-Taste können Sie über die Zeichen „^[“ nachbilden.

Damit während der Fernsteuerung kein Datensalat ensteht, wenn Nutzer und fernbedienendes Programm gleichzeitig RTOS-WORD füttern, arbeitet RTOS-WORD bei Eingaben aus `/VI` und der Tastatur prinzipiell immer einen Befehl zu Ende ab, bevor er nachsieht, ob sich auf einem anderen Kanal etwas getan hat. Bei der Direktadressierung ist ein Communication-Element eine Ausführungseinheit.

Es gibt hierbei eine Ausnahme: Hat der Nutzer den Eindruck, daß die Fernsteuerung wegen einer unsinnigen Sequenz hängt, kann er diese mit „^U“ abbrechen.

An Hand des folgenden Beipieles soll die Fernsteuerung verdeutlicht werden. Die Zeile 1000 soll markiert werden und in der Zeile 1000 die 30. Spalte angelaufen werden. Die Editortask habe wie oben den Namen „FERNSTEUER“.

Zuerst ein Shellskript:

```
O /VO/FERNSTEUER; ECHO \27'Z1000';: Identisch mit ^[Z1000
O /VO/FERNSTEUER; ECHO-N ^EB^J^EK;: Zeile 1000 markieren.
O /VO/FERNSTEUER; ECHO ^K^BC30;   : Zurueck auf Zeile 1000 und
                                  : Spalte 30 anlaufen.
EXIT(0);
```

Das gleiche leistet auch das folgende PEARL-Programm:

```
MODULE;
SYSTEM;
  PIPE: /VO/FERNSTEUER;
PROBLEM;
  SPC PIPE DATION   OUT ALPHIC;
  DCL ESC INV CHAR INIT(''\1B\'');
AA: TASK;
  PUT ESC,'Z1000' TO PIPE BY A,A,SKIP;
  PUT '^EB^J^EK'  TO PIPE BY A;
  PUT '^K^BC30'   TO PIPE BY A,SKIP;
END;
MODEND;
```

4.6 Alphabetisches Verzeichnis der Kommandos

Befehl	Nr.	Kurzerklärung
TAB	65	Cursor auf nächste Tabulatorspalte (siehe Tab. 4.3 und 4.4)
DEL	5	Zeichen unter dem Cursor löschen (siehe Tab. 4.3 und 4.4)
^A	13	Cursor ein Wort nach links
^C	37	nächste Bildschirmseite
^F	14	Cursor ein Wort nach rechts
^G	22	identisch mit Backspace
^H	9	Cursor ein Zeichen nach links
^I	65	identisch mit Tabulatortaste (siehe Tab. 4.3 und 4.4)
^J	11	Cursor eine Zeile tiefer
^K	12	Cursor eine Zeile nach oben
^L	10	Cursor ein Zeichen nach rechts
^M		identisch mit der Returntaste (CR) (siehe Tab. 4.3 und 4.4)
^N	21	Zeilenumbruch („harter“ Return)
^Q		Xon schicken (siehe Seite 268)
^R	38	vorherige Bildschirmseite
^S		Xoff schicken (siehe Seite 268)
^T	23	bis Wortende löschen
^U	27	Löschen rückgängig machen/Befehl abbrechen
^V	11	Cursor Zeile tiefer
^W	36	Text Zeile nach oben scrollen
^Y	24	Zeile löschen
^Z	35	Text Zeile nach unten scrollen
^_	1	Insertmodus ein/aus
^^	87	Suchen / Suchen und ersetzen wiederholen

Tabelle 4.8: RTOS-WORD-Kommandos mit einem Buchstaben

Die Befehle des ^B-Submenüs sind hinter denen des ^X-Submenüs aufgeführt.

Befehl	Nr.	Kurzerklärung
^E␣		Menü beenden
^E0	69	Marke 0 setzen
^E1	69	Marke 1 setzen
⋮	⋮	⋮
^E9	69	Marke 9 setzen
^EA	90	Batch Datei abarbeiten
^EB	50	Blockanfang markieren
^EC	52	Block kopieren
^ED	47	automatisches Sichern
^EE	89	Eingabeprotokoll ein/ausschalten
^EG	56	Block in Blockpuffer kopieren
^EH	60	Blockbefehle ein/aus
^EI	54	Block seitlich scrollen
^EJ	49	Datei löschen
^EK	51	Blockende markieren
^EL	48	Textnamen ändern
^EM	57	Blockpuffer einfügen
^EN	43	Text wechseln
^EO	42	weitere Datei öffnen
^EQ	45	Text ohne Abspeichern verlassen
^ER	59	Datei einlesen
^ES	46	Speichern des Textes
^ET	74	Blockpuffer löschen
^EU	40	Editor suspendieren
^EV	53	Block verschieben
^EW	58	Block in Datei schreiben
^EX	41	Text speichern und verlassen
^EY	55	Block löschen
^EZ	44	Text unter neuem Namen speichern und verlassen

Tabelle 4.9: RTOS-WORD-Kommandos im „E“-Submenü

Befehl	Nr.	Kurzerklärung
^O␣		Menü beenden
^OI	66	Tabulator setzen
^ON	67	Tabulator löschen
^OR	68	rechten Rand verändern
^OU	3	automatisches Einrücken ein/aus
^OW	4	Wortumbruch ein/aus
^OX	2	Klingel ein/aus

Tabelle 4.10: RTOS-WORD-Kommandos im „O“-Submenü

Befehl	Nr.	Kurzerklärung
^P␣		Menü beenden
^PA	8	SOH ($01) einfügen
^PB	8	STX ($02) – “ –
^PC	8	ETX ($03) – “ –
^PD	8	EOT ($04) – “ –
^PE	8	ENQ ($05) – “ –
^PF	8	ACK ($06) – “ –
^PG	8	BEL ($07) – “ –
^PH	8	BS ($08) – “ –
^PI	8	HT ($09) – “ –
^PJ	8	LF ($0A) – “ –
^PK	8	VT ($0B) – “ –
^PL	8	FF ($0C) – “ –
^PN	8	S0 ($0E) – “ –
^PO	8	S1 ($0F) – “ –
^PP	8	DLE ($10) – “ –
^PQ	8	DC1 ($11) – “ –
^PR	8	DC2 ($12) – “ –
^PS	8	DC3 ($13) – “ –
^PT	8	DC4 ($14) – “ –
^PU	8	NAK ($15) – “ –
^PV	8	SYN ($16) – “ –
^PW	8	ETB ($17) – “ –

Tabelle 4.11: RTOS-WORD-Kommandos im „P“-Submenü

Befehl	Nr.	Kurzerklärung
^X␣		Menü beenden
^X0	70	Cursor auf Marke 0 setzen
^X1	70	Cursor auf Marke 1 setzen
⋮	⋮	⋮
^X9	70	Cursor auf Marke 9 setzen
^XA	86	Text suchen und ersetzen
^XB	29	Cursor auf Blockanfang
^XC	20	Cursor zum Dateiende
^XD	16	Cursor zum Zeilenende
^XE	17	Cursor zum oberen Bildschirmrand
^XF	85	Text suchen
^XH	71	Hilfestufe wählen
^XI	88	Replace-Text einfügen
^XK	30	Cursor auf Blockende
^XQ	91	Kommando wiederholen
^XR	19	Cursor zum Textanfang
^XS	15	Cursor zum Zeilenanfang
^XU	28	letzte gelöschte Zeile einfügen
^XV	31	Cursor an Position vor „Suchen/Ersetzen“
^XX	18	Cursor zum unteren Bildschirmrand
^XY	25	ab Cursorposition bis Zeilenende löschen
^XZ	26	ab Cursorposition bis Zeilenanfang löschen

Tabelle 4.12: RTOS-WORD-Kommandos im „X“-Submenü

Befehl	Nr.	Kurzerklärung
^B␣		Menü beenden
^BA	84	Position des nächsten Textfensters
^BI	76	Farbe der Statuszeile ändern
^BK	79	Farbe der Kommandozeile ändern
^BL	82	Dateiauswahlfenster erzeugen
^BM	77	Farbe markierter Blöcke ändern
^BS	80	Textfensterbreite ändern
^BT	78	Textfarbe ändern
^BW	83	Dateiübersichtsfenster erzeugen
^BZ	81	Zeilenanzahl des Textfensters ändern

Tabelle 4.13: RTOS-WORD-Kommandos im „B“-Submenü

Befehl	Nr.	Kurzerklärung
Esc ␣		Menü beenden
Esc ↑	24	Zeile löschen
Esc ↓	6	Leerzeile einfügen
Esc →	7	Leerzeichen einfügen
Esc ←	5	Zeichen unter Cursor löschen
Esc A	12	Cursor aufwärts
Esc B	11	Cursor abwärts
Esc C	10	Cursor ein Zeichen rechts
Esc D	9	Cursor ein Zeichen links
Esc E	6	Leerzeile einfügen
Esc F	39	halbe Seite vorwärts blättern
Esc G	34	physikalische Spalte anlaufen
Esc H	73	Text komprimieren
Esc I	62	Pufferzeile einfügen
Esc K	64	Pufferzeile editieren
Esc L	6	Leerzeile einfügen
Esc M	24	Zeile löschen
Esc N	75	neue logische Zeilennummern
Esc O	61	Zeilenrest in Zeilenpuffer kopieren
Esc P	5	Zeichen unter Cursor löschen
Esc Q	7	Leerzeichen einfügen
Esc R	24	Zeile löschen
Esc S	63	ab Cursor nach Inhalt des Puffers suchen
Esc T	66	Tabulator an Cursorposition setzen
Esc U	67	Tabulator an Cursorposition löschen
Esc V	72	Bildschirm neu aufbauen
Esc X	41	Text speichern und verlassen
Esc Y	32	physikalische Zeilennummer anlaufen
Esc Z	33	logische Zeilennummer anlaufen

Tabelle 4.14: RTOS-WORD-Kommandos im „ESC“-Submenü

4.7 Standardmäßig unterstützte Terminals

RTOS-WORD unterstützt die Terminaltypen Televideo und VT52/100/220/320. Die Ansteuerung dieser Terminals soll in diesem Unterabschnitt beschrieben werden. Arbeiten Sie an einem Terminal, können Sie eine Anpassung mit Hilfe von Unterabschnitt 4.8 vornehmen. Über den RTOS-UH-Bedienbefehl „`SD`“ können Sie RTOS-WORD ihren Terminaltyp mitteilen:

- „`SD␣/TYA␣3300`“ für ein Televideo-Terminal
- „`SD␣/TYA␣3301`“ für ein VT52-Terminal
- „`SD␣/TYA␣3302`“ für ein Terminal der Typen VT100/220/320, das im 7 Bit Mode betrieben wird.

RTOS-WORD benötigt für seine Arbeit die Steuersequenzen gemäß der nachfolgenden Tabelle:

Funktion	Televideo	VT52	VT100/220/320
Bildschirm löschen	`1B2A`	`1B481B4A`	`1B5B324A`
Cursor positionieren	`1B3D`*rrcc*	`1B59`*rrcc*	`1B5B`*RR*`3B`*CC*`66`
inverse Darstellung	`1B29`	`1B70`	`1B5B376D`
normale Darstellung	`1B28`	`1B71`	`1B5B306D`
Zeilenende löschen	`1B54`	`1B4B`	`1B5B304B`

rr/*cc* steht für die Zeilen- und Spaltennummer in Binär-Darstellung mit einem Offset von $20. Die linke obere Ecke wird mit z. B. bei einem Televideo-Terminal mit „`1B3D2020`“ adressiert. *RR* und *CC* sind die Zeilennummern in ASCII-Darstellung mit einem Offset von 1. Bei einer zweistelligen Spaltenzahl ist also die Steuersequenz um 1 Byte größer als bei einer einstelligen. Bei einem VT100 Terminal beispielsweise wird die linke obere Ecke mit „`1B5B313B3166`“ erreicht, mit „`1B5B31313B333266`“ erreichen Sie die Position „11. Zeile/32. Spalte.“

Sollte Ihr Terminal mit den obigen Squenzen nicht zurecht kommen, müssen Sie eine eigene Anpassung gemäß Unterabschnitt 4.8 durchführen.

4.8 Das Konfigurationsmodul

Dieses Modul hat zwei völlig voneinander unabhängige Funktionen: Einerseits können Sie eine Terminalansteuerung konfigurieren, die RTOS-WORD nicht standardmäßig unterstützt. Andererseits können Sie für den Window-Modus die Startgröße und -position des ersten Fensters vorgeben sowie die Standardfarben für Text, Blöcke, Kommando- und Statuszeile vorgeben. In beiden Modi sind einige Defaultparameter veränderbar.

4.8.1 Die Anpassung an Ihr Terminal

RTOS-WORD benötigt die Funktionen „Bildschirm löschen“, „Cursor positionieren“, „normale Zeichendarstellung“, „inverse Zeichendarstellung“ und „ab Cursor bis Zeilenende löschen.“ Die Sequenz zur Cursorpositionierung darf maximal 12 Byte lang sein, alle anderen Sequenzen haben die Maximallänge acht. Die Sequenzen für Ihr Terminal finden Sie im dazugehörigen Handbuch.

Die Cursorpositionierung benötigt die anzusteuernde Zeile und Spalte. Daher ist in die Befehlssequenz für die Positionierung mit Hilfe eines Codes einzubauen, wie Ihr Terminal die Zeilen- und Spaltendarstellung erwartet: $80 steht als Platzhalter für die binäre Zeilenposition. Der auf die $80 folgende Wert ist der Offset zu Zeile 0 und wird nicht ausgegeben, sondern auf den Wert der aktuellen Zeile addiert. Ebenso steht die $81 als Platzhalter für die aktuelle binäre Spalte. Auch hier wird der folgende Wert addiert. Die Ansteuersequenz lautet also im Televideo-Modus `1B3D 8001 8101`. Die Platzhalter für die Zeilen- und Spaltenposition in ASCII-Darstellung lauten $83 und $82. Der darauf folgende Wert ist wieder der Offset. Bei einem VT320-Terminal lautet die Befehlssequenz zur Cursoransteuerung „`1B5B 8301 3B82 0166`“.

Als weitere Parameter im Terminalmode stehen im Konfigurationsmodul die Bildschirmzeilenzahl. Außerdem können Sie vorgeben, ob RTOS-WORD im Einsetz- oder im Überschreibmodus und im ein- bzw. ausgeschalteten Einrückmodus anläuft. Sie können außerdem die Anzeige der logischen Zeilennummer unterdrücken. Im Beispielmodul in Unterabschnitt 4.8.2 finden Sie die Erläuterungen zur Konfigurierung.

4.8.2 Beispielmodul

Das folgende Listing stellt ein Beispiel für ein Konfigurationsmodul dar.

```
*-------------------------------------------------------*
*        Muster-Konfigurationsmodul fuer WORD           *
*        Universalmodul mit allen moeglichen Parametern *
*-------------------------------------------------------*
       DC     0,0,0,0,$0010  Modul-Kopf                 *
       DC.L   NAME-$         Zeiger auf den Namen       *
       DC     0              Name ist relativ angegeben *
NAME   DC.B   'WORD_para',$FF Name des Moduls (alle User)*
*NAME   DC.B   'WORD_par1',$FF Name des Moduls (USER1)   *

*             Parameter         Kommentar               *
*-------------------------------------------------------*
*                  Sektion 'TERMINAL'                   *
```

```
*-------------------------------------------------------*
CLEAR  DC     $1B2A,0,0,0       Bildschirm loeschen      *
POS    DC     $1B3D,$8020,$8120 Positionieren            *
       DC     0,0,0                                      *
INVERS DC     $1B29,$0000,0,0   Schrift invers           *
NORMAL DC     $1B28,$0000,0,0   Schrift normal           *
DELEND DC     $1B54,0,0,0       Bis Zeilenende loeschen  *

*      Laenge des Bildschirms-4 (min. 7, max. 21)       *
SCRELI DC     15                                         *

*      Insert- oder Replace-Modus                       *
*      Insert-Modus:  $0          Replace-Modus: $FFFF  *
INSMOD DC     $FFFF              Start im Replace-Modus  *

*      linker Rand mit Zeilennummer: 0 ohne: 8;<>0      *
LEFTMA DC     8                  keine Zeilennummer      *
*-------------------------------------------------------*
*                 Sektion 'MODE'                        *
*-------------------------------------------------------*
*      Diese Sektion gilt fuer WIM und TER-Modus
*      $0     =   Standardwerte laden                   *
*      $FFFF  =   umgekehrter Modus (<>0)               *
       DC.B   'MODE'         Sektions-Kennung           *
_INSMO DC     $0             einfuegen     (Standard)   *
INDENT DC     $0             einruecken    (Standard)   *
WRAP   DC     $0             kein Umbruch  (Standard)   *
_LEFTM DC     $0             linker Rand MIT Zeilennummer *
       DS     12             reserviert
*-------------------------------------------------------*
*                 Sektion 'WINDOW'                      *
*-------------------------------------------------------*
*      Diese Sektion gilt nur im WIM-Modus              *
*      Farben                                           *
*      0,0     =   Standardwerte laden                  *
*---------------------- Farben -------------------------*
rot        EQU   1
gruen      EQU   2
braun      EQU   3
marine     EQU   4
lila       EQU   5
tuerkis    EQU   6
```

```
grau       EQU   7
anthrazit  EQU   8
hellrot    EQU   9
hellgruen  EQU  10
gelb       EQU  11
blau       EQU  12
pink       EQU  13
hellblau   EQU  14
weiss      EQU  15
schwarz    EQU   0
       DC.B   'WINDOW'        Sektions-Kennung             *
BLOCK  DC.B   anthrazit,weiss Hinter-/Vordergrund Bloecke  *
TEXT   DC.B   grau,schwarz    Hinter-/Vordergrund Text     *
STATUS DC.B   grau,schwarz      "       "    Statuszeile   *
COMMND DC.B   grau,schwarz      "       "    Kommandozeile *
XPOS   DC      5              Startposition:  5. Spalte     *
YPOS   DC      1              Startposition:  1. Zeile      *
XSIZE  DC      80             Startgroesse:  80 Spalten     *
YSIZE  DC      26             Startgroesse:  26 Textzeilen  *
*-------------------------------------------------------------*
       DS     12              reserviert
       END
```

Beachten Sie die folgenden Erläuterungen:

- Ein Konfigurationsmodul `WORD_par1` wirkt nur auf User 1. Entsprechendes gilt für die anderen User. Soll es für mehrere User gelten, muß es `WORD_para` lauten. Sind sowohl ein userspezifisches als auch ein allgemeingültiges Modul geladen, hat das userspezifische Vorrang.
- Die Terminalsektion wird nur ausgewertet, wenn die Device-Facilities der seriellen Schnittstelle auf $3303 stehen.
- Alle anderen Sektionen werden ausgewertet, wenn das Modul zu dem User gehört (s. o.).
- Die den einzelnen Labeln zugeordneten Bytelängen müssen auf jeden Fall eingehalten werden.
- Mindestens eine Sektion muß im Modul enthalten sein.
- Innerhalb einer Sektion darf die Parameterreihenfolge nicht geändert werden.
- Folgt hinter einem Komma ein weiterer Parameter, darf kein Leerzeichen zwischen Komma und Parameter stehen.

- Terminalsektion: Ist Ihre Ansteuersequenz kürzer als die vorgegebene Länge, müssen Sie Nullbytes/Nullwörter auffüllen. RTOS-WORD ignoriert diese bei der Ansteuerung des Terminals. `SCRELI` gibt die Anzahl der darzustellenden Textzeilen an. Ihr Terminal muß mindestens 4 weitere Zeilen darstellen können. Statt `INSMOD` und `LEFTMA` sollten Sie Sektion „MODE“ benutzen.
- Windowsektion: Sind Vorder- und Hintergrundfarbe gleich oder hat eine der beiden Farben einen ungültigen Wert, werden die Standardwerte verwendet. Bei der Positionierung hat die linke obere Ecke des WiM den Wert 1/1. Hat das zu öffnende Fenster keine Zeile oder keine Spalte, wird die Defaultgröße verwendet. Sind Maße zu groß, werden die Maximalwerte verwendet.

4.9 Besonderheiten bei der Einbindung in das Betriebssystem RTOS-UH

1. Arbeiten Sie an einem Terminal, wird die Schnittstelle im Xon/Xoff-Protokoll betrieben. Trotzdem kann es zu Verwirrung auf dem Bildschirm kommen. In einem solchen Fall können Sie mit „ESCV“ den Bildschirm neu aufbauen lassen.
2. Es gibt VT52-Terminals, die Schriften nicht invers darstellen können. Die in Unterabschnitt 4.7 angegebenen Sequenzen sind zwar korrekt und auch korrekt programmiert, aber es gibt unterschiedliche VT52-Terminals. Auf jeden Fall funktionieren alle Befehle korrekt. RTOS-WORD kann jedoch markierte Blöcke nicht anzeigen.
3. Haben Sie aus Versehen „^S“ gedrückt, kann es vorkommen, daß ihr Terminal dieses sofort abfängt und die Schnittstelle blockiert. Sollte es einmal vorkommen, daß sich auf ihrem Terminal nichts mehr ändert, drücken Sie erst einmal „^Q“ zur Freigabe der Schnittstelle, bevor Sie glauben, daß RTOS-UH abgestürzt ist.
4. RTOS-WORD verhindert in den meisten Fällen, daß Sie mit „^A“, „^B“ und „^C“ das Kommandointerface erreichen. Sehen Sie trotzdem einmal den Eingabeprompt, kann dies zwei Ursachen haben: Das Drücken der BREAK-Taste unterdrückt RTOS-UH absichtlich nicht, um immer noch in das Betriebssysstem kommen zu können. Arbeiten noch andere Tasks auf ihrer Schnittstelle – besonders gemein sind die, die ab und zu auf die Tastatur pollen –, kann es dazu kommen, daß der Unterdrückungsmechanismus ausgeschaltet wird. Wollen Sie genau danach eine Seite nach unten blättern („^C“ !!), haben Sie den Datensalat. Zum Beheben brauchen Sie lediglich die CR-Taste, gefolgt von einem „ESCV“ zu drücken.

5. Erscheinen statt der eckigen Klammern „[“ und „]“ deutsche Buchstaben, dann steht Ihr Terminal, Ihre Terminalemulation oder Ihr Fenster auf deutschem Zeichensatz. Falls Sie diese Zeichenausgabe nicht stört, brauchen Sie nichts weiter zu tun; RTOS-WORD arbeitet weiterhin einwandfrei. Wollen Sie die eckigen Klammern statt der Umlaute betrachten, müssen Sie auf einen englischen Zeichensatz umstellen.

4.10 Statusmeldungen und Eingabeaufforderungen

Verschiedene Aktionen von RTOS-WORD, wie z. B. das Suchen von Text, dauern etwas länger. Damit Sie sich nicht wundern, warum RTOS-WORD nicht mehr auf Ihre Eingaben reagiert, geben Kommandos, die länger dauern können, eine Statusmeldung aus. Diese sind im folgenden dargestellt.

`Please wait: I'm searching.` Es wird nach einem String gesucht. Die Meldung verschwindet, wenn der String erreicht oder die Datei abgearbeitet ist.

`Please wait: I'm packing.` Diese Meldung erscheint beim Verlassen incl. Abspeichern des bearbeiteten Textes. Nach dem Verdichten der `/ED`-Datei bzw. Zurückschreiben der Arbeitskopie auf das Original hat RTOS-WORD diesen Text aus seiner Verwaltung entfernt.

`Please wait: I'm saving.` Die Datei wird gesichert. Nach dem Sichern können Sie mit der Bearbeitung fortfahren.

`Please wait: I'm loading.` RTOS-WORD legt die lokale Arbeitskopie an. Mit dem Aufblättern des Textes verschwindet diese Meldung.

`taskname` `suspended waiting` Im Terminal-Mode haben Sie den Editor mit „^EU“ unterbrochen. Fahren Sie mit der Arbeit fort, indem Sie von der Shell aus „`C` *`taskname`*“ eingeben.

`O. K., never mind!` Sie wollten einen noch nicht vorhandenen Text editieren. Bevor Sie jedoch damit begonnen haben, haben Sie sich anders entschieden.

Die folgende Auflistung enthält alle Eingabeaufforderungen. Sie müssen alle Eingabeaufforderungen, die nicht mit „`(y/n)`“ enden, mit CR quittieren, es sei denn, bei der Erklärung steht etwas anderes. Die Eingabeaufforderungen, die direkt nach einer Befehlseingabe erscheinen, enthalten einen Verweis auf den Befehl, der die Meldung ausgibt.

`Set Tab at position:` Sie wollen einen neuen Tabulator einfügen („^OI“, Nr. 66).

`Del Tab at position:` Sie wollen einen Tabulator löschen („^ON“, Nr. 67).

`Set right margin at:` Sie möchten den rechten Rand neu setzen („^OR“, Nr. 68).

`Enter steps (-)= left:` Sie können einen Block ein-/ausrücken („^EI“, Nr. 54).

`Destroy old file? (y/n)` Sie wollen einen Block herausschreiben („^EW“, Nr. 58) und die von Ihnen angegebene Datei ist vorhanden. Wenn Sie mit „y“ antworten, wird sie überschrieben, sonst wird die Aktion abgebrochen. Diese Meldung erscheint auch, wenn Sie einen Block an einen Drucker schicken.

`Enter search string:` Sie wollen einen Text suchen oder suchen und ersetzen („^XF“, Nr. 85 bzw. „^XA“, Nr. 86). Geben Sie bitte den Suchtext ein. Eine Bestätigung ist auch mit Esc möglich.

`Enter replace string:` Erscheint nach beim Suchen und Ersetzen („^XA“, Nr. 86). Geben Sie bitte den Text ein, der den gesuchten ersetzen soll. Eine Bestätigung ist auch mit Esc möglich.

`Enter file name:` Sie wollen etwas aus einer Datei lesen oder in Datei schreiben. Geben Sie bitte den Namen dieser Datei an.

`Options ?` Sie können nun die Such- bzw. Such-und-Ersetz-Optionen eingeben. Die Ausführung kann auch mit Esc forciert werden.

`Exchange ? (y/n)` Rtos-Word hat beim „Suchen und Ersetzen“ den Suchtext gefunden und fragt Sie nun, ob er ihn ersetzen soll. „y“ ersetzt den Text.

`Enter repeat command:` Sie wollen ein Makro zyklisch ausführen („^XQ“, Nr. 91). Sie können eine Folge von Kommandos eingeben, die dann immer wieder abgearbeitet wird.

`Repeat rate (0-9):` Bei der zyklischen Ausführung eines Makros gibt die von Ihnen angeschlagene Ziffer die Zykluszeit in 0,5 sec an. Die Zahl darf nicht mit Cr bestätigt werden.

`Help level (0 oder 2):` Die Hilfestufe („^XH“, Nr. 71) wird festgelegt. „0“ schaltet aus, „2“ schaltet ein. Die Zahl darf nicht mit Cr bestätigt werden.

`This file will not be saved! (y/n)` Sicherheitsabfrage, wenn Sie eine geänderte Datei ohne Abspeichern verlassen wollen („^EQ“, Nr. 45) . Antworten Sie mit „y“, so wird die Datei tatsächlich nicht gespeichert.

`No chance! You still want exit? (y/n)` Sie haben RTOS-WORD beim Aufruf eine `/ED`-Datei übergeben oder mit „^EO“ (Nr. 42) geöffnet. Diese Datei können Sie weder umbenennen („^EL“, Nr. 48) noch ohne Veränderung verlassen („^EQ“, Nr. 45). Auch ein Verlassen mit neuem Namen („^EZ“, Nr. 44) ist nicht möglich.

`Zoneselect-Enter Line:` Nach einem „EscZ“ (Nr. 33) können Sie die logische Zeilennummer eingeben, die Sie anlaufen wollen.

`Enter save Time (Min):` Erscheint nach dem Befehl „Automatisches Sichern“ („^ED“, Nr. 47). Geben Sie den Abstand in Minuten an, nach dem jeweils automatisch gesichert werden soll.

`Set cursor to column:` Nach dem Drücken von „EscG“ (Nr. 34) können Sie nun die Spalte angeben, auf die der Cursor positioniert werden soll.

`Next open pos&size; X:` Diese Meldung leitet die Koordinateneingabe und Fenstergröße für das nächste zu öffnende Fenster („^BA“, Nr. 84) ein. Hier ist die Spaltenposition des nächsten Fensters gefragt.

`Y:` Geben Sie bitte die Zeilennnummer ein, in der das nächste Fenster geöffnet werden soll.

`width:` Diese Meldung erwartet die Anzahl von Spalten, die das nächste zu öffnende Fenster haben soll.

`height:` Hier können Sie die Zeilenzahl angeben, die das nächste zu öffnende Fenster haben soll.

`status line colors: foreground:` Hier ist nach der Vordergrundfarbe der Statuszeile („^BI“, Nr. 76) gefragt. Die Zuordnung zwischen Farbe und Zahl finden Sie in Tabelle 4.7.

`Command line colors: foreground:` Geben Sie bitte die Vordergrundfarbe der Kommandozeile („^BK“, Nr. 79) entsprechend Tabelle 4.7 ein.

`Normal text colors: foreground:` Sie wollen Textfarbe ändern („^BT“, Nr. 78). Geben Sie bitte die Buchstabenfarbe entsprechend Tabelle 4.7 ein.

`Selected text colors: foreground:` Diese Meldung erwartet die Vordergrundfarbe markierter Blöcke („^BM“, Nr. 77) entsprechend Tabelle 4.7.

`background:` Sie wollen die Farbe der Kommando- oder der Statuszeile, des normalen oder des markierten Textes ändern. Geben Sie die Hintergrundfarbe über eine Zahl gemäß Tabelle 4.7 ein.

`Enter window columns:` Diese Meldung erscheint bei einer Änderung der Fensterspaltenzahl („^BS“, Nr. 80). Geben Sie die neue Spaltenzahl ein.

`Enter window lines:` Sie wollen die Fensterhöhe verändern („^BZ“, Nr. 81). Geben Sie bitte die neue Höhe in der Anzahl von Textzeilen ein, die Sie gleichzeitig sehen wollen.

4.11 Fehlermeldungen

Neben den hier aufgeführten Fehlermeldungen verwendet RTOS-WORD für einige Bedienbefehle, die auf Dateien arbeiten, den Report-Error-Mechanismus von RTOS-UH. Bei einem Fehler produziert RTOS-WORD nicht eine Standardfehlermeldung, sondern fragt die Betreuungstask nach deren Fehlertext und gibt diesen aus. Diese Meldungen können Sie daran erkennen, daß sie mit einem `#` und dem Namen der Betreuungstask beginnen.

Alle Fehlermeldungen, die mit `(CR)` enden, müssen von Ihnen mit einem CR quittiert werden.

`No Workspace. Try later` Ihr Rechner hat nicht genügend freien Speicher. Um mit RTOS-WORD arbeiten zu können, müssen Sie freien Speicher durch Entladen oder Terminieren von Tasks oder Entfernen von Files der Datenstation `/ED` schaffen.

`***COMMAND-ERROR (CR):` Sie haben eine unzulässige Taste oder Tastenkombination gedrückt.

`Command not avail.(CR)` Sie haben im Terminal-Modus einen Befehl aufgerufen, der nur im Window-Modus erlaubt ist.

`***NO MEM:SUSPENDED` RTOS-WORD benötigt noch eine `ED`-Speichersektion, und RTOS-UH hat nicht mehr genügend freien Speicherplatz. Die Editortask hat sich selbst suspendiert. Zum Fortsetzen sollten Sie mindestens 4 KByte Speicher freigeben. Anschließend können Sie die Task mit dem Bedienbefehl „`C` *`taskname`*“ fortsetzen.

`Sorry. Can't find. (CR)` Diese Meldung erscheint, wenn eine Textsuche fehlschlug.

`BAD POINTER/BREAKDOWN.` Interne Zeigerstrukturen sind zusammengebrochen. Die Editortask hat sich terminiert. Die `/ED`-Datei sollte allerdings noch vorhanden sein. Sie können sie noch umkopieren, die Änderungen der letzten auf dem Terminal sichtbaren Textseite sind aber verloren. Dieser Fehler sollte eigentlich nicht vorkommen, laßt sich jedoch immer provozieren, wenn der Nutzer über Befehle an die Datenstation `/ED` das Arbeitsfile manipuliert.

`Not implem. yet (CR)` Sie haben eine Tastenkombination gewählt, die für eine spätere Verwendung vorgesehen ist.

`Block def. wrong (CR)` Eine Ausführung des von Ihnen gewünschten Blockbefehles ist nicht möglich, da entweder die Blockende- vor der Blockanfangmarke steht oder eine der beiden Marken nicht gesetzt ist.

`Not enough memory (CR)` Bei einer Blockoperation kann RTOS-WORD mangels freiem Speicher keinen Plockpuffer einrichten.

`Unable to open! (CR)` RTOS-WORD sieht sich nicht in der Lage, die gewünschte Datei zu öffnen. Meist ist ein simpler Tippfehler schuld.

`Unknown device! (CR)` Das von Ihnen angegebene Device existiert auf diesem Rechner nicht.

`Read error! (CR)` Die Datei konnte zwar geöffnet werden, aber der Filemanager, der das Device betreut, konnte die Datei nicht bis zum Dateiende lesen.

`Write error! (CR)` Der Schreibvorgang wurde zwar korrekt begonnen, mußte jedoch vorzeitig beendet werden (z. B. Diskette voll).

`Line too long! (CR)` Mit Ihrer Operation würde die Zeile zu lang, deshalb wird sie nicht ausgeführt. Direkt nach dem Laden besagt diese Meldung, daß die Datei zu lange Zeilen hatte, die RTOS-WORD zwangsumgebrochen hat.

`**Aborted command (CR)` Sie haben einen Kommando mit „^U“ abgebrochen. RTOS-WORD bestätigt den Abbruch und bittet Sie diese Bestätigung zu quittieren.

`Unable to open file. Bye, Bye!` RTOS-WORD kann mit diese Datei nicht öffnen. Kann z. B. auftreten, wenn ein Ordner des Pfades nicht existiert oder der Dateiname der Betreuungstask des Devices zu lang ist. Diese Meldung erscheint nur im Terminalmode.

`Sorry, can't work on this device. Bye, Bye!` Die von RTOS-WORD zu lesende Textdatei steht auf einem nicht rückspulbaren Device. Da das Lesen solcher Dateien problematisch für RTOS-WORD werden kann, wird dieser Versuch abgebrochen.

`WRONG LDN (MODE)` Sie wollen RTOS-WORD von einem Device bedienen, welches nicht die Datenstationseigenschaft „dialogfähiges Datenterminal“ besitzt. RTOS-WORD bricht ab und terminiert sich.

`Can't read input file. Bye, Bye!` Die Eingabedatei ist zwar vorhanden, kann aber nicht korrekt gelesen werden. RTOS-WORD bricht ab und terminiert sich.

`Can't write output file. Bye, Bye!` Ihr Rechner hat zuwenig freien Speicher. Die temporäre `/ED`-Datei kann nicht angelegt werden.

`Write-error on` *`filename`*`! Please take /ED/`*`name`* Beim Verlassen der Datei kann RTOS-WORD die lokale Kopie nicht zurückschreiben. Sie können `/ED/`*`name`* – *name* bezeichnet die lokale Arbeitskopie – mit Hilfe des Bedienbefehles „`CP`“ kopieren.

`Read-error on` *`filename`*`! Please take /ED/`*`name`* Beim Verlassen der Datei kann RTOS-WORD die lokale Kopie nicht zurückschreiben. Sie können `/ED/`*`name`* mit Hilfe des Bedienbefehles „`CP`“ kopieren.

`Can't delete file (CR)` Die angegebene Datei konnte nicht gelöscht werden.

4.12 Technische Daten

Code:	ca. 44 Kbyte; Hilfesystem: ca. 3 KByte Code ist wiedereintrittsfest. Dadurch können mehrere Editoren gleichzeitig laufen.
Datenbereich:	4 KByte + 12 KByte für jedes File Platz für Blockpuffer je nach Größe des Blockes Bei Nicht-`/ED`-Dateien Platz für lokale Textkopie
Betriebsmodi:	Window-Modus: Eigenes größeneinstellbares Fenster für jeden Text (Window-Manager notwendig) Terminal-Modus: Konsolenfenster als Arbeitsfenster (Bei Terminals und Terminalemulationen)
Unterstützte Terminals:	Televideo; VT52/100/220/330 Andere Typen über Konfigurationsmodul
Fenstergröße:	Terminal-Modus: Standard 80 Spalten / 24 Zeilen Andere Größen über Konfigurationsmodul Window-Modus: Maximalgröße durch Grafikauflösung bestimmt
Anzahl editierbarer Texte:	Terminal-Modus: nur durch Speicherplatz begrenzt Window-Modus: 100 Texte pro Editortask
Textmaximalgröße:	nur durch Speicherplatz begrenzt die ersten 65500 Zeilen sind editierbar, Textspalten auf 231 begrenzt
Erlaubte Zeichen:	ASCII-Werte von \$20...\$FF, fast alle Sonderzeichen
Farben:	16 Farben für Text und Block sowie Kommando- und Statuszeile
Ansteuerung:	Über Terminal oder Pipes Ausführung von Batchdateien möglich

(Leere Seite vor neuem Kapitel)

Kapitel 5: Programmieren in PEARL

5.1 Die PEARL-Compiler-Familie

5.1.1 Compilertypen und Zielprozessoren

Die Kompilation von PEARL90-Programmen kann auf jedem **RTOS–UH**-Rechner, aber auch auf Fremdsystemen erfolgen. Für die handelsüblichen PCs gibt es das sog. C-VCP-Paket, das den Original RTOS/PEARL-Compiler (es ist wirklich Bit für Bit der gleiche binäre Code!) auf solchen Rechnern als Cross-Compiler nutzbar macht. Für einige Unix-Systeme gibt es ein entsprechend angepasstes C-VCP-Paket, für das die gleiche absolute Kompatibilität gilt. Mit dem Paket sind auch Linker und Assembler auf Fremdsystemen einsetzbar.

Insgesamt existieren zur Zeit außer den Demo-Versionen anscheinend 6 verschiedene PEARL90-Compiler. Alle Varianten decken exakt den gleichen PEARL-Sprachumfang ab und haben die gleiche Revisionsnummer, weil sie in Wirklichkeit aus dem gleichen Quellfile übersetzt wurden. Die Varianten sind:

„MINI“: Diese Variante kann nur Code für den Prozessortyp MC68000 generieren. Die so erzeugten (Code-) S-Rekords sind auf allen **68K-RTOS–UH**-Zielsystemen ablauffähig. Es wird aber nicht immer der jeweilige Prozessortyp optimal ausgenutzt. Als Gleitkommaformat wird das **RTOS–UH** eigene Format verwendet.

„MAXI“: Dies ist der eigentliche professionelle Standardcompiler. Mit ihm kann wahlweise Code für die Zielprozessoren MC68000, MC68020, MC68040 und MC68020+MC68881 erzeugt werden.

Als Defaulteinstellung dient der Prozessortyp des Rechners, auf dem der Compiler läuft, der Entwicklungsrechner. Diese Einstellung erfolgt automatisch beim Aufruf des Übersetzers. Der 68040 allerdings wird bei dieser Defaulteinstellung behandelt, als sei er ein Gespann, bestehend aus 68020+68882. Von diesem Spezialfall abgesehen, wird immer optimal auf das Entwicklungssystem abgestimmter Code erzeugt. Wenn man sicher ist, daß man die 68040-Welt mit den S-Rekords nie verlassen wird oder man harte Echtzeitbedingungen einhalten muß, so sollte man mit der unten

beschriebenen Methode den Prozessortyp explizit auf „`P=68040`“ einstellen. Gleiches gilt für den Typ `68060`.

„PowerPC“: Dieser Compiler kodiert für den Prozessor PowerPC. Auch er ist vom Sprachumfang her völlig identisch zu den anderen Varianten. Er läßt sich sowohl auf Hardware-Float als auch auf Software-Float einstellen.

„CROSS68“: In Wirklichkeit ist es der Maxi-Compiler, Tatsächlich ist er im Binärcode Bit für Bit mit ihm identisch. Mit Hilfe des C-VCP läuft er jedoch auf allen möglichen Fremdsystemen. Er defaultiert als Prozessortyp, den er aus dem Gastsystem natürlich nicht ermitteln kann, stets MC68000.

„CROSSPPC“: In Wirklichkeit ist es der PowerPC-Compiler, für ihn gilt das über den „CROSS68“ gesagte sinngemäß. Er defaultiert auf Software-Float.

„QUICK“: Auch dies ist der 68K-Maxi-Compiler. Er setzt den Defaulttyp an Hand des Entwicklungsrechners wie dieser. Weil der Compiler mit einem speziellen Übersetzer behandelt wurde, läuft er jedoch mit sehr viel höherer Arbeitsleistung, mindestens doppelt, meist sogar dreimal so schnell wie der Standard-Maxi. Aufgerufen wird er mit „`QP`“. Sein einziger Nachteil ist, daß er für seine Arbeit deutlich mehr Speicher verbraucht, nämlich ca. 200 kByte statt ca. 50 kByte zur Ablage seines Codes. Wenn man Programme auf der Winston-68k-RTOS-Emulation entwickeln will und häufig kompilieren muß, so kann man diesen Compiler einmal laden und dann zeitsparend als residente Shellerweiterung benutzen.

Eine Quick-Version für den PowerPC existiert noch nicht.

Meist ist das Zielsystem nicht identisch mit dem Entwicklungssystem, insbesondere wenn einer der beiden CROSS-Compiler z. B. unter MS-DOS/Windows oder Unix benutzt wird. Dann kann – außer beim „Mini“ – die Zielprozessor-Defaulteinstellung durch eine Prozessorzuweisung in der ersten PEARL--Programmzeile übersteuert werden. Diese Übersteuerung kann mit Hilfe der eingebauten benamten Konstanten `P_68K` oder `P_PPC` und dem Preprozessorbefehl `#IFDEF P_68K` (bzw. `#IFDEF P_PPC`) vom verwendeten Zielprozessorsystem abhängig gemacht werden. Dazu stehen mit `P= ...` folgende Steueranweisungen zur Verfügung:

`P=68000;` „MINI“–kompatibler Mode, der erzeugte Code ist voll kompatibel zum „MINI“-Übersetzer.

Floatdarstellung: **RTOS**–**UH**-Format

Befehlsumfang: MC68000 + virtuelle Codes

`P=68020;` nutzt Befehle des MC68020, daher ist der Code nicht kompatibel zum „MINI“ und kann nur in Zielsystemen mit MC68020 und darüber exekutiert werden.

Floatdarstellung: **RTOS**–**UH**-Format

Befehlsumfang: MC68020 + virtuelle Codes

`P=68040;` nutzt Befehle, die speziell nur im 68040-System implementiert sind. Die Angabe führt dazu, daß die mathematischen Funktionen, die der 68040 nicht „on chip“ hat (sin, tan etc.), durch echtzeitkonforme, besonders schnelle und jederzeit unterbrechbare Unterprogramme realisiert werden. Dies ist eine wichtige

! → Spezialität unseres 68040-RTOS/PEARL. Man beachte, daß ohne diese Angabe auch in unserem System die in anderen Systemen übliche, zwar von Motorola vorgeschlagene, aber schlechte Lösung mit F-line Emulation benutzt wird und so das Echtzeitverhalten ganz erheblich (Rechnung im Supervisorprozeß!!) verschlechtert wird. Es ist noch in der Diskussion, ob später nicht auch die automatische Anpassung diese 68040-Option generieren soll.

Floatdarstellung: IEEE-Format

Befehlsumfang: MC68020 + MC68881 + virtuelle Codes + Sonderfunktionen.

`P=68040(`*n*`);` Wie oben, jedoch werden nun *n* (3...8) Floatingpoint-Register als

Kontext gerettet. Man braucht diese Option, wenn eigene Assemblerprogramme höher numerierte Register FPx benutzen. Die Kontextswitchzeit wird im Gegensatz zum 68020+68881 beim 68040 normalerweise nicht relevant verschlechtert, wenn man zur Sicherheit immer `P=68040(8);` einsetzt.

`P=68060;` vermeidet einige Befehle, die auf dem Prozessor `68060` nicht vorhanden sind und emuliert werden müssen, ansonsten wie `P=68040`. Die Modes `68040` und `68060` erzeugen Programme, die auf beiden Prozessoren auch über Kreuz lauffähig sind. Lediglich kleinere Optimalitätseinbußen können eintreten.

Floatdarstellung: IEEE-Format

Befehlsumfang: MC68020 + MC68881 + virtuelle Codes + Sonderfunktionen.

`P=68060(`*n*`);` Wie oben, jedoch werden nun n (3...8) Floatingpoint-Register als Kontext gerettet.

`P=68881;` nutzt Befehle des MC68020 und des MC68881/2 (FPU), daher ist der übersetzte Code nur auf Systemen mit dieser Hardware einsetzbar. Auf Prozessoren des Typs 68040 und 68060 sind die Programme lauffähig, jedoch nur über die im Echtzeitbereich ungünstige und teilweise erheblich langsamere Trapemulation des Motorola-Ansatzes.

Floatdarstellung: IEEE-Format

Befehlsumfang: MC68020 + MC68881 + virtuelle Codes
Zusätzliche MC68881 typische Einbaufunktionen

`P=68881(`*n*`);` ermöglicht eine Angabe der beim Taskwechsel zu rettenden FPU–Register. Diese Angabe bezieht sich auf alle in dem Modul befindlichen Tasks, gilt also modulweit ($1 \leq n \leq 8$). Defaultwert für n ist 1, es wird also das FPU–Register 0 gerettet.

`P=MPC604;` ist die Standardeinstellung für die Prozessoren 603/604 ohne Benutzung der Gleitkommaeinheit.

Floatdarstellung: **RTOS**–**UH**-Format

`P=MPC604+FPU(`*n*`);` ist die Einstellung für die Prozessoren 603/604 bei Benutzung der Gleitkommaeinheit. Mit Angabe der Zahl n wird die

Anzahl der zu verwendenden Gleitkommaregister vorgegeben, maximal möglich sind 32. Ist n Null, so wird die Gleitkommaeinheit nicht benutzt. Eine hohe Zahl erfreut zwar – je nach Problem – unter Umständen den Compiler, verschlechtert jedoch die Echtzeit-Performance des Systemes, da der zu rettende Kontext bei der Taskumschaltung ziemlich großvolumig werden kann. Der Compiler erhöht die angegebene Zahl bei Bedarf auf seine Mindestzahl (typ. 4 Register).

Floatdarstellung: IEEE-Format (Wenn n `> 0` ist)

`P=MPC405;` ist die Standardeinstellung für die PowerPC-Prozessoren IBM 405. Es wird Code generiert, der den sog. `lwarx/stwarx` flaw (CPU-Fehler) umgeht. Werden 405-Prozessoren mit normalen PowerPC-Compilern beschickt, ist mit sehr schwerwiegenden Fehlern beim Handling mit Semaphoren und Bolts zu rechnen! Der Code ist etwas weniger effizient aber korrekt auf normalen PowerPCs lauffähig.

Hinweis: Linken Sie auf gar keinen Fall Module mit unterschiedlichen `P= ...`-Steueranweisungen zusammen! Schwer zu findende Fehlfunktionen durch unterschiedliche Floatformate und/oder unterschiedliche Benutzung des Gleitkommarechenwerkes sind sonst möglich. Auch Laufzeitparameterfehler beim Prozeduraufruf im Test-Mode können dadurch entstehen.

5.1.2 Sprachliche Besonderheiten des UH–PEARL

Die Implementierung richtet sich nach dem „PEARL90 Sprachreport“, der bei der Fachgruppe 4.4.2 (Echtzeitprogrammierung, PEARL) der Gesellschaft für Informatik verfügbar ist. Aus diesem Report wird die neue DIN-Norm 66253 hervorgehen. Allerdings ist die zur Zeit freigegebene Version des PEARL-Compilers noch nicht vollständig mit dieser DIN-Norm 66253 kompatibel. Historisch bedingt gibt es auch noch Erweiterungen, die über die DIN-Norm und den Sprachreport hinausgehen. Sie wurden wegen der Portabilität auf Basis des `CALL`-Konstruktes realisiert (jedoch ohne den zeitaufwendigen Maschinencode), damit **RTOS**–**UH**–PEARL–Programme auch auf anderen PEARL-Systemen verwendbar gemacht werden können.

BEZEICHNER: Es sind — wie allgemein üblich — Ziffern, Kleinbuchstaben und Großbuchstaben erlaubt. Bezeichner dürfen jedoch nicht mit einer Ziffer beginnen. Zwischen Groß– und Kleinbuchstaben wird semantisch unterschieden. Die Bezeichner dürfen maximal aus 24 Zeichen zusammengesetzt werden. Das Zeichen Underscore („_ “) darf ebenfalls innerhalb von Bezeichnern benutzt werden.

SCHLÜSSELWORTE: Alle Schlüsselworte müssen in Großbuchstaben geschrieben werden (PEARL-Norm).

5.1.2.1 Datentypen im RTOS/PEARL

Typ	Länge	typ. Konstante/INIT
FIXED(1...15)	2 Bytes	21527 –500 100(15)
FIXED(16...31)	4 Bytes	471(31) 0(31) –2(31)
FLOAT(1...23)	4 Bytes	1.23456E–05 0.245
FLOAT(23...55)	8 Bytes	3.1414567893617(55)
CHAR(1...255)	n Bytes	'Abcdefgh–XYZ' 'a'
BIT(1...16)	2 Bytes	'01000100'B 'AFFE'B4
BIT(17...32)	4 Bytes	'AFFE1234'B4
DUR(ATION)	4 Bytes	2 HRS 5 MIN 0.4 SEC
CLOCK	4 Bytes	13:45:2.004
dur, clock		'Atom' ist 1 msec!
SEMA	2 Bytes	PRESET(2)
BOLT	2 Bytes	Initial immer „FREE“
STRUCT	? Bytes	INIT komponentenweise
REF typ	4 Bytes	INIT(Identifier)

Tabelle 5.1: Datentypen in **RTOS–UH**/PEARL

Bei den Verbunddaten (`STRUCT`) können Komponentennamen frei gewählt werden. Der Übersetzer verarbeitet jedoch nur max. 1023 gleichzeitig „lebende“ Verbundtypen: Verbundtypen, die bei Verlassen eines Blockes (Prozedur, Task, Begin/End-Block) ungültig wurden, belasten diese Bilanz nicht weiter. Gemeint ist ja auch nicht die Anzahl der Daten*objekte* sondern die Anzahl der Daten*typen*. Diese 1023-er Grenze ist daher in der Praxis kaum je relevant.

Mit Hilfe der `LENGTH`–Anweisung kann die Defaultlänge der Objekte `FIXED`, `FLOAT`, `CHAR` und `BIT` eingestellt werden. Ohne `LENGTH`–Statement gilt:

FIXED	=	FIXED(15)
FLOAT	=	FLOAT(23)
CHAR	=	CHAR(1)
BIT	=	BIT(1)

Defaultlängen der PEARL-Objekte

Achtung: Das `LENGTH`–statement wirkt auch auf die `FIXED` und `FLOAT` Konstanten ohne nachgestellte Länge!

Beispiel:

```
LENGTH FLOAT(55);    ! Defaultlaenge longfloat
DCL  X FLOAT;        ! Objekt ist longfloat
......
X=3.141567890123;    ! Konstante ist doppelt genau
```

Eine Übersicht über den Implementationsstand		
	Objekt	Abweichung
–	SIGNALE	Noch nicht implementiert
–	ARRAYS	Keine Total–E/A
–	ARRAYS	Keine Slices
–	DATION	Keine Stationsfelder, keine Untergliederung (CYCLIC ...), DCL nur in PROC/TASK
–	BIT/CHAR	Keine Slices
–	BIT	CAT für BIT nicht implem.
–	PROCS	Keine lokale Def. innerhalb PROCs/TASKs
–	REF_CHAR	Nicht implementiert
–	BY TYPE	Nicht implementiert
+	Arrays	Volle 32 Bit adress., Totalzuweisung
+	BIT/CHAR	Beliebige Expr./ im Selektor *xy*.CHAR(*expr*)
+	PROC(EDURE)	Der Selbstaufruf (Rekursion) ist erlaubt
+	Multi–module	Mehrere Systemteile können gebunden werden
+	E/A–Anweisung	Ausdr./Functions mit eigener E/A möglich
+	Hilf/Test	An/abschaltbare Hilfsfunktionen
+	TYPE	Definition auch für Grunddatentypen und Arrays erlaubt
+	Einbaufunktionen	Für Basisgrafik, Ein–/Ausgabe
+	ST(dation)	Statusabfrage Datenstation
+	SEMASET expr.	Dynamische SEMA-Reinitialisier

Tabelle 5.2: DIN/PEARL90–Abweichungen

Wichtiger Hinweis:

Eine Totalzuweisung von Verbunddaten (Strukturen) ist nur möglich, wenn sie vom gleichen benamten Typ sind. Gleiche Verbunddatentypen also stets mit gleichem `TYPE` deklarieren, bzw. spezifizieren! Als Prozedurwert ist daher logischerweise nur „ge`TYPE`ter“ Mode sinnvoll!

5.2 Preprozessor-Anweisungen

In den Übersetzer ist ein kleiner Pseudo-preprozessor eingebaut, der nicht wirklich einen extra Durchlauf erfordert, sondern begleitend zur Compilation in diese eingreifen kann.

Preprozessorbefehle müssen stets am Anfang einer Zeile stehen, allerdings dürfen sie eingerückt werden. (Wovon man bei geschachtelten `#IF` s auch Gebrauch machen sollte!)

Die Preprozessorbefehle lauten:

`#DEFINE ...`	Definiere eine benamte Konstante
`#INCLUDE ...`	File einbetten
`#IF ...`	Compiliere Folgetext wenn Bedingung nicht Null ergibt
`#IFDEF ...`	Compiliere Folgetext wenn Objekt existiert
`#IFUDEF ...`	Compiliere Folgetext wenn Objekt nicht existiert
`#ELSE;`	Alternativer Zweig zum `#IF`
`#FIN;`	Beendet Wirkung letztes `#IF..`

Die Preprozessorbefehle werden im folgenden einzeln erläutert.

5.2.1 Die Preprozessoranweisung DEFINE

Mit Hilfe dieses Preprozessorbefehles können benamte Konstanten mit Namen *identifier* definiert werden:

`#DEFINE` *identifier* `=` *xcompconstexpression*;

Die Wirkung entspricht compilerintern einer syntaktisch erweiterten Form der Anweisung

`DCL` *identifier* `INV FIXED INIT(`*xconstexpression*`)`;

Dabei steht *xcompconstexpression* für eine erweiterte Form von *xconstexpression* – wie unten erläutert. Ob eine Konstante vom Typ `FIXED(15)` oder `FIXED(31)` angelegt wird, entscheidet der Preprozessor an Hand des Zahlenwertes.

Die mit `#DEFINE` definierten Objekte dürfen innerhalb des PEARL-Textes überall dort benutzt werden, wo auch die Verwendung der mit `DCL` eingeführten benamten Konstanten erlaubt ist – zum Beispiel bei Feldgrenzenfestlegungen, in `CASE-` Konstrukten – und natürlich in den Preprozessor-`#IF`s. .

Innerhalb des Ausdruckes *xconstexpression* sind die 3 (keine Division!) Ganzzahlgrundrechenarten mit Klammerung erlaubt. Als Objekte sind dabei andere vorher definierte benamte Konstanten oder Zahlen zugelassen.

Der Ausdruck *xcompconstexpression* wird auf Basis von *xconstexpression* auf 4 alternative Arten gebildet (gezeigt an der benamten Konstante `Test`):

`#DEFINE Test =` *xconstexpression* oder
`#DEFINE Test =` *xconstexpression* `>` *xconstexpression* oder
`#DEFINE Test =` *xconstexpression* `==` *xconstexpression* oder
`#DEFINE Test =` *xconstexpression* `\=` *xconstexpression*

Die Vergleichsoperationen (größer, gleich, ungleich) sind nur auf der obersten Ausdrucksebene zugelassen und erzeugen Ganzzahlwerte, nämlich 1 wenn die Bedingung erfüllt ist und 0 wenn die Bedingung nicht erfüllt ist.

! → Steht ein `#DEFINE` innerhalb einer Task oder Prozedur, so ist die Gültigkeit des damit eingeführten Objektes genau wie bei einem lokalen DCL auf den Prozedur- und Taskblock beschränkt, in dem es definiert wird. Im Gegensatz zum DCL darf `#DEFINE` allerdings auch noch spät zwischen Prozeduren und Tasks ge-

setzt werden und wirkt dann dauerhaft für den Rest des Modules. Ratsam ist eine solche Verwendung jedoch allenfalls zum Nachdefaultieren mit Hilfe eines vorgelagerten `#IFUDEF`.

Beispiele:
```
#DEFINE rownumber = 400;
#DEFINE columns = rownumber*6;
#DEFINE arraysize = rownumber*columns;
#DEFINE largecase = arraysize > 32767;
```

5.2.2 Die INCLUDE-Anweisung

Bei diesem Preprozessorbefehl schaltet der Compiler vorübergehend seinen Input auf einen anderen File um. Im Übersetzerprotokoll macht der Compiler bei der Zeilennummer erkennbar, ob und im wievielten Level der Inclusion die protokollierte Quelltextzeile gefunden wurde.

Der Substitutionsmechanismus umfaßt stets komplette Zeilen. Das `INCLUDE`-Statement sollte darum **allein in einer Zeile** stehen. Wie bei allen Preprozessorbefehlen darf ihm lediglich ein Leerfeld vorausgehen, sonst moniert der Übersetzer einen Syntax-Fehler. Alle Zeilen des Textes im zu includenden File werden vom Compiler an Stelle der `INCLUDE`-Zeile bearbeitet. Das Statement hat folgende Syntax:

`#INCLUDE` *filepathlist*;

An der Stelle von *filepathlist* steht ein String, der von dem Betriebssystem, auf dem der Compiler gerade läuft, akzeptiert wird und einen File mit dem einzuschiebenden Text bezeichnet.

Wir nehmen einmal an, daß wir einen zu includenden File mit dem Namen `/H0/SPCs/Projekt1.P` haben, der folgende 2 Zeilen enthalte:

```
SPC Einw() STRUCT(/Name CHAR(20),Alter FIXED/) GLOBAL;
SPC Haus() STRUCT(/Strasse CHAR(30),No FIXED/) GLOBAL;
```

Das zu übersetzende Programm laute wie folgt:

```
MODULE Test;
PROBLEM;
DCL ...
#INCLUDE /H0/SPCs/Projekt1.P;
DCL ...
 ...
 ...
MODEND;
```

Wir nehmen an daß beim Compilieren des Hauptfiles das Protokoll eingeschaltet war, dieses sieht wie folgt aus:

```
=    1 MODULE Test
=    2 PROBLEM;
=    3 DCL ....
a    1 SPC Einw() STRUCT(/Name CHAR(20),Alter FIXED/) GLOBAL;
a    2 SPC Haus() STRUCT(/Strasse CHAR(30),No FIXED/) GLOBAL;
=    4 #INCLUDE /H0/SPCs/Projekt1.P;
=    5 DCL ...
 ...
```

Im obigen Beispiel wurde der zu "includende" File mit einer vollen RTOS-Pathlist bezeichnet. Wenn man unter **RTOS–UH** oder MS-DOS entwickelt, ist allerdings auch eine „relative" Fileangabe möglich: Bezugspunkt ist die Position in der File-Hierarchie, in der der File, der das `#INCLUDE` enthält, selbst steht. Eine relative Angabe liegt vor, wenn der String hinter `#INCLUDE` nicht mit dem Zeichen „`/`" beginnt.

Angenommener Bedienbefehl: `P /H0/TEXQ/ANALYS LO ...`

Dann führt ein `#INCLUDE DRAW` innerhalb des Files „`ANALYS`" zur Inclusion des Files, der unter `/H0/TEXQ/DRAW` steht.

Man kann sich im Filebaum sowohl in Richtung auf die Wurzel als auch hin zu den Blättern bewegen. Mit `#INCLUDE ../SS1/QU1` adressiert man bei obigem Beispiel den File, der unter `/HO/SS1/QU1` steht. Relative File-Inclusion ermöglicht den Transport des Systemes in andere Ordner oder auf andere Medien, ohne Änderung der Quelltexte. Auch eine Compilation des zusammengesetzten Textkonglomerates über das Netz ist damit ohne Eingriff in die Quellfiles möglich.

Wenn der Include-Text selbst wieder ein `#INCLUDE` enthält, so beginnt die Zeilennummerierung erneut bei 1, allerdings steht dann der Buchstabe „`b`“ am Zeilenanfang. Wir können also am Startbuchstaben erkennen, in der wievielten Ebene der Include-Staffelung wir uns befinden.

Enthält das `#INCLUDE` im included Text eine relative Fileangabe, so wird relativ auf den File, in dem dieses `#INCLUDE` steht, nach obigem Muster Bezug genommen.

Ob der included Text protokolliert wird oder nicht, hängt davon ob, wie der Compilerstatus bei Ausführung des `#INCLUDE` war. Wenn im included Text mit `/*+L */` oder `/*-L*/` der Status geändert wird, so hat das nur für die Zeilen des eingeschobenen Textes und als Startstatus für weitere Inludes im included Text Wirksamkeit. Der Compiler rettet seinen Protokollstatus beim `#INCLUDE` und restauriert ihn danach wieder.

! → Maximal können bis zu 8 Rekursionslevel bei der Tiefenstaffelung des `#INCLUDE`s bearbeitet werden. Natürlich darf eine solche `INCLUDE`-Kette nicht in sich selbst zurückführen.

5.2.3 Bedingte Kompilation: die Preprozessoranweisung IF

Mit dieser Anweisung kann der Compilerlauf in Abhängigkeit von Ausdrücken mit benamten Konstanten bestimmte Teile des Quelltextes ignorieren. Wenn die Bedingung wahr ist (was beim `#IF`-Argument einem Ganzzahlwert ungleich Null entspricht), so wird der folgende Text ganz normal übersetzt. Ein eventuell folgendes `#ELSE` unterdrückt dann den anschließenden Text bis zum `#FIN`. Ist die Bedingung unwahr (d.h. das `#IF`-Argument ergibt einen Ganzzahlwert von 0), so wird entsprechend umgekehrt verfahren.

Es gibt 3 verschiedene `#IF`:

`#IF` *xcompconstexpression*;
`#IFDEF` *identifier*;
`#IFUDEF` *identifier*;

Die Bedeutung von *xcompconexpression* wurde bereits auf Seite 287 genauer erläutert. Es handelt sich entweder um einen Ganzzahlausdruck oder um einen der 3 zulässigen Elementarvergleiche (`>`, `==` oder `\=`).

Beim `#IFDEF` ist die Bedingung erfüllt, wenn das Objekt mit Namen *identifier* dem Compiler bekannt ist, beim `#IFUDEF` genau dann, wenn es dem Compiler nicht bekannt ist.

Wenn der Übersetzer ein Protokoll anfertigt, so wird bei einem `#IF` mit nicht erfüllter Bedingung hinter dem `#IF`-statement Text zur Information ergänzt, wie man an folgendem Beispielprotokollauszug sieht:

```
=  123 #DEFINE Test = 3;
=  124 #IF Test > 5; [Condition is false]
```

Der Text in eckigen Klammern wurde vom Compiler generiert und unterdrückt gleichzeitig ein ggf. noch in der Zeile stehendes, aber nun totes PEARL-Statement.

! → Im Gegensatz zum `#INCLUDE` darf bei diesen Anweisungen hinter dem Preprozessorbefehl PEARL-Text stehen, der je nach Bedingung beachtet oder ignoriert wird. Natürlich dürfen dies keine Preprozessorbefehle sein, weil diese immer am Anfang einer Zeile stehen müssen.

Beispiele:

```
#IFUDEF  Arraysize;
   #IFDEF  defaultsize;
     #DEFINE  Arraysize=defaultsize;
   #ELSE ;
     #DEFINE Arraysize = 1000; ! Not-Default
   #FIN;
#FIN;  DCL Array(Arraysize) FIXED;
```

Mit einem kleinen Trick kann auch die Einhaltung bestimmter logischer Bedingungen oder von Leistungsbeschränkungen überwacht werden:

```
#IF Datalen > Frame; ! Test limit, skip if illegal
**** Wrong configuration: Datalen > Frame **** ; ;
#ELSE;  ! Compile normally
.....
#FIN;
MODEND;
```

Hier provoziert ein künstlich erzeugter Übersetzungsfehler die Ausgabe einer (PEARL-syntaktisch falschen) Hinweiszeile. Innerhalb der Zeile darf es kein Semikolon geben, am Ende muß mindestens eines (besser 2) stehen, damit der Compiler neu aufsetzen kann.

5.2.4 Bedingte Compilation: Schaltbarer Kommentar

Ein einfacheres Mittel als die umrahmende Verwendung von `#IF` und `#FIN` ist durch den „schaltbaren Kommentar“ (switched comment) gegeben. Die syntaktische Konstruktion dazu ist ein normaler Zeilenkommentar, der wie üblich durch das Zeichen „!“ eingeleitet wird. Allerdings kann durch den nachfolgenden Text die Wirkung des Zeichens „!“ in Abhängigkeit von Voreinstellungen aufgehoben werden:

`!:TS1  PUT x,y TO A1;` ist Kommentar, wenn TS1 nicht definiert ist,
ist Kommentar, wenn TS1 < 1 definiert ist,
PUT wird übersetzt, wenn TS1 > 0 definiert ist.

Man beachte, dass das Symbol hinter dem Doppelpunkt nur in der Liste der benamten 16-Bit Konstanten gesucht wird, dazu also bitte auf die Größe der definierten Konstanten achten. Wenn der Compiler die benamte Konstante erkannt hat, so ersetzt er in der erzeugten Ausgabeliste das Zeichen „:“ durch das Zeichen „-“ (= hinter ! steht Kommentar, Konstante < 1) oder das Zeichen „+“ (= das ! ist wirkungslos, Anweisung dahinter ist gültig weil Konstante $>$ 0). Diese Option ist ab den Compilerversionen 16.4 (Oktober 2003) implementiert. Von älteren oder fremden Compilern werden die Anweisungen stets nur als Kommentar interpretiert.

5.3 Globale Sondereinstellungen des Compilers

5.3.1 SETLINE, MAXERR und MODE

Im Quelltextbereich vor dem „MODULE“-Statement ist die modulglobale Vereinbarung bestimmter Betriebsmodi des Übersetzers möglich. Zwei davon betreffen die Programmgröße und den ROM-Mode. Sie werden auf Seite 295 gesondert beschrieben.

`MAXERR=10;` Bei Eintritt des 11.ten vom Compiler entdeckten Fehlers wird der Compilerlauf abgebrochen, mit der Meldung MAXERR-limit. Mit MAXERR=0 erfolgt beim ersten Fehler der Abbruch, usw.

`SETLINE=1000;` Die aktuelle Zeile des Compilerprotokolles erhält die Nummer 1000. Auch der Linemarker für die Fehlerdiagnose verwendet die so manipulierte Zeilenzählung. Gedacht war diese Option, um Fehler einfacher bestimmten Modulen zuordnen zu können. Beachten Sie bitte, daß der Linemarker auf Zeilennummern unterhalb von ca. 32000 beschränkt ist. Mit der 2003 eingeführten Modul-ID gibt es nun eine bessere Möglichkeit, das fehlerverursachende Modul durch einen erweiterten SETLINE-Befehl ausfindig zu machen:

`SETLINE=1,453;` Die Zeilennummerierung startet bei 1 und das Modul erhält die Nummer 453. Der Compiler erzeugt nun bei eingeschalteter Marker-Option (siehe Seite 299) zusätzlichen Code beim Beginn einer Prozedur und nach einem Prozeduraufruf, der die aktuelle Modul-ID auf eine Zelle im Task-Workspace schreibt. Diese sogenannte Modul-ID wird beim DL-Shellbefehl sowie im Fehlerfall ausgegeben. Allerdings muss dazu auf dem ausführenden System die zugehörige Systemoption eingeschaltet sein - aus Gründen der Kompatibilität zu alten Systemen ist das leider notwendig. Die Codeverlängerung ist meist nur sehr gering und tritt hinter den Vorteil zurück. Bitte verwenden Sie keine Modul-IDs, die oberhalb von 29999 liegen! Diese Nummern sind für bestimmte kommerzielle Softwarepakete reserviert.

`MODE=FULLCC;` Full character compare: Beim Vergleich von `CHAR`-Strings werden andere Hyperprozessorbefehle benutzt, die bei Längenungleichheit der Strings den kürzeren mit Blanks verlängern und den Vergleich über die gesamte Länge ausführen.

`MODE=NOLSTOP;` No line stop: An den Stellen des Programmes, die mit eingeschaltetem Line-Marker (siehe Seite 298) übersetzt werden, generiert der Compiler in diesem Mode nicht den üblichen Trap, sondern eine ganz erheblich schnellere „MOVE-Konstruktion“. Der zu zahlende Preis ist etwas längerer Code und der Verzicht auf den Zeilenstop beim Tracen des Programmes. Bei Fehlermeldungen erhält man aber nach wie vor die letzte überlaufene Zeile angezeigt. Auch das `DL`-Kommando funktioniert noch wie gewohnt. Gedacht ist diese Option für ausgetestete Programme, die in dieser Form an Kunden ausgeliefert werden können. Ohne Geschwindigkeitsnachteile erhält man so im Falle einer Fehlfunktion eine wertvolle Information vom Anwender.

`MODE=PAD;`
`MODE=NOPAD;` (No) Padding: Die neuen Compiler (ab 15.4-E) legen `FLOAT-` und `STRUCT-` objekte auf durch 4 teilbaren Adressen ab, da dies meistens höhere Geschwindigkeiten ergibt. Im „Padding“-Mode werden zusätzlich auch die relativen Ablagen innerhalb von Strukturen bei Floats und Structs auf durch 4 teilbare Werte erhöht. (Padding = Auffüllen mit blinden Bytes). Das Ausschalten des Modes ist nur zum Overruling nach Includes nötig, denn ohne Angabe ist der Padding-Mode nicht aktiv. Achtung: RISC-Prozessoren sollten möglichst im padding-mode laufen, denn sie verlieren wegen unklarer Zeigerinhalte sonst sehr viel Effizienz – auch wenn gar keine Strukturen benutzt werden! Denken Sie beim binären Schreiben und Lesen von Strukturen sowie beim Linken von Modulen daran, daß die Padding-Modes durchgängig gleichartig gesetzt sein müssen.

5.3.2 Modulgröße, ROM-Code

MODULGRÖSSE: Der einphasige Compiler kann den für den **RTOS–UH**–Lader erforderlichen Kopfeintrag der Modulgröße nur mit Hilfe des Programmierers schaffen. Dazu wird vor der `MODULE`–Anweisung ein spezielles Size–Statement plaziert:

1. `S=$6500; MODULE test; ...` (Form 1)
2. `SC=$6500; MODULE test; ...` (Form 2)

Die Hexzahl hinter `S` wird als Kopfeintrag dem Lader übergeben und ermöglicht ihm später die bedarfsgerechte Platzsuche. In der Form 2) wird der Speicherplatz vor dem Ladevorgang gelöscht (auf `$0000` gesetzt). Damit sind alle Variablen mit einem definierten Wert initialisiert (wohlgemerkt: nur direkt nach dem Laden, nicht vor jedem Start des geladenen Programms!). Besonders sinnvoll ist der Einsatz dieser Option bei der Erzeugung ROM–fähigen Codes mit dem PROM-Befehl (s. u.), da sonst u. U. Zufallsdaten platzraubend im EPROM deponiert werden.

Wird der `S`–Parameter nicht angegeben, so wird ein Ersatzwert von `$2000` eingesetzt.

Der Compiler prüft bei der Modulbilanz die Einhaltung der durch `S=$...` vorgegebenen Obergrenze. Wird sie überschritten, so wird ein `SIZE-LIMIT-ERROR` ausgegeben, das Programm kann später nur mit zusätzlichem `SZ`–Parameter beim `LOAD`–Befehl geladen werden (Sonst Fehlermeldung: `>>LOAD/`*xy*`: wrong address loader input`). Die Modulgröße ist praktisch nach oben nicht begrenzt, da der Compiler automatisch auf Langadressierung umschaltet. Allerdings gibt es bei der Länge des Innencodes von `REPEAT`-Blöcken, `IF/THEN/ELSE`- und `CASE`-Konstrukten eine Begrenzung auf 32 kB, die an dieser Stelle jedoch normalerweise nicht erreicht wird. Die Klippe kann durch eine vernünftige Modularisierung umschifft werden.

Schlußbilanz: Der PEARL–Compiler gibt am Ende der Übersetzung eine Bilanz über die Länge des erzeugten `VAR`iablen– und `CODE`–Teils aus. Falls bei Verwendung der `CODE/VAR`–Option der generierte Code frei verschieblich ist, wird zusätzlich die Information `SHIFTABLE` ausgegeben. Dies bedeutet, daß der `CODE`–Teil nicht unbedingt an der bei der Übersetzung angegebenen Adresse im EPROM abgelegt werden muß, es ist dann jede beliebige Ablageadresse im EPROM erlaubt. Die `SHIFTABLE`-Eigenschaft eines Modules geht durch das Setzen von Marken, Aufrufe von weit entfernten oder weiter hinten stehenden Prozeduren sowie durch globale Definitionen/Bezüge verloren.

ROM–CODE: Der Compiler erlaubt die **RTOS–UH**–kompatible Erzeugung ROM–fähigen Codes mit Trennung zwischen (Modul–) Variablen– und Code–Bereich. Der Compiler erkennt diese Betriebsart an der Angabe zusätzlicher Code– und Variablen–Adressen, die allerdings vom Linker (nicht jedoch vom `PROM`-Befehl) überschrieben werden können:

`S=$`*`size`*`,CODE=$`*`epromadresse`*`,VAR=$`*`ramadresse`*`;` oder:
`SC=$`*`size`*`,CODE=$`*`epromadresse`*`,VAR=$`*`ramadresse`*`;`

Die Hexzahl *epromadresse* gibt die Startadresse des Codes im EPROM an, die Hexzahl hinter `VAR` die Adresse, mit der beginnend die Modulvariablen im RAM abgelegt werden sollen. Beim EPROM– und RAM–Layout orientiert man sich an den Code– und Variablen–Längenangaben des Compilers.

Der mit dieser Option erzeugte Code ist nicht im RAM ablauffähig; er kann jedoch vom Linker ggf. mit anderen Modulen zusammen zu EPROM-tauglichen `S`-Records konvertiert werden. Wie in der Anfangszeit von **RTOS–UH** ist es aber auch weiterhin möglich, nach Laden solcher S-Records mit Hilfe des `PROM`–Befehls aus ihnen Eprommer–geeignete S-Records zu machen.

5.3.3 Codegenerierung unterdrücken

Die Codegenerierung kann nur global unterdrückt werden. Dies geschieht bei der Aktivierung des Compilers durch `CO NO` (siehe dazu Seite 180). Eine wesentliche Zeitersparnis ist damit normalerweise zwar nicht verbunden, man erspart sich aber die Bereitstellung einer CO–Datei, wenn man zunächst nur an der syntaktischen Prüfung seines Programmes interessiert ist.

5.4 Lokale Hilfs– und Testmodi des Compilers

Neben den rein „global“ einstellbaren Modes gibt es mit Hilfe besonders aufgebauter Kommentarzeilen Möglichkeiten für „lokale“ Einstellungen. Der einzige global **und** lokal einstellbare Mode ist die Übersetzung mit bzw. ohne Übersetzungsprotokoll. Dagegen kann die einmal global an- bzw. abgeschaltete Codegenerierung nicht mehr lokal beeinflußt werden. Typische rein lokale Modes sind etwa die Markiereroption, die Testoption sowie die Maschinenkodeprotokollierung (s. u.). Einige der Modes nehmen nach Ende eines `#INCLUDE`-Files wieder ihren alten Zustand vor der Inklusion ein.

Als steuernde Kommentare kommen nur die mit `/* .... */` umrahmten Sequenzen in Frage. Der Zeilenkommentar (mit ! eröffnet) ist dafür nicht geeignet. Findet der Compiler einen Kommentar, der mit `/*+` oder `/*-` beginnt, den er aber nicht als Steuerkommentar versteht, so wird in der Compilerbilanz eine Warnung erzeugt. Diese Warnung enthält die letzte Zeilennummer, in der ein solcher unverständlicher Steuerkommentar gefunden wurde. Man sollte also z.B. zum Ungültigmachen des Protokollswitches (s.u.) aus `/*+L` nicht `/*+ L` sondern `/* +L` machen.

5.4.1 Übersetzungsprotokoll ein–/ausschalten

Es wird (siehe Seite 180) global durch `LO NO` aus– bzw. mit `LO` */device/file* oder fehlender `LO`–Parameter eingeschaltet. Für die lokale Steuerung ist eine Kommentaranweisung

`/*+L ... beliebiger Text */;` zum Einschalten, bzw.
`/*-L ... beliebiger Text */;` zum Abschalten vorgesehen.

Unabhängig davon, ob das Übersetzungsprotokoll eingeschaltet ist oder nicht, werden Programmfehler in jedem Fall in das `LO`–Medium ausgegeben. Auch die fehlerhafte(n) Zeilen erscheinen mitsamt dem Fehlerzeiger. Wird diese Option innerhalb eines `#INCLUDE`-Files benutzt, so gilt sie nur für den eingebundenen Text, bzw. weitere unterlagerte Inklusionen. Nach Ende des Files wird der alte Mode wieder eingestellt.

5.4.2 Codeprotokollierung ein–/ausschalten

Urpsrünglich für die Überprüfung der korrekten Compilerfunktion gedacht, dann aber im System belassen, existiert eine Option zur Auflistung des generierten Maschinen– bzw. Hyperproccodes. Sie wird mit

`/*+P ... beliebiger Text */;` eingeschaltet und mit
`/*-P ... beliebiger Text */;` wieder ausgeschaltet.

In das Übersetzungsprotokoll eingebettet, erscheint bei dieser Option für jeden generierten Befehl eine Zeile. Dabei wird der auf den Modulanfang relativierte Programmzähler, der Befehlmnemo sowie die Liste der Operanden (unter Verwendung der PEARL–Namen!) ausgegeben. Der generierte Code gehört jeweils zur nächsten protokollierten PEARL–Programmzeile, wird also quasi mit dieser abgeschlossen. Die Hyperprocbefehle sind im Teil für Systemprogrammierer weiter hinten erläutert, die 68000– bzw. PowerPC-Maschinenbefehle erscheinen mit modifizierten Mnemos, etwa `ADDX` statt `ADD.L` (`X`=xtend 32 bit), können in der Regel aber leicht identifiziert werden. Die entsprechende Liste befindet sich auf Seite 599 im Abschnitt 8.7. Mit „`>>`“ versehene Operationen sind sog. „Loader-messages“, die i. a. nicht interessieren dürften. Dieser Mode nimmt mit dem Ende eines `#INCLUDE`-Files wieder seinen alten Zustand ein.

5.4.3 Markierungsoption ein–/ausschalten

Der UH–Compiler ermöglicht mit Hilfe dieser Option die statische und dynamische Einbettung von PEARL–Zeilennummern und ggf. Modul-IDs (siehe Seite 293) in das generierte Maschinenprogramm. Damit wird z. B. der Zeilenstop auf Hochsprachebene (siehe TRACE, Seite 216) ermöglicht, aber auch die Ausgabe der letzten exekutierten PEARL–Zeile – bei erweitertem `SETLINE`-befehl auch der Modul-ID – im Falle von Laufzeitfehlern vorbereitet. Die Kommentare

`/*+M ... beliebiger Text */;` dienen zum Ein- sowie mit
`/*-M ... beliebiger Text */;` zum Abschalten der Option.

Zeilen, die nur ein Fragment einer mehrzeiligen Anweisung enthalten, können nicht markiert werden. Der Compiler setzt vor den Code der ersten PEARL–Anweisung jeder Zeile, die im Bereich eingeschalteter Option liegt, einen Spezialbefehl. Dieser besteht aus einem TRAP oder Hyperprocbefehl mit nachfolgender Zeilennummer, die mit Hex–Digits dezimal zu lesen ist. Aufgabe dieses Befehles ist, die Zeilennummer in eine spezielle Zelle der Task zu schreiben und dabei zu prüfen, ob für die Zeile ein Zeilenstop vorliegt. Wenn es der Platz erlaubt, sollte möglichst die komplette zu prüfende Task mit eingeschalteter Option übersetzt werden, da sonst leicht Fehlinterpretationen über den Fehlerort möglich sind. Da die Zeilennummernkodierung aus Kompatibilitätsgründen zu älteren („PEARL80“) Systemen etwas eigenwillig erfolgen muß, können Zeilennummern größer als 34575 nur modulo 30000 kodiert werden, d.h. die Zeile 34566 erscheint als 4566 usw. Es ist besser, die Modul-ID (siehe Seite 293) zu verwenden, statt – wie früher üblich – mit großen vorab vergebenen Zeilennummernblöcken zu arbeiten. Die Geschwindigkeitsverluste durch diese Option sind im Allgemeinen nur bei wenigen Prozent zu vermuten, können jedoch in Sonderfällen untragbar hoch werden. Die `+M`–Option ermöglicht mit Hilfe des `DL`–Befehles (S. 131) jederzeit Schnappschüsse der aktuellen Zeilennummer laufender Tasks.

Hinweis:

Wenn der Compiler mit `MODE=NOLSTOP` arbeitet, so tritt durch die `+M`-Option eine erheblich geringere Verlangsamung als im Normalfall ein. Allerdings ist dann das oben erwähnte Zeilentrace (Zeilenstop) nicht möglich. Diese Art der eingeschränkten Zeilenmarkierung hat den Vorteil, daß sie in vielen Fällen dauerhaft im Programm belassen werden kann. Auch später können dann Betriebsfehler noch gut analysiert werden.

5.4.4 Seitenvorschub im Protokoll erzeugen

Mit Hilfe dieser Option kann im Programmprotokoll des Compilers auf einem Drucker ein Seitenvorschub erzeugt werden. Damit können Programmdokumente übersichtlicher gestaltet werden.

`/*+N ... beliebiger Text */` Seitenvorschub

Nur mit dem Zeichen `+` vor dem `N` erfolgt die gewünschte Aktion. Steht die Kommentarzeile als einzelne Zeile zwischen (und nicht innerhalb) von PEARL-Anweisungen, dann wird sie im Protokoll normalerweise oben auf der neuen Seite gedruckt.

5.4.5 Index–, Selektor– und Parametertest aktivieren

Mit Hilfe dieser `T`–Option generiert der Compiler einen modifizierten Code, der zur Laufzeit die Einhaltung von Zugriffsgrenzen und die Korrektheit von Prozedurparameterlisten überprüft. Nach Ende eines `#INCLUDE`-Files kehrt bei dieser Option der alte Zustand vor der Inklusion zurück.

`/*+T ... beliebiger Text */;` Testmode einschalten.
`/*-T ... beliebiger Text */;` Testmode ausschalten.

Feldindex– und Characterselektor prüfen:

Bei mehrdimensionalen Feldern wird nicht jeder einzelne Index, sondern nur der effektive 32 Bit lange eindimensionale Resultatindex überprüft. Damit kann die Zerstörung von Code oder Daten ausserhalb des Feldes bzw. der Zeichenkette mit Sicherheit unterbunden werden. Charakterselektoren werden ebenfalls überprüft, nicht jedoch Selektoren in Bitstrings. (Die Überschreitung von Bitselektorgrenzen erzeugt falsche Ergebnisse, richtet aber keinen Schaden an).

Bei Verletzung der durch das originäre `DCL` fixierten Grenzen wird eine Systemmeldung abgesetzt, die entweder die letzte registrierte Zeilennummer oder — falls nicht vorhanden — die Speicherstelle des Zugriffscodes enthält. Die Meldung enthält einen Hinweis, ob der Index bzw. Characterselektor zu groß oder zu klein ist („Overflow“ bzw. „Underflow“). Die Task wird angehalten, kann aber mit `CONTINUE` fortgesetzt werden; der Zugriff erfolgt dann ersatzweise auf das erste Element des Feldes bzw. der Zeichenkette.

Prozeduraufrufparameter prüfen:

Bei allen Prozeduraufrufen überprüft der Compiler schon zur Compilezeit, ob jeder einzelne Parameter mit der ihm bekannten Definition oder Spezifikation verträglich ist. Nun ist aber immer noch möglich, daß der Programmierer eine falsche Spezifikation externer Prozeduren kodiert hat. Ist der Testmode eingeschaltet, so erfolgt ein zusätzlicher Test zur Laufzeit, indem bei jedem Funktions- und Prozeduraufruf der Parameterprüfeinstieg der Zielprozedur angesprungen wird. Wenn die Zielprozedur vom PEARL-Compiler erzeugt wurde, oder korrekt in Assemblersprache abgefaßt wurde, so erfolgt dabei jetzt ein sogenannter „Signaturcheck“: Die Definition jeder Prozedur wird nach einem Algorithmus in eine 32 Bit lange „Signatur“ umgerechnet, die mit sehr großer Wahrscheinlichkeit bei anderen Definitionen anders ausfällt. Stimmt die an der Aufrufstelle für die Prozedur (bzw. des Zeigers auf eine Prozedur) gültige Signatur (aus z.B. `SPC .... GLOBAL`) nicht mit der am Zielort aus der tatsächlichen Definition der Prozedur errechneten Signatur überein, so erfolgt eine Laufzeitfehlermeldung „`... wrong parameterlist`“.

Die Ursache für den Fehler ist fast immer die oben erwähnte falsche Spezifikation der externen und später angelinkten Prozedur oder eine falsch besetzte `REF`-Variable (Prozedurzeiger). Eine weitere mögliche Ursache kann sein, daß auf Aufrufer- und Prozedurseite unterschiedliche Gleitkommaformate benutzt werden.

Wichtiger Hinweis:

Wenn eine Task nach einem Parameterfehler vom System angehalten wurde, so sollte sie auf gar keinen Fall fortgesetzt werden. Das System kann anders als bei einem Indexfehler hier keine Notreparatur durchführen. Wurden zum Beispiel zu kleine Strukturen angeboten o.ä., so droht bei Fortsetzung der Task eine Zerstörung wichtiger Speicherzellen. Der Effekt solcher Veränderungen kann dann erst sehr viel später auftreten.

Eine Parameterüberprüfung bei C-kodierten Unterprogrammen ist mangels entsprechender Vorrichtungen in den C-Compilern nicht möglich.

5.4.6 EPROM–Prozedur erzeugen

Es können Prozeduren global gemacht werden, um dann im EPROM oder auf der Boot–Diskette als beständiger Teil des Systemes mit abgelegt zu werden. Der Lader kann diese Prozeduren dann finden, wenn die Referenz nicht schon vorher durch seine Inputdateien befriedigt wurde.

```
xyz:PROC /*+G*/ (.....); ! erzeugt 14er Scheibe
! Siehe dazu auch Scheibenkonzept Kapitel 9 Seite 625
```

Der Kommentar `+G` muß hinter dem Schlüsselwort `PROC` bzw. `PROCEDURE` kommen, nur dann wird er berücksichtigt und bewirkt die Erzeugung einer 17-er Scheibe für das Symbol `xyz` des obigen Beispiels.

Wichtiger Hinweis:

Die Prozedur wird durch die Angabe der `+G`–Option nicht im PEARL-Sinne global gemacht, dies muß, wie in PEARL vorgeschrieben, mit dem Schlüsselwort `GLOBAL` erfolgen.

Es kann mit dieser Option eine ganze Bibliothek von Prozeduren erzeugt werden, die dann im EPROM oder Boot–Bereich abgelegt werden können, damit sie global zur Verfügung stehen. Allerdings werden nachträglich ins verwaltete RAM geladene Module nur bei Benutzung von `LOADX` berücksichtigt.

5.4.7 Prozedurparameterstrukturanalyse unterdrücken

Im Normalmode wird bei jedem Prozeduraufruf bei der Übergabe von Datenstrukturen (`STRUCT[...]`) an Prozeduren schon zur Compilezeit überprüft, ob die aktuelle Struktur aus der gleichen Typdefinition wie der Formalparameter hervorgegangen ist. Im alten PEARL80 wurde nur zur Laufzeit und nur die Größe (in Bytes) der Formal- mit der Aktual-Struktur verglichen (max 32kB im alten System). Um die Übertragung alter PEARL80-Programme zu erleichtern, kann die Compilezeitprüfung, ob Aktualstrukturen als Parameter zugelassen sind, auf diesen Stand reduziert werden. Gegenüber der Verwendung von `VOID`-Pointern hat man dann immer noch den Vorteil, daß garantiert die Objektgröße genau paßt.

```
/*-S ... beliebiger Text */; Reduzierte Prüfung.
/*+S ... beliebiger Text */; Wieder volle Prüfung.
```

! → Dieser Mode kehrt nach einem `#INCLUDE` **nicht** in den alten Zustand zurück.

```
xy:PROC(A STRUCT[a FLOAT(23),b FIXED(15)]);
.........
DCL X STRUCT[x FLOAT(23), y CHAR(2)];
/*-S  Abschalten des pingeligen Tests */
CALL xy(X);
```

Im Normalmode würde der Compiler diesen Aufruf zurückweisen, jetzt akzeptiert er ihn, weil die Objekte in der Größe passen.

5.4.8 Prozedurarbeitsspeicher reservieren

In unserem PEARL-System gibt es bekanntlich keinen „Stack“ und damit im Gegensatz zu archaischen C–Systemen auch keinen Stacküberlauf. Wenn Prozeduren aufgerufen werden, so bietet der Compiler ihnen zunächst den aktuell noch freien Platz des Aufrufers an. Reicht dieser nicht aus, so holt die Routine sich bedarfsgerecht Speicher über den entsprechenden RTOS-Trap. Beim Verlassen der Routine wird dieser wieder zurückgegeben. Jede Task erhält vom Compiler standardmäßig etwa 1 kByte zusätzlichen Speicher im sog. A5-Space, der für von dieser Task ausgehende Prozeduraufrufe vorgesehen ist. Durch vor dem Prozeduraufruf angelegte und wieder verlassene `BEGIN-` ...`END-` oder `REPEAT-` Blöcke kann der für Prozeduren verfügbare Speicherbereich im Aufrufmoment allerdings beliebig größer als 1 kB sein.

Mit der hier beschriebenen R-Option kann der Wert für die zusätzliche Reservierung von 1 kB auf jeden beliebigen Wert verändert werden. Damit kann man bei Kleinstanwendungen Platz sparen oder aber beim Aufruf von Prozeduren mit mächtigen lokalen Datenbereichen den Aufruf des Speicherhol-Traps verhindern, was zu einer höheren Ablaufgeschwindigkeit führt. Eine kleine Minimalreserve von etwa 40 Bytes kann man jedoch nicht unterdrücken, auch nicht durch Setzen von R=0.

`/*+R=`*`hexazahl`* `nach Luecke bel. Text */;`
`/*-R=`*`hexazahl`* `nach Luecke bel. Text */;` gleiche Wirkung.

hexazahl ist dabei eine max. 8-stellige Hexadezimalzahl mit oder ohne führendes `$`-Zeichen am Anfang. Sie muß ohne Zwischenraum direkt hinter `/*+R=` oder `/*+R` stehen und mit einem Blank abgeschlossen werden.

Da es sich um Kommentar handelt, werden syntaktische Fehler nicht angezeigt, die Wirkung unterbleibt also u.U. ohne Warnung. Der eingestellte Wert gilt bis zum Ende des Modules für alle Tasks. Er kann jedoch am Anfang jeder Task (nach oder in der Definitionszeile) neu gesetzt werden. Der Compiler wertet den aktuell eingestellten Wert bei der Bearbeitung des zum Ende der Task gehörenden `END` aus.

Beispiel:
```
/*+R=$5000 20 kByte vorhalten */
/*+R5000   Gleiche Wirkung    */
/*+R$5000      ''             */
```

5.4.9 Konstantenpool leeren

Der Compiler verwaltet normalerweise autonom seinen Konstantenpool derart, dass es nicht zu oft zu eingestreuten Konstantenblöcken kommt, über die der Programmcode mit Sprungbefehlen hinweg kommen muss. Trotzdem kann es in extremen Situationen – eine Formel erstreckt sich über hunderte von Zeilen - dazu kommen, dass Konstanten nicht mehr als PC-relative 16-Bit Adressen erreichbar sind und der Lader / Linker das Programm nicht montieren kann. Das Problem besteht nur beim 68K-Prozessor. Durch die eingeschobene Zeile

Beispiel:
```
/*+F Flush constants */
```

gewinnt man durch das erzwungene Leeren des Konstantenpools – im günstigsten Fall – maximal einige wenige KB, um die dann die nachfolgende Anweisung im 68k-Code länger sein kann.

5.4.10 Default-PRIO setzen

Wenn bei der Definition von PEARL-Tasks keine Priorität angegeben wird, so setzt der Compiler den Wert auf den Defaultwert von 48. Dieser Defaultwert kann jedoch verändert werden durch folgenden Kommentartext:

Beispiel:
```
/*+D=1000 Set def. PRIO to decimal 1000 (low) */
```

Alle folgenden Task-Deklarationen ohne Prioritätsangabe verwenden nun den neuen Defaultwert. Natürlich kann eine weitere Zeile später den Wert wieder anders besetzen. Das Gleichheitszeichen ist optional. Ist die Angabe durch syntaktische Fehler für den Compiler unverständlich, so behandelt er sie ohne Fehlermeldung wie einen Kommentar. Es wird aber in der Compilerbilanz eine entsprechende Warnung ausgegeben. Der zulässige PRIO-Bereich ist 1 ... 32767.

Wichtig: In Shellmodulen kann dieser Mechanismus vor der Definition einer Shell-Proc benutzt werden, um die Defaultpriorität des Shellprozesses vorzugeben.

5.5 Umgang mit Datenstationen in PEARL

5.5.1 Festlegungen im Systemteil

Die heutigen PEARL-Compiler können alle möglichen Datenstationen des Zielsystemes an Hand der im Zielsystem gültigen Bezeichner im Systemteil einbinden. Erreicht wird dieses dadurch, daß der Lader alle dem Compiler unbekannten Gerätebezeichner an Hand der Liste innerhalb des Zielsystemes in die richtigen `LDN/DRIVE`-Gespanne umsetzt. Wenn der Lader nicht fündig wurde, so reagiert er mit einer Fehlermeldung. Im Gegensatz zu den früheren Compilern sind nun nur noch wenige Bezeichner dem Übersetzer bekannt.

Auch wenn die Compiler immer noch die ältere Syntax mit Doppelpunkt oder Punkt als Separator zwischen Gerätebezeichner und Pathlist beherrschen, so sollte man bei Neuprogrammierung nur noch die Form mit „`/`“ als Separator verwenden. Nur diese Syntax wird hier beschrieben. Allgemein gilt:

XYZ`:/`*device*`/`*pathlist*`(TFU=`*int*`,NE,MB=$`*hexno*`,AI=$`*hexno*`)->;` oder
XYZ`:/`*device*`/`*pathlist*`(TFU=`*int*`,NE,MB=$`*hexno*`,AI=$`*hexno*`)<-;` oder
XYZ`:/`*device*`/`*pathlist*`(TFU=`*int*`,NE,MB=$`*hexno*`,AI=$`*hexno*`)<->;`

Hinweis:

> Benutzen Sie möglichst keine Zusatzparameter `MB` oder `AI` bei Dations, die mit `<->` in beiden Richtungen betrieben werden

In solchen Fällen kann es Konflikte bei der Bedeutung der Mode-Bits geben. Abhilfe ist möglich durch eine Aufteilung auf zwei richtungsgebundene `DATION`s.

XYZ Logischer Name im Programm

device entweder compilerbekannter Name (s. u. z. B. `A1`, `PP`, `ED`) oder Gerätename aus dem Zielsystem oder `LD/`*`ldn.drive/`* mit

ldn Warteschlangennummer im Betriebssystem **RTOS–UH**.

drive Laufwerksnummer (Untergliederung) des Gerätes. Wenn diese Angabe samt Punkt fehlt, ist drive=0.

pathlist Alphanum. string mit Separatoren „/“, dient eventuell nur als Platzhalter für `OPEN ... BY IDF(...)`, dann lang genug vorsehen!!

`TFU` Größe des Communication–Elements, `TFU=1` kann für Dialoggeräte sinnvoll sein (Terminal), belastet aber das Betriebssystem mit großem Overhead. Wenn `TFU` fehlt, so wird der Wert von 128 angesetzt.

`NE` No Error–Message–Flag. Der Programmierer wird mit der Funktion `ST` die Fehlerüberwachung in eigener Regie durchführen. Fehlt `NE`, so werden Fehlermeldungen ausgegeben.

`MB=$` Es werden die Mode–Bits des Communication–Elements gesetzt! Bei `AI` muß dann das linke Byte null sein, sonst gibt es eine Fehlermeldung. Hexno. ist 2 stellig (keine TIMEOUT Angabe).

`AI=$` Zusatzinformation `$`*xxxx* für E/A–Treiber laut besonderer Beschreibung. Fehlt `AI=`, so wird ein Wert von Null angenommen. Das linke Byte toggelt beim Laden (exklusiv–oder) die korrespondierenden MODE–Bits des Communikation–Elements, das rechte ist zur Angabe eines Timeouts. Hexno. 2 (ohne TIMEOUT) oder 4 stellig (inklusive TIMEOUT–Angabe).

Die Klammer `(TFU ... AI=)` darf wie einzelne Parameter fehlen, die Reihenfolge der Parameter muß aber eingehalten werden. Es muß eigentlich keine Station über `LD/../` definiert werden. Wie oben erwähnt, kann der Lader dem Compiler unbekannte Symbole später aus den Tabellen im Zielsystem selbst in *ldn/drive* umwandeln. Am Ende des Compilerlistings bei der Schlußbilanz gibt der Compiler alle sogenannten „Extra–Devices“ aus. Diese müssen natürlich im Zielsystem vorhanden sein.

Folgende Gerätebezeichner sind dem PEARL–Compiler vorab bekannt und müssen darum in allen Zielsystemen auf der gleichen `LDN/DRIVE` liegen:

Gerätename	Bedeutung	Ldn	Drive
A1	Console	0	0
A2	Serielles Port	2	0
ALDV	Actual load device Lader setzt Ldn/Drive	?	?
B1	Console buffered	0	2
B2	Serielles Port buffered	2	2
C1	Console scanned	0	6
C2	Serielles Port scanned	2	6
ED	ED–Filesystem	0	0
NIL	Schwarzes Loch	15	0
PP	Printer Parallelport	10	0
TY..	Aktuelles Terminal, s.u.	?	?
VI	Virtual Input	8	0
VO	Virtual Output	7	0
XC	Remote command	9	0

Tabelle 5.3: Compilerbekannte Gerätebezeichner in PEARL

Die Datenstationen mit Namen `/TY, /TYA, ...` etc. werden ebenfalls zur Ladezeit durch das aktuelle Terminal des Nutzers ersetzt, wobei allerding nur `LDN` und `DRIVE` übernommen werden, nicht eine evtl. vorhandene Pathlist.

Es ist zulässig, Usernamen gleich Systemnamen zu setzen, z. B. genügt `...;A1;` statt etwa `...;A1:/A1<->;`

5.5.2 Beschreibung AI und MB-Parameter

Mit dem `AI`– oder `MB`–Parameter besteht die Möglichkeit, die Ein–/Ausgabe umzuparametrieren. Dabei ist zu beachten, daß zwischem dem Verhalten des Laufzeitsystems und des Device–Treibers Unterschiede bestehen können! Das Verhalten des Laufzeitsystems ist ab Seite 313 beschrieben, die Beschreibung der Device-Treiber beginnt auf Seite 389. Um zu verdeutlichen, was gemeint ist, folgendes Beispiel: Es soll eine `CHAR`–Variable eingelesen werden. Die Eingabe soll mit einem LF beendet werden, es wird ein entsprechendes CE generiert. Der Device–Treiber erkennt ein LF, für ihn ist der Auftrag erledigt, er gibt das CE an das Laufzeitsystem zurück. Für das Laufzeitsystem ist die Eingabe erst bei der entsprechenden Anzahl Zeichen oder einem CR beendet, es schickt das CE wieder zum Device–Treiber, wo erneut auf eine Eingabe gewartet wird. Zusammenfassend kann festgehalten werden, daß das Laufzeitsystem i. a. nur ein CR als vorzeitiges Eingabeende erkennt.

Um die Ein-/Ausgabekanäle getrennt zu beeinflussen, ist es sinnvoll, jeweils getrennte DATIONs zu vereinbaren. Die Vorbesetzung des CE–Mode Bytes ist abhängig von dem Deviceparameter (siehe SD–Befehl 1.Byte).

	Output	Input
für ein Device mit Echo :	$40	$B8
für ein Device ohne Echo :	$40	$BA

Merke: Mit AI werden die selektierten Bits getoggelt, mit MB dagegen direkt gesetzt. MB beeinflußt nur das Mode-byte, während mit AI im rechten Byte Time-out-parameter beeinflußt werden können.

Ausgabe

Für Ausgaben (->) erzeugt der Compiler ein CE, in dem im oberen Mode–Byte nur das Bit für die Output–Direction gesetzt ist. Es gelten also keinerlei Ende–Bedingungen, der Transfer wird über die tatsächliche Länge gesteuert. Die Ausgabe wird ohne „WAIT“ durchgeführt, d. h. das Programm wartet nicht auf das Ende der Ausgabe, sondern läuft weiter. Folgende Veränderungen dieser Standardeinstellung sind denkbar:

AI=$8000
MB=$80
: Das Wait–Bit wird gesetzt. Das Programm wartet auf das Ende einer Ausgabe, bevor es fortgesetzt wird. Der erfolgreiche Abschluß der Ausgabe kann mit der ST–Funktion überprüft werden.

AI=$4000
MB=$40
: Diese Umstellung auf Eingaberichtung ist unsinnig und kann vom System nicht richtig umgesetzt werden

AI=$2000
MB=$20
: Die Ausgabe wird vorzeitig abgebrochen, wenn im Ausgabetext ein CR ($13) entdeckt wird.

AI=$1000
MB=$10
: Die Ausgabe wird vorzeitig abgebrochen, wenn im Ausgabetext ein LF ($10) entdeckt wird.

AI=$0800
MB=$08
: Die Ausgabe wird vorzeitig abgebrochen, wenn im Ausgabetext ein EOT ($04) entdeckt wird.

AI=$0400
MB=$04
: Bei seriellen Schnittstellen: Es wird verhindert, daß mit Ctrl A, Ctrl B oder Ctrl C das Bedieninterface „aufgeweckt“ wird. Wenn über eine serielle Schnittstelle Ein– und Ausgaben durchgeführt werden, wobei in den Eingabestrings ^A/B/C vorkommen kann, so muß dieses Bit auch bei der Ausgabe gesetzt werden.

  Bei Floppy/Winch: Der Autoclose des Filemanagers wird unterdrückt, d. h. die Datei wird nicht mit dem Lesen des letzten Bytes geschlossen.

`AI=$0200` `MB=$02` Für die Ausgabe ohne Bedeutung, aber — s. o. — wenn die Eingabe ohne Echo läuft, muß das Bit auch bei der Ausgabe gesetzt sein.

`AI=$0100` `MB=$01` Die Ausgabe wird binär durchgeführt, d. h. es werden alle Zeichen (von `$00` bis `$FF`) übertragen. Es wird nicht mehr auf X_{on}/X_{off} reagiert, nur der Hardware–Handshake (RTS/CTS) bleibt erhalten, falls die Hardware dazu in der Lage ist.

Eingabe

Für die Eingabe (`<-`) wird ein CE mit dem Standardmodebyte `$B8` oder `$BA` erzeugt. Damit wird auf das Ende der Eingabe gewartet (`$80`); die Eingabe wird mit einem Cr (`$20`), einem Lf(`$10`) oder einem Eot (`$08`) vorzeitig beendet. Bei Geräten, deren Eigenschaft, die explizite Unterdrückung des Echos zuläßt, wird zusätzlich `$02` aufgeordert, ansonsten sind die rechten 3 Bit zunächst nicht gesetzt, können aber (s.u.) aktiviert werden.

`AI=$8000` `MB=$80` Das Programm würde nicht auf das Ende der Eingabe warten. Wird vom System unterdrückt, da ein Programm i. a. auf eine Eingabe reagiert!

`AI=$4000` `MB=$40` Wird vom System abgefangen. Ein `AI` wirkt nun wie ein `MB`.

`AI=$2000` `MB=$18` Die Eingabe wird nicht mehr mit einem Cr (`$13`) beendet.

`AI=$1000` `MB=$28` Die Eingabe wird nicht mehr mit einem Lf (`$10`) beendet.

`AI=$0800` `MB=$30` Die Eingabe wird nicht mehr mit einem Eot (`$04`) beendet.

`AI=$0400` `MB=$x4` Serielle Schnittstelle: Mit einem `^A`, `^B` oder `^C` wird der Kommandoprozessor nicht mehr aufgeweckt. Diese Zeichen werden wie normale Zeichen behandelt. War bei der letzten Eingabe eines Programms dieses Bit gesetzt, ist der Kommandoprozessor auch weiterhin gesperrt! Sie kommen jetzt als User nur mit einem `BREAK` wieder in das System.

Floppy/Winch: Der Autoclose des Filemanagers wird nicht durchgeführt.

`AI=$0200`
`MB=$02`
Je nach Parametrierung der Schnittstelle wird das Echo ein oder ausgeschaltet, bzw. bei Verwendung von `MB` definitv ausgeschaltet. Vorsicht beim Filemanager: die Bits haben dort andere Bedeutungen! Wenn nicht sicher ist, welche Schnittstellenparameter beim Start des Programms gelten, sollte zuerst ein `SD`–Befehl über den `/XC/` bzw. mit Hilfe der `EXEC`-Funktion abgesetzt werden.

`AI=$0100`
`MB=$01`
Es wird auf binären Transfer umgeschaltet, d. h. es werden auch Zeichen mit gesetztem 8 Bit an das Programm weitergeleitet. Sollen alle Zeichen weitergeleitet werden, müssen auch die Ende–Bedingungen aus– und die Kommandounterdrückumg eingeschaltet werden! Dann ist die Eingabe erst beendet, wenn genau die erwartete Anzahl Zeichen (über `TFU=...` einstellbar) eingelesen wurde. Es wird kein Software–Handshake (X_{on}/X_{off}) mehr durchgeführt. RTS/CTS werden weiterhin unterstützt. Wenn die „Gegenseite“ nicht auf RTS/CTS reagiert, muß dafür gesorgt werden, daß der Eingangspuffer (i. a. 31 oder 255 Zeichen) nicht überläuft, sonst gehen Zeichen verloren! Vorsicht beim Filesystem: Das Bit hat dort besondere Bedeutungen.

Beispiel: Die Schnittstelle ist auf `$3300` gesetzt (siehe `DD`–Befehl). Es sollen binäre Daten über diese Schnittstelle empfangen werden. Für den `AI`–Parameter ist dann `AI=$3F00` oder `MB=$87` (lies: keine besondere Ende–Bedingung / kein Echo / kein Kommandoprozessor / binärer Transfer) einzusetzen.

Timeout: Mit dem hinteren Byte des `AI`–Paramters kann ein Timeout für den Transfer über eine serielle oder parallele Schnittstelle gesetzt werden. Der Timeout gilt für ein ganzes Communication–Element, d. h. der Transfer muß in der angegebenen Zeit komplett durchgeführt sein. Die `ST`–Funktion gibt Aufschluß über das Auftreten eines Timeouts.

`AI=$xx80`
Das oberste Bit gibt an, daß ein Timeout gewünscht ist. Mit den restlichen 7 Bit (`0` bis `$7F`) kann die Länge des Timeouts eingestellt werden. Die Zeitbasis sind 512 msec. Dies ist also die kürzeste einstellbare Timeout Zeit. Die längste Zeit ist 128 $*$ 0,512 sec entspricht 65,5 sec.

Beispiel:
```
AI=$xx80        ! Timeout mit T = 512 msec
AI=$xxA5        ! Timeout mit T = 18,9 sec
```

Beispiel für die Anwendung von Datenstationen:

```
MODULE XYZ;
SYSTEM;
   Myfile:   /X0/dies/ist/nur/ein/platzhalter;
   Winch1:   /LD/3.2/USR/GE/test(TFU=400)<->;
   ! nach Moeglichkeit immer Mnemos anstelle von 'LD' verwenden.
   Terminal: /A1/Dialog (TFU=1,AI=$3C00)<->;
   Termreset:/A1/Reset <->;
   ! i. a. ist '/TYA' besser als '/A1'

PROBLEM;
   SPC (Winch1,Myfile) DATION INOUT ALPHIC;
   SPC Terminal DATION INOUT ALPHIC CONTROL(ALL);
   SPC Termreset DATION INOUT ALPHIC CONTROL(ALL);

TS1:TASK;
 DCL mist CHAR(20);
  OPEN Myfile BY IDF(textvariable oder konstante);
  /* Lies 20 Zeichen */;
  GET mist FROM Terminal BY A(20);
  ....  /* im Sondermodus */;
  /* normalmodus einschalten*/
  GET FROM Termreset BY SKIP;
END;
```

Im allgemeinen besteht keine Notwendigkeit mehr, Datenstationen über das oben benutzte `LD`-Konstrukt anzusprechen. Einzige Ausnahme sind Fälle, in denen man EPROM-Software für ein fremdes Zielsystem entwickelt, weil dann kein Lader involviert ist, der die Anschlüsse herstellt. Alternativ zum `LD`-Konstrukt hat man dann allerdings noch den Linker (siehe Seiten 163 ff.) zur Verfügung, der mit seiner `DEVICE`-Option die fehlende Laderfunktion ersetzen kann.

Mit dem `OPEN ... BY IDF (...)` wird der Inhalt der Textvariablen oder Textkonstante als aktueller Filename für die `DATION` (hier Myfile) eingesetzt. Durch diese Anweisung kann eine Dation für verschiedene Files genutzt werden, ohne die Notwendigkeit, das PEARL–Programm zu ändern.

Es ist zu beachten, daß bei Benutzung von `OPEN ... BY IDF (...)` im Systemteil ein genügend langer Dummy–Name für die Pfadliste als Platzhalter vorgesehen wird. Je nach **RTOS–UH**-Implementierung sind 24, 48, 64 oder mehr Zeichen bei der Pfadliste möglich.

Hinweis!

Eine Anweisung der Form `OPEN ... BY IDF('/H0/abcd')`, also mit einer auf der Root-Ebene beginnenden Pathlist, erlaubt in neueren Systemen neben der Filenamensänderung auch das Neubestimmen von `LDN` und `DRIVE` zur Laufzeit. Um dieses im Systemteil zu dokumentieren, wird empfohlen, bei derartigen Datenstationen im Systemteil als Gerät `/NIL` zu verwenden!

Bei der im Beispiel angegebene Dation (hier `/A1/Dialog` mit Usernamen „`Terminal`“) wird mit `TFU=1` auf Einzelzeichenübertragung geschaltet. Mit `AI=$3C` werden die normalerweise eingesetzten Mode–Bit's des Compilers (`$B8`) getoggelt, sodaß bei einer Eingabe (`GET ... FROM Terminal;`) ein CE mit einem Mode–Byte (`$86`) entsteht. Es wird auf die Zeichen Cr (`$0D`) , Lf (`$0A`) und Eot (`$04`) nicht mehr gesondert reagiert, sie werden in die Inputvariable eingetragen. Weiterhin ist die Funktion Supress–Command eingeschaltet, sodaß der Kommandointerpreter nicht mehr durch ein (`$01` – `$03` , `Ctrl A--C`) angestoßen wird, dieses gilt solange bis entweder ein `BREAK` oder ein weiteres `GET/PUT` auf die gleiche `DATION` (hier `A1.Reset`) mit anderem Usernamen (hier `Termreset`) und anderem `AI` ausgeführt wird.

5.5.3 Besonderheiten bei der formatierten Eingabe („GET“) im UH–PEARL

- Das System blockt die Eingabe in Sätze je nach der `TFU` des Systemteiles, im Standard jeweils 128 Zeichen. Ein vorzeitiges Satzende erzeugen im Standard: Cr, Lf und Eot(Mode–Byte = `$B8`, mit Wait). Dies kann durch das linke Byte von `AI` ggf. modifiziert werden (siehe dazu auch Abschnitt 8.3.2 „Die Modebytes“ auf Seite 562). Die `AI`–Option ist mit allergrößter Vorsicht zu benutzen. Mit jedem `GET` wird an der alten Stelle — im alten Satz! — weitergelesen. Wenn das unerwünscht ist, sollte das Format mit `SKIP` beginnen.

- Vorzeitige Eingabefeldbegrenzung bei `E`, `F`, `B`, `T`, `D`–Formaten.

  Steht in dem vereinbarten Eingabefeld (*width*) ein Komma, so beginnt hinter dem Komma das Eingabefeld für das nächste Eingabedatum.

  Bsp. `GET i,j,k,l FROM A1 BY SKIP,(4)F(20);`

  Als Eingabe würde die Zeile `-100,2,33,5,` korrekt akzep-

tiert. Man braucht also nicht 4 mal 20 Zeichen vorzusehen. Tut man es dennoch, so dürfen natürlich keine Kommata vorkommen. Auch das Zeichen CR beendet ein Eingabefeld.

- Vorzeitige Eingabefeldbegrenzung beim `A`–Format.

  Sie findet überhaupt nur statt, wenn im Format des `GET` das automatisch angepaßte `A`–Format steht (d. h. ohne Klammer mit Parametern). Beim Format `A(`*x*`)` wird in jedem Fall die vorgesehene Anzahl Zeichen gelesen.

  Das Eingabefeld endet beim freien `A`–Format vorzeitig, wenn das Zeichen CR eingelesen wird. Dieses wird in Space (`$20`) verwandelt, auch der Rest des einzulesenden String wird mit Spaces aufgefüllt.

  Bsp.
  ```
  DCL C CHAR(70);
  GET C FROM A1 BY SKIP,A;
  ```

  Wenn nun `ABCDEFGHI(CR)` eingegeben wird, so steht hinterher im String `C` der Text `ABCDEFGHI` mit 61 folgenden Spaces. Diese Option war nicht unumstritten, sie erleichtert aber die schnelle Kommandoanalyse für Dialoge mit PEARL–Programmen.

5.5.4 Besonderheiten bei der formatierten Ausgabe („PUT“) im UH–PEARL.

Es wird in Sätzen mit Länge der `TFU` transferiert, im `MODE`–Byte ist nur das Output–Bit (Mode–Byte = `$40`, ohne „Wait“) gesetzt. Durch `AI` kann ggf. ein Warten auf Ausgabeende erzwungen werden, in diesem Falle kann auch der Erfolg der Operation mittels der `ST(...)`–Funktion abgefragt werden. Bei `PUT` ohne `WAIT` wird der Wert der `ST(...)`–Funktion nicht beeinflußt.

5.5.5 Erweitertes OPEN/CLOSE–Statement

Das `OPEN ... BY IDF(...)`–Statement ist erweitert um die Optionen [`,(NEW` | `OLD` | `ANY`) [`,EXCLUSIVE`]].

Es bedeuten:

`NEW`: Die Datei darf beim Aufruf noch nicht existieren.
`OLD`: Die Datei muß beim Aufruf schon existieren.
`ANY`: Existenz der Datei wird nicht überprüft. Ist die Datei schon vor-

handen, so wird sie nur geöffnet; ist sie noch nicht vorhanden, so wird sie eingerichtet.

`EXCLUSIVE`: Die Datei wird für ausschließlichen Zugriff der aufrufenden Task geöffnet.

Das `CLOSE`–Statement ist erweitert um die Optionen
`[ BY CAN | BY PRM ]`.

`BY PRM`: normales Schliessen mit Dateierhalt.
`BY CAN`: Schließen und Löschen der Datei.

5.5.6 E/A–Formate

Formate dienen zur formatierten Ein–/Ausgabe der verschiedenen Datentypen. Datentyp und Konvertierungsformat müssen miteinander verträglich sein, sonst gibt es zur Laufzeit Fehlerreaktionen. Das RTOS/PEARL Funktionsprinzip ist dabei wie folgt zu beschreiben:

Der Ablauf des Geschehens wird von der mit dem Schlüsselwort **BY** angeschlossenen Formatliste gesteuert. Diese Seite ist also der treibende Part. Je nach dort gefundenem Format besorgt der Prozessor sich aus der Objektliste ein Objekt. Ist die Format-Liste abgearbeitet, verbleiben aber dabei noch unbearbeitete Objekte in der Objektliste, so beginnt die Konvertierung bei der letzten Blockwiederholklammer oder ganz von vorne, wenn es eine solche nicht gibt.

Die Argumente der Formate sind, der Syntax folgend, `FIXED`-Ausdrücke. Bei der Formatbeschreibung werden folgende Symbole verwendet:

s: Vorzeichen. Bei Ausgabe wird '+' durch ' ' ersetzt, bei der Eingabe kann '+' entfallen (Default–Wert).

Z: Ziffer. Z– –Z meint eine beliebige, durch Argument und Format bestimmte Anzahl von Ziffern.

[ ] kennzeichnet optionale Elemente.

5.5.7 Datenkonvertierungsformate

Bei der Eingabe von Daten wird ein Eingabefeld unabhängig von einer eventuell größeren Eingabefeldbreite durch ein Komma ',' abgebrochen, allerdings nicht bei einer Eingabe im `A`–Format. Bei letzterem ist das Komma ein legales Eingabezeichen.

F–Format

Das `F`–Format verarbeitet `FIXED`– und `FLOAT`–Zahlen. Syntax:

F (*Feldbreite* [, *Nachkommastellen* [, *Skalierung*]])

Ausgabe:

- Ausgabefeldaufbau: ' sZ– –Z[.Z– –Z]'
- Es werden *Feldbreite* Zeichen erzeugt
- Die Ausgabezeichenfolge steht rechtsbündig im Ausgabefeld.
- Wird *Nachkommastellen* nicht angegeben, so entfällt auch der Dezimalpunkt; bei `FLOAT` wird ge`ROUND`et.
- Der ausgegebene Wert entspricht der auszugebenden Zahl, multipliziert mit $10^{Skalierung}$.
- Unsignifikante führende Nullen werden durch Leerzeichen ersetzt.

Eingabe:

- Eingabefeldaufbau: ' [[s]Z– –Z[.[Z– –Z]]] '
- Ein leeres Eingabefeld ergibt 0.
- Die Eingabezeichenfolge kann beliebig im Eingabefeld stehen.
- Führende ' ' werden nicht ausgewertet.
- *Nachkommastellen* und *Skalierung* defaultieren zu 0.
- *Nachkommastellen* wird nicht ausgewertet, sondern von der tatsächlichen Eingabe bestimmt.
- Abgespeichert wird der Eingabewert mit um *Skalierung* verschobenem Dezimalpunkt.

E–Format

Das `E`–Format verarbeitet `FIXED`– und `FLOAT`–Zahlen. Syntax:

E (*Feldbreite* [, *Nachkommastellen* [, *Signifikanz*]])

Ausgabe:

- Ausgabefeldaufbau: ' sZ– –Z.Z– –ZEsZZ'
- Es werden *Feldbreite* Zeichen erzeugt.
- Die Ausgabezeichenfolge steht rechtsbündig im Ausgabefeld.
- Es folgen *Nachkommastellen* Zeichen auf den Dezimalpunkt.
- Die Mantisse umfaßt *Signifikanz* Zeichen.

Eingabe:

- Eingabefeldaufbau: ' [[s][Z– –Z.Z– –Z][E[s][Z[Z]] '
- Die Eingabezeichenfolge kann beliebig im Eingabefeld stehen.
- *Nachkommastellen* wird nicht ausgewertet.

A–Format

Das `A`–Format verarbeitet Zeichenketten.

A [(Länge)]

Ausgabe:

- wird *Länge* nicht angegeben, so wird die deklarierte Länge der auszugebenden Zeichenkette verwendet.
- Es werden *Länge* Zeichen erzeugt.
- Die auszugebende Zeichenkette beginnt linksbündig im Ausgabefeld.
- ggf. wird nach rechts mit ' ' aufgefüllt.

Eingabe:

- wird *Länge* nicht angegeben, so wird maximal die Länge der einzulesenden Zeichenkettenvariable verwendet. Die Eingabe von Carriagereturn (CR) beendet die Eingabe vorzeitig.
- wird *Länge* angegeben, so werden genau *Länge* Zeichen eingelesen.
- Die Zuweisung an die einzulesende Zeichenkette erfolgt linksbündig; ggf. wird nach rechts mit ' ' aufgefüllt.

B–Format

Das `B`–Format verarbeitet Bitketten.

(**B** | **B3** | **B4**) [(Länge)]

Die I/O ist in binärer (`B`), octaler (`B3`) oder hexadezimaler (`B4`) Form möglich.

Ausgabe:

- wird *Länge* nicht angegeben, so wird die genaue *Länge* der auszugebenden Bitkette verwendet.
- Es werden *Länge* Zeichen erzeugt.
- Die auszugebende Bitkette wird rechtsbündig im Ausgabefeld eingetragen.
- ggf. wird nach links mit `0` aufgefüllt.

Eingabe:

- wird *Länge* nicht angegeben, so wird die *Länge* der einzulesenden Bitkette verwendet.
- Die Zuweisung an die einzulesende Bitkette erfolgt rechtsbündig; ggf. wird mit `0` nach links aufgefüllt.
- *Länge* > *Länge* der einzulesenden Bitkette wird nicht unterstützt.

T–Format

Das `T`–Format bearbeitet Daten des Typs `CLOCK`.

T (*Feldbreite* [, *Dezimalstellen*])

Ausgabe:

- Die Ausgabe erfolgt rechtsbündig im Ausgabefeld.
- *Dezimalstellen* wird modulo 3 verwendet.

Eingabe:

- *Dezimalstellen* wird nicht verwendet.
- Die Eingabe kann beliebig im Eingabefeld stehen, Leerzeichen bei den Doppelpunkten sind erlaubt.

D–Format

Das `D`–Format bearbeitet Daten des Typs `DURATION`.

D (*Feldbreite* [, *Dezimalstellen*])

Ausgabe:

- Die Ausgabe erfolgt rechtbündig in das Ausgabefeld.
- *Dezimalstellen* wird modulo 3 ausgewertet.

Eingabe:

- *Dezimalstellen* wird nicht verwendet.
- Die Eingabe kann beliebig im Eingabefeld stehen.
- `HRS`, `MIN`, `SEC` müssen mit mindestens einem Leerzeichen von den umgebenden Zahlen getrennt werden.

LIST–Format

Das `LIST`–Format bearbeitet Daten jeden Typs.

LIST

Normalerweise tut dieses Format genau das, was man erwartet. (Zumindest meistens, und bei nicht zu hohen Ansprüchen) Bei der Eingabe von Gleitkommazahlen müssen diese allerdings einen Dezimalpunkt im Text besitzen, da sonst durch die Defaultbesetzung ein Punkt dazwischen gequetscht wird.

LIST-Ersatzmechanismus für die einzelnen Datentypen:

Datentyp:	Ersatzformat:
BIT(1 ... 32)	B4(adapted len)
CHAR(x)	A
CLOCK	T(13,3)
DUR	D(25,3)
FIXED(1 ... 15)	F(7,0,0)
FIXED(16 ... 31)	F(11,0,0)
FLOAT(1 ... 23)	E(13,6)
FLOAT(24 ... 55)	E(23,16)

Tabelle 5.4: Ersatzformate bei `LIST`

5.5.8 Steuerformate

Wiederholfaktor

Umklammerte Teile der Formatliste oder einzelne Formate können wiederholt werden, indem ein umklammerter Wiederholfaktor vorangestellt wird.
Syntax:

(*Ganzzahlkonstante*) *Einzelformat* oder
(*Ganzzahlkonstante*)(*formatliste*)

Beispiele:

```
PUT I,K TO XY BY (2)(X(5),F(7));
PUT I,K TO XY BY (2)F(12);
```

R–Format

Das R–(Remote)–Format dient zur Abarbeitung vorher vereinbarter Formate. Syntax:

R (*Format–Label–Identifier*)

In dem mit R angeschlossenen Format gilt wieder die normale Format-Syntax. Dort sind weitere R-Formate erlaubt. Allerdings kellert das Laufzeitsystem nur maximal 3 Rekursionsstufen. Bei Überschreitung erfolgt eine Laufzeitfehlermeldung.

X–Format

X [(Ganzzahl)]

Ausgabe:

- Es werden soviel Leerzeichen produziert wie die Ganzzahl angibt.

Eingabe:

- Es werden soviel Zeichen vom Eingabestrom überlesen und dabei ignoriert wie die Ganzzahl angibt.

SKIP–Format

SKIP [(Ganzzahl)]

Ausgabe:

- Es werden soviel Carriage–Returns (CR) ausgegeben wie die Ganzzahl angibt. Wenn für das Gerät die Eigenschaft „Add Linefeed to CR“ eingestellt ist, so erfolgt für jeden CR zusätzlich ein Linefeed.

Eingabe:

- Evtl. angebrochene, noch nicht zu Ende gelesene Eingabezeilen werden überlesen, d. h. der Datenrest der Zeilen wird ignoriert.

PAGE–Format

PAGE [(Ganzzahl)]

Ausgabe:

- Es werden soviel Seitenvorschubsteuerzeichen ($0C) ausgegeben wie die Ganzzahl angibt.

Eingabe:

- Die Wirkung entspricht dem `SKIP`-Format.

5.5.9 Report- und Positionierungsformate

Diese Formate erlauben die Statusabfrage bzw. Positionsabfrage oder Positionsveränderung. Zur Zeit ist nur das `RST`-Format implementiert.

RST–Format

RST(Fixedvariable)

Von dem Moment an, in dem der Prozessor dieses Format überläuft, werden alle zeitlich folgenden Fehler bei der E/A-Ausführung in der angegebenen Variablen abgelegt. Es handelt sich um denselben Wertevorrat (0 = kein Fehler etc.), wie er auch bei der `ST`-Funktion (siehe Seite 332) von einer Datenstation abgefragt werden kann. Man kann auf diese Weise jeden einzelnen Konvertierungsschritt einer formatierten E/A mit einzelnen Reportvariablen überwachen:

```
GET A,B FROM xyfile BY RST(stAconv),F(10),RST(stBconv),B4(4);
```

In diesem Beispiel wird das Konvertierungsprotokoll für das Einlesen von „`A`“ in der Variablen „*stAconv*“ und das Konvertierungsprotokoll für „`B`“ in der Variablen „*stBconv*“ deponiert. Man kann also ggf. entscheiden, welche Anfrage zu wiederholen ist.

Geradezu existenziell notwendig ist das `RST`-Format für das „`CONVERT`“-Statement“, in das ja bekanntlich keine Datenstation sondern statt dessen eine `CHAR`-Variable involviert ist. Beim `CONVERT` gibt es somit keinen Status einer Datenstation, den man mit `ST` abfragen könnte – hier hilft das `RST`–Format ganz besonders.

5.6 Umgang mit Feldern und Zeigern

5.6.1 Besonderheiten bei Feldzugriffen

Wie in der Norm vorgesehen können Arrays beliebig viele Dimensionen haben und es ist bei jeder Dimension ein beliebiger Startindex vorgebbar. Der Compiler rechnet grundsätzlich in einem 32 Bit Adreßraum, es gibt also dabei außer Speicherplatzproblemen praktisch keine Beschränkungen hinsichtlich der Feldgrößen. Das gleiche gilt natürlich auch für Felder, die Komponenten innerhalb einer Struktur sind.

In unserem PEARL90 dürfen die Feldindizes sowohl vom Typ `FIXED(15)` als auch `FIXED(31)` sein. Beliebige Mischungen sind möglich, auch darf man eine `FIXED(15)`-Größe einsetzen, wenn die zugehörige Feldobergrenze außerhalb des 16-Bit Zahlenbereiches liegt.

Der Zugriff auf Elemente mehrdimensionaler Felder erfordert im 32 Bit Adressraum eine oder mehrere 32x32 Bit Multiplikationen, für die auf den einfachen Prozessoren der 68000-Familie (68010, 68008, 68302 etc.) kein Maschinenbefehl vorhanden ist. Ein zeitlich ungünstiger Unterprogrammaufruf ist nötig und belastet diese Chips zusätzlich. Unser PEARL90-Compiler untersucht daher bei allen Feldzugriffen, ob nicht vielleicht eine 16x16 Bit Multiplikation genügt, deren 32 Bit Ergebnis dann den Feldzugriff ermöglicht. Dazu muß der Compiler das Feld als „kleines“ Feld erkennen können. Leider funktioniert diese Untersuchung nicht, wenn Felder mit dem Mechanismus des virtuellen Feldes `(,)` bzw. `(,,)` etc. in Prozeduren importiert oder über `(,) REF ..` angesprochen werden. Hier muß der Compiler sicherheitshalber immer mit 32 Bit-Arithmetik rechnen.

Die Kurzformel kommt genau dann zum Einsatz, wenn

1. die Feldgrenzen dem Compiler bekannt sind und
2. klein genug für 16-bit Arithmetik sind, sowie
3. alle aktuellen Indizes vom Typ `FIXED(15)` sind.

Beispiel:

```
DCL stidx INV FIXED INIT(100);
DCL enidx  INV FIXED INIT(-1);
 ....
DCL ar1(stidx:enidx,stidx:enidx) FIXED(31)
 ....
DCL (i,j) FIXED(15);
ar1(i,j)= ....
```

ergibt 16 Bit Zugriffsrechnung.

Tip:

Man kann etwas Geschwindigkeit gewinnen, wenn die definierten unteren Feldgrenzen jeweils zu Null gewählt werden, wie das bei der Programmiersprache C Standard ist. Es entfällt dann eine Addition pro Feldzugriff. Dies gilt nur, wenn dem Compiler der Feldaufbau bekannt ist.

5.6.2 Arbeiten mit Zeigervariablen

Zunächst eine **WARNUNG**: Wer mit `REF`–Variablen arbeitet, begibt sich in einen Raum, in dem unser Compiler keinen Schutz vor Unfug mehr geben kann! Eine Zeigervariable kann wer weiß wohin zeigen — und damit kann wer weiß was im System zerstört werden. Gleichwohl sind aber mit `REF`s Konstrukte programmierbar, die ihren (gefährlichen) Einsatz rechtfertigen.

Bis auf den Typ `REF CHAR()` sind Zeiger gemäß PEARL90 Standard implementiert. Sie dürfen also nicht nur auf einfache Objekte sondern auch auf Felder, Prozeduren, Tasks, Semaphore, Datenstationen usw. zeigen.

Musterbeispiel:

```
TYPE MENSCH STRUCT[ Name CHAR(10), Alter FIXED,
                    Nachbarlinks REF MENSCH,
                    Nachbarrechts REF MENSCH  ];
DCL (Maier,Mueller,Schulze) MENSCH;
DCL Arbeitszeiger REF MENSCH INIT (Maier);
```

5.6.2.1 Positionieren eines Zeigers

Auf der linken Seite einer Zuweisung steht dabei eine Zeigervariable. Die rechte Seite muß in der Lage sein, eine Objektadresse zu erzeugen. Es darf rechts also keine Konstante stehen. Auch Ausdrücke ergeben in der Regel keine Adresse, es sei denn, ein Feldzugriff oder Prozeduraufruf liefert ein Ergebnis vom Typ `REF` ab. Der Objekttyp muß exakt stimmen. Das Objekt `NIL` ist immer passend.

```
Arbeitszeiger         = Maier  ;
Maier.Nachbarlinks    = NIL    ;
Maier.Nachbarrechts   = Mueller;
...
Schulze.Nachbarrechts = NIL    ;
```

5.6.2.2 Vergleich von Zeigern

```
IF Arbeitszeiger IS Maier THEN ...
IF Maier.Nachbarlinks ISNT NIL THEN ..
```

Im UH–PEARL muß nur einer der beiden Partner der Operatoren `IS` und `ISNT` ein Zeiger sein. Der zweite Partner muß aber eine Adresse liefern können oder das Element `NIL` sein.

5.6.2.3 Dereferenzierung

Beim Dereferenzieren wird auf das „gezeigte“ Objekt zugegriffen. Auf der rechten Seite einer Zuweisung wird eine Zeigervariable automatisch dereferenziert, es sei denn, sie wird direkt der linken Seite, auf der wiederum eine Zeigervariable steht, zugewiesen. Es wird jedoch **nicht** dereferenziert, wenn eine Zeigervariable als Prozedurparameter eingesetzt wird und die Prozedur als Formalparameter einen Zeiger (per value oder per Ident) auf diesem Platz erwartet. Zeiger, denen ein Selektor folgt, werden in jedem Fall (auch auf der linken Seite) zunächst dereferenziert. Bei Strukturen kann dabei erneut ein Zeiger entstehen, wie das folgende Beispiel zeigt.

```
Schulze.Nachbarrechts = Arbeitszeiger;  ! Zeiger1=Zeiger2
Arbeitszeiger.Nachbarrechts.Alter = 25; ! Deref durch Selektor
Arbeitszeiger.Nachbarlinks.Nachbarlinks.Name ='Krueger';! ''
```

Das Dereferenzieren von Prozedurzeigern (s.u.) führt zur Ausführung der Prozedur, auf die der Zeiger zeigt und kann einen Wert ergeben, wenn es sich um eine Funktionsprozedur handelt. Wegen der Möglichkeit mehrstufiger Verwirrungen ist als Funktionsergebnis der Typ Prozedurzeiger jedoch nicht zugelassen.

5.6.2.4 Verschiebung

Mit Hilfe der Einbaufunktion `REFADD` kann ein Zeiger, der auf ein einfaches Datenobjekt zeigt, auf das im Speicher folgende gleichartige Objekt verschoben werden. Diese Verschiebung geht sehr schnell (Beim 68k mit addi constant to memory) ist aber zumindest vorläufig eine Spezialität des UH–PEARL. Der Verschiebewert kann vom Typ `FIXED(15)` oder `FIXED(31)` sein und bezeichnet die Anzahl Objekte, um die geschoben wird. Er wird intern automatisch mit der Länge des Objektes multipliziert. Eine Verschiebung von Feldzeigern (z.B. `REF (,)`), Prozedurzeigern o.ä. ist nicht möglich und auch nicht sinnvoll.

```
DCL Zeiger REF FIXED(31);
....
CALL REFADD(Zeiger,2); ! 2 Elemente ueberspringen
```

5.6.2.5 CONT - Operator

Der Operator `CONT` ist nur auf der linken Seite einer Zuweisung sinnvoll und darum auch nur dort implementiert. Ein einzelner Zeiger auf der rechten Seite einer Zuweisung wird, wie oben erläutert, nicht dereferenziert, wenn links ein Zeiger (ohne `CONT` davor!) steht.

```
CONT Arbeitszeiger = Mueller;
```

Hier wird das Objekt, auf das Arbeitszeiger zeigt, mit den Daten des Verbundes `Mueller` besetzt.

5.6.2.6 Übergabe von Zeigern an Prozeduren

Bei Prozeduraufrufen und bei den Operatoren können Zeiger auch per `IDENT` übergeben werden, d. h. der Prozedur wird nur die Adresse des Zeigers mitgeteilt. Diese `IDENT`-Übergabe von Zeigern ist nur dann sinnvoll, wenn die Prozedur den Zeiger selbst (und nicht die Inhalte, auf die er zeigt) verändern will. Prozeduren oder Operatoren können auch Zeiger als Ergebnistyp erzeugen.

Wenn eine Prozedur explizit einen Parameter vom Typ `REF IDENT` erwartet, so **muß** als Aktualparameter auch ein Zeiger angeboten werden. Bei Übergabe per value genügt dagegen ein adreßerzeugendes Objekt, also keine Konstante und kein Ausdruck.

Zeiger auf Felder enthalten einen kompletten Feldbeschreibungsblock, in dem implizit die innere Untergliederung des Feldes enthalten ist. Je nach Anzahl der Dimensionen n ist das eine größere Anzahl von Bytes, nämlich (16+n*8). Werden solche Zeiger `VOID`-Zeigern zugewiesen, so wird nur die Adresse des ersten Feldelementes transferiert.

5.6.2.7 Zeiger auf Prozeduren

Zeiger auf Prozeduren sind streng typgebunden. Der Compiler weist einem Prozedur-Zeiger nur dann die Einsprungadresse der Prozedur zu, wenn die zum Zeiger gehörende Signatur exakt mit derjenigen der anzuschließenden Prozedur übereinstimmt. Dereferenzierung eines Prozedurzeigers bedeutet, daß die Prozedur, auf die er zeigt, aufgerufen wird. Je nach Prozedurtyp ist dem Zeiger eine umklammerte Parameterliste nachzustellen. Ein einsam hingeschriebener Prozedurzeiger wird dereferenziert, die angeschlossene Prozedur damit aufgerufen.

```
DCL PP REF PROC(FIXED);   Prozedurzeiger
TUES:PROC(a FIXED);       Signatur wie oben
   ....
END;                      Ende von TUES

PP=TUES;                  Zuweisung an Zeiger
PP(5);                    Aufruf von TUES
```

Selbstverständlich überprüft der Compiler auch hier, ob die angebotenen Parameter geeignet sind. Dazu verwendet er die Informationen aus der Zeigerdefinition bzw. -spezifikation.

5.6.2.8 Wichtige Tips

Initialisieren Sie neu deklarierte Zeiger stets mit einer verwertbaren Objektadresse, eventuell einen Dummy dafür anlegen. Nie langlebige Zeiger auf kurzlebige Objekte zeigen lassen!

In PEARL wurden absichtlich Funktionen wie `MALLOC` ('C') oder `NEW` ('Pascal') nicht definiert, weil man unklare Zeitbedingungen bei der „Untergrundarbeit" befürchtete. Der Mangel läßt sich ja auch meist leicht umgehen. Legen Sie sich dazu ein genügend großes Datenfeld an und kodieren Sie eine Routine, die Zeiger zurückgibt. Beim Schaffen eines neuen Objektes holt diese Routine dieses aus dem Vorrat. Auf diese Weise lassen sich vernetzte Datenbanken etc. sogar auf Massenspeichern samt Zeigern retten, wenn dafür gesorgt wird, daß die Basisadresse des raumspendenden Feldes stets gleich ist (`DCL` auf Modulebene und mit Festadresse laden).

```
/* Pufferplatz f"ur MALLOC, NEW*/
TYPE MENSCH ....;
DCL ENORM(500000) MENSCH;
DCL GIBHER REF MENSCH, Nochfrei FIXED(31);
.......
GIBHER = ENORM(1);      ! Auf erstes Objekt
CALL REFADD(GIBHER,-1); ! Einen Platz zurueck
Nochfrei = 500000(31);  ! Alles noch frei
.......
MALLOC_MENSCH:PROC RETURNS(REF MENSCH);
  CALL REFADD(GIBHER,1); ! Zeiger weiter
  Nochfrei = Nochfrei - 1(31);
  IF Nochfrei GE 0(31) THEN RETURN(GIBHER);
  ELSE suche garbage ...  RETURN(..);
        oder PUT 'Leider kein Platz mehr...'
  FIN;
  TERMINATE; ! Abbruch
END; ! Ende von MALLOC_MENSCH
.......
Pointer=MALLOC_MENSCH(); ! Func call, not address assignm.
.......
```

Die leere Parameterliste in der letzten Zeile oben ist notwendig, damit der Compiler nicht irrtümlich vermutet, daß man die Adresse der Prozedur `MALLOC_MENSCH` der Zeigervariablen `Pointer` zuweisen möchte, was zu einem Typfehler führen würde. Durch die leere Parameterliste wird klar bestimmt, daß die Prozedur zunächst aufzurufen ist und der zur Laufzeit abgelieferte Zeiger bei der Zuweisung zu verwenden ist.

Ein pfiffigeres Programm würde natürlich im obigen Beispiel statt der „garbage collection“ eine Kette zurückgegebener Plätze anlegen und könnte so einzelne Lücken ganz schnell wieder neu besetzen.

5.7 Einbaufunktionen

5.7.1 Mathematische Funktionen

Das System stellt wichtige, mathematische Elementarfunktionen zur Verfügung. Diese werden syntaktisch wie PEARL–Funktionsprozeduren behandelt; sie dürfen im Systemteil nicht spezifiziert werden, da der Zugang zu den Routinen sonst verschüttet ist. Die Argumente müssen vom Typ `FLOAT(23)` oder `FLOAT(55)` sein; der Resultattyp entspricht dem Argumenttyp. Welche Funktionen es gibt, ist der unten folgenden Tabelle zu entnehmen.

Neben dem Compiler bekannten Funktionen gibt es noch zwei wichtige Funktionen außerhalb der PEARL-Definition: `RANF` und `DRANF`. Diese Funktionen dienen zur Erzeugung von Zufallszahlen; sie müssen im Systemteil spezifiziert werden. Die Beschreibung finden Sie auf Seite 344.

Im UH–PEARL sind neben den sogenannten „monadischen“ Operatoren der DIN–Beschreibung darüberhinaus die wichtigen mathematischen Funktionen als „Einbaufunktionen“ realisiert, d. h. sie werden vom Compiler eingebettet, ohne daß zur Laufzeit Zeitverluste zur Parameterübergabe, Speicherplatzbeschaffung etc. auftreten können. Sie sind infolge der automatischen Anpassung an einfache oder doppelte Floatgenauigkeit auch viel bequemer zu benutzen, als dies mit Bibliotheksroutinen möglich wäre. Normalerweise ist der Code der Einbaufunktionen als Scheibe („mathx.y“) im **RTOS–UH** vorhanden. Allerdings können auch bei Fehlen der Scheibe (z. B. in kleinen Laufsystemen) über den Lader oder Linker noch die nötigen Anschlüsse hergestellt werden. Aus diesem Grunde sind in der Tabelle rechts außen auch die internen Systemnamen der Einsprungadressen angegeben.

ACOS(expression)	Arcus Cosinus	#SACOS	#DACOS
ASIN(expression)	Arcus Sinus	#SASIN	#DASIN
ATAN(expression)	Arcus Tangens	#SATAN	#DATAN
COS(expression)	Cosinus	#SCOS	#DCOS
EXP(expression)	Exp. funktion Basis e	#SEXP	#DEXP
LD(expression)	Logarithmus dualis	#SLD	#DLD
LG(expression)	Logarithmus Basis 10	#SLG	#DLG
LN(expression)	Logarithmus naturalis	#SLN	#DLN
PI(expression)	Zahl π	#SPI	#DPI
SIN(expression)	Sinus	#SSIN	#DSIN
SQRT(expression)	Quadratwurzel	#SSQRT	#DSQRT
TAN(expression)	Tangens	#STAN	#DTAN

Tabelle 5.5: Mathematische Funktionen in PEARL

Alle in Tabelle 5.5 aufgeführten Funktionen sind dem Compiler bekannt, ihre Spezifikation in PEARL-Modulen erübrigt sich damit. Verwendet ein Nutzer gleichnamige Arrayvariablen oder Funktionsnamen, ist der Zugang zu den entspechenden Einbaufunktionen verschüttet, d. h. diese sind nicht mehr zugänglich.
Während der Compilation überprüft der Compiler den Mode (`FLOAT`!) des in jedem Fall zu umklammernden Argumentes. Allen Funktionen müssen `FLOAT(23)` oder `FLOAT(55)` Argumente bereitgestellt werden. Anhand der Genauigkeit des Übergabeparameters wird festgelegt, ob die mathematische Funktion mit einfacher (3. Spalte Tabelle 5.5) oder doppelter (4. Spalte Tabelle 5.5) Float-Genauigkeit ausgeführt werden soll. Bei der Funktion `PI` dient das Argument lediglich zur Genauigkeitsfestlegung und nicht zur Rechnung.

Beispiele:

```
X = SIN( a * b );
Z = LN( 25.0(55) );          Z muß vom Typ FLOAT(55) sein!
v = ASIN( SIN(y) );          Hauptwerte beachten,nicht v = y!
a = PI( 1.0(23) );           Ermittle π in einfacher Float-Genauigkeit.
```

Erscheint nach dem Laden des PEARL–Programmes eines oder mehrere der oben rechts angegebenen Internsymbole (das Numerozeichen „`#`“ dient dazu als Indikator) als undefiniert, so fehlt die „Math–Scheibe“, und Sie müssen den Einbaufunktionsfile beim Laden mit einbinden oder mit dem Linker vor dem Laden einbauen.
Die Einbaufunktionen können Laufzeitfehlermeldungen erzeugen, die durch Überschreitung von Grenzwerten verursacht sind. Dabei ist zu unterscheiden, ob die FPU 68881 bzw. der 68040 oder das Software–Float–Paket zum Einsatz kommt. Die Hardware produziert „`wrong operand`“ etc., während die Software meist eine namentliche Bezeichnung der Einbaufunktion liefert. In beiden Fällen ist mit Hilfe des Tasknamens bei eingeschalteter Line-tracer-option der Fehlerort jedoch i.a. gut zu ermitteln.

Im Compiler-Mode mit eingeschalteter FPU (Gleitkommahardware) sind zusätzliche Einbaufunktionen vorhanden, die ursprünglich aus dem Koprozessor MC 68881 stammen. Bei den Prozessoren 68040/60 und PowerPC werden diese Sonderfunktionen durch Software nachgebildet. Alle Funktionen liefern Ergebnisse des Typs `FLOAT`.

ATANH(expression)	arcus Tangeshyperbolicus
COSH(expression)	Cosinushyperbolicus
EXPM1(expression)	$(e^x) - 1$
INT(expression)	runden auf nächstliegende Integer
INTRZ(expression)	Fraktion abschneiden
LNP1(expression)	LN(x)+1
NEG(expression)	Negieren der Floatzahl
SINH(expression)	Sinushyperbolicus
TANH(expression)	Tangenshyperbolicus
TENTOX(expression)	10^x
TWOTOX(expression)	2^x

Tabelle 5.6: Mathematische Funktionen beim 68881-PEARL

Anmerkung: Die FPU ist auf den Mode „`round to nearest`“ voreingestellt.

Wichtige Warnung!

Bei den 68040/60-Systemen sollte bei Gebrauch der mathematischen Funktionen wenn irgendmöglich der Mode `P=68040` oder `P=68060` im Compiler angewählt werden. Sonst wird der kompatible F-Line Emulationsmode gemäß Motorola-Vorschlag gewählt, der bezogen auf RTOS–Standard aus prinzipiellen Gründen sehr schlechte Echtzeiteigenschaften (z. B. Berechnung des `SIN` in einem Supervisorprozeß) ergibt!

5.7.2 Die Funktion „ST“ zur Statusabfrage von Datenstationen

Durch das neue `RST`-Format in PEARL90 ist diese Funktion in vielen Fällen entbehrlich geworden, zumal mit Hilfe des `RST`-Formates ein E/A-Vorgang genauer protokolliert werden kann. Dennoch werden die Statusinformationen hier beschrieben, da sich das `RST`-Format intern auf die `ST`-Funktion abstützt. Man beachte jedoch, daß man aus Kompatibilitätsgründen, wann immer neue Programme geschrieben werden, das `RST`-Format bevorzugen sollte.

Aufruf: `I=ST(`*`Dationname`*`)` bzw.
`CASE ST(`*`Dationname`*`)+1` wie skalares `FIXED(15)`.

Beschreibung:

Der Compiler generiert einen Zugriff auf eine `FIXED(15)`–Zahl, die im `SYSTEM`–Block der angegebenen Datenstation abgelegt ist. Diese Zahl gibt an, ob und wenn welche Besonderheiten beim letzten `PUT` oder `GET` aufgetreten sind. Die Zahlenwerte finden Sie in der Tabelle 5.7. Der `ST`–Wert wird zu Beginn jedes `PUT` oder `GET` auf Null gesetzt. Der Wert „Null“ bezeichnet daher den ereignislosen Betriebsfall.

Mit Hilfe der `ST`–Abfrage ist es also möglich, z. B. syntaktisch falsche Eingaben programmgesteuert zu erkennen und geeignet darauf zu reagieren, und zwar unabhängig davon, ob die E/A-Fehlermeldungen der Station eingeschaltet sind oder nicht.

Die Funktion kann auch das Ergebnis der Einbaufunktionen `REWIND`, `SEEK`, `SAVEP`, `APPEND` und `SYNC` überprüfen. Der `ST`–Wert wird nicht auf Null gesetzt! Zuweisung `ST=0` über `PUT(GET) TO(FROM) xy BY LIST`.

ST-Wert	Bedeutung
0	Korrekte ereignislose Funktion.
1	END–OF–FILE. Daten konnten nicht gelesen werden.
2	Read–Error. Verschiedene Ursachen, z. B. File nicht exist.
3	Falsche Eingabe–Syntax, z. B. Buchstabe bei Zahleneingabe.
4	Exponent (Float) out of range.
5	FIXED–Input Zahlenüberlauf.
6	B–Format Eingabestring zu lang.
7	Timeout
8	Anzahl auszuführender SKIPS negativ.
9	Anzahl auszuführender X oder PAGE negativ.
10	Überlauf öffnender Klammern im Format.
11	Datentyp und Format passen nicht zueinander.
12	Datum in der Liste, aber kein Datenformat im Format.
13	Überzählige schließende Klammer im Format.
14	Dieses Gerät ist für diese Operation nicht geeignet.
15	Pathlist zu lang. Liste gekürzt.

Tabelle 5.7: Standardwerte der `ST`-Funktion bei der PEARL-E/A

Wenn im PEARL–Programm ein Objekt mit Namen ST definiert wurde, so ist der Zugang zur Einbaufunktion verschüttet. Die Funktion darf daher (wie alle Einbaufunktionen) nicht mit SPC spezifiziert werden!

Bei der Übertragung eines solchen PEARL–Programmes auf andere PEARL–Systeme kann man sich dann eine kompatible Funktion mit Hilfe dort (eventuell) vorhandener Signale (ON–Blöcke) selbst definieren.

Beispiel:
```
LBL: PUT 'Geben Sie bitte x ein' TO TERM BY SKIP,A;
GET x FROM TERM BY SKIP,F(10);
IF ST(TERM) NE 0 THEN GOTO LBL; FIN;
```

5.7.2.1 Sonstige ST–Werte

Wird das NE–Flag gesetzt, so kann mit der ST–Funktion der Status nach einer Operation abgefragt werden. Man erhält die Fehlernummern gemäß Tabelle 5.8.

ST	Bedeutung	/Fx	/ED	/VI	/Ax
30	Framing Error				x
31	sonst. Error serielle				x
32	Parity error				x
33	Data Adress Mark Error	x			
34	Track 000 not found	x			
35	Aborted Command Error	x	x	x	
36	Controller fault	x			
37	ID-Field not found	x			
38	CRC-Error in ID or Data	x			
39	Uncorrectable Data Error	x			
40	Bad Block found	x			
41	Drive not ready	x			
42	Device Write Protected	x			
43	Disk Changed	x			
44	Drive not Present	x			
45	No UHFM Disk	x			
46	Directory active	x			
47	Path List Error	x			
48	Directory in System	x			
49	Directory not found	x			
50	File not found	x	x	x	
51	Disk full Error	x	x		
52	File System not consistent	x			
53	Drive busy (RAW)	x			
54	File in System	x	x		
55	Root Directory or Disk full	x			
56	No DOS Disk	x			
57	Fatal Error in Block 0	x			
58	Exclusiv open(no access right)	x			
59+	reserved				

/VO ist wie /VI. Festplatten und Netzstationen siehe unter /Fx.

Tabelle 5.8: `ST`–Werte bei abgeschaltetem `NE`-Flag

5.7.3 Bitmapping Basis–Grafik

Es sind 3 elementare Basisroutinen integriert, die einfache Pixeloperationen unterstützen. Sie arbeiten auf dem für die ausführende Task aktuell vereinbarten Grafikschirm, bzw. Grafikfenster.

```
CALL SETPIX(xpos, ypos, colour);
CALL GETPIX(xpos, ypos, colour);
CALL LINE(x1pos, y1pos, x2pos, y2pos, colour);
```

In Wirklichkeit handelt es sich hier nicht um Prozeduraufrufe (die Syntax mit dem optionalen `CALL` ist nur wegen der PEARL–kompatibilität gewählt worden), sondern um extrem schnelle Ansprünge (Nur `JSR/RTS` als overhead!) der globalen Systemroutinen `#SSETP`/`#SGETP`/`#SLINE` wobei die `FIXED(15)`–Parameter in die Register `D0 ... D4` geladen werden. Falls in Ihrem System diese Routinen nicht vorhanden sind, können sie leicht als Maschinenprogramme selbst geschrieben und beim Laden angelinkt werden. Dabei hat man in den Routinen die Register `D0-D7` und `A0-A3` frei verfügbar. Im Standard ruft `LINE` intern die Funktion `SETPIX` auf, so daß ggf. nur `SETPIX` und `GETPIX` angepaßt werden müssen.

`SETPIX` und `GETPIX` sind zueinander komplementär, d. h. mit `GETPIX` wird der dritte Parameter (muß variabel sein!) auf den mit `SETPIX` oder sonstwie erzeugten Wert gebracht.

`LINE` führt eine Pixel–Interpolation durch, wobei Start– und Endpixel gesetzt und durch Zwischenpixel verbunden werden.

Das Löschen von Punkten und Linien erfolgt durch Schreiben mit Anwahl der Hintergrundfarbe.

Beispiel für einen Kasten:

```
CALL LINE(i,j,i+10,j,1);
CALL LINE(i,j,i,j+10,1);
CALL LINE(i+10,j,i+10,j+10,1);
CALL LINE(i,j+10,i+10,j+10,1);
```

Wenn alles in Ordnung ist, erscheint eine (wasserdichte) quadra- tische Kiste mit der linken oberen Ecke auf Position (`i,j`).

Wie bei allen Einbaufunktionen, so dürfen auch hier die Objekte `SETPIX`, `GETPIX` und `LINE` nirgends weder deklariert noch spezifiziert werden. Anderenfalls sind die Funktionen nicht mehr zugänglich.

Eine negative Farbe bedeutet, daß das Pixel durch Invertieren des Altzustandes erzeugt wird, nach erneutem Aufruf erscheint dort also wieder das alte Bild.

Bitte beachten:

Bedenken Sie, daß der Ablauf dieser Funktionen je nach Rechner evtl. mit sehr hoher Geschwindigkeit ausgeführt wird und unter Umständen nichts zu sehen ist, wenn das Objekt zu früh wieder gelöscht wird! Mögliche Abhilfe: Operation mit dem Vertikal–Prozeßinterrupt und `WHEN ...` synchronisieren, so daß mindestens ein Vollbild abgewartet wird.

5.7.4 Besondere E/A–Operationen

Auch hier handelt es sich nicht um normale Unterprogrammaufrufe, sondern um Sonderfunktionen des UH–Compilers, die aus Kompatibilitätsgründen zu anderen Systemen das `CALL`–Konstrukt benutzen. `ST` wird in jedem Fall geändert, bei fehlerfreier Funktion auf „Null“, sonst auf den entsprechenden Code gesetzt.

`CALL REWIND(`*`dationname`*`);` ..... Zurückspulen eines Files.
`CALL SYNC(`*`dationname`*`);` ..... Speicher–File–Synchronisation.
`CALL SEEK(`*`dationname,longfixed`*`);` ..... Aufsuchen des x.Byte im File.
`CALL SAVEP(`*`dationname,longfixed`*`);` ..... Position des Files retten.
`CALL APPEND(`*`dationname`*`);` ..... Anhang an den File vorbereiten.

`REWIND`: Der angegebene File wird zurückgespult. War er nicht vorhanden, so wird er neu eingerichtet. Ist das Gerät nicht rückspulbar, so erfolgt Meldung und `ST` = 14.

`SYNC`: Der angegebene File sowie alle anderen Files dieses Laufwerkes (Drive) wird mit dem Inhalt auf dem Medium so abgeglichen, daß z. B. danach alle Daten im Falle des Netzausfalles gerettet sind. Der Positionszeiger des Files, der Öffnungszustand etc. bleibt unverändert. Möglicherweise erfolgt überhaupt keine Aktion weil Memory und Medium (Floppy/Winch) bereits übereinstimmen. Im Fehlerfalle: Meldung und `ST` = 14.

`SEEK`: Der File wird auf die Position gebracht, die durch den Parameterwert vom Typ `FIXED(31)` bezeichnet wurde. Dies kann eventuell sogar hinter die aktuelle Schreibschlußmarke geschehen, wenn der File von früherer Schreibaktivität noch größer angelegt ist als er zur Zeit benutzt wird!

`SAVEP`: Die aktuelle Position (das wievielte Byte des Files das nächste zu schreibende sein wird) wird in der `FIXED(31)`–Variablen abgelegt und steht dann für einen späteren `SEEK` zur Verfügung.

Das Paar `SEEK/SAVEP` ermöglicht den Aufbau verketteter Daten auf den Massenspeichermedien mit schnellem Zugriff, weil immer nur der Block der jeweils betroffenen Stelle wirklich eingelesen wird. Meiden Sie möglichst die Nähe des File–Endes, da manche Filehandler nach dem Einlesen des letzten Bytes einen Auto–Close für den File durchführen und damit das Laufzeitsystem so auf die (nach `SEEK`) falsche Fährte locken! Am besten unterdrückt man die Autoclose-Operation durch `MB=$04`, näheres auf Seite 308.

`APPEND`: Verlängern des Files. Beim Aufruf dieser Funktion wird an das Fileende positioniert, danach kann mit `PUT`, `WRITE` der File erweiternd geschrieben werden. Zur Zeit im UH– und DOS–Filemanager implementiert. Im Fehlerfall: `ST` = 14.

Wie bei allen Einbaufunktionen, so dürfen auch hier die scheinbaren Prozedurnamen nicht deklariert oder spezifiziert werden, da der Übersetzer dann die Nutzerobjekte benutzen würde.

Beispiel:

```
CALL SAVEP( MyFile, I);
...
CALL SEEK( MyFile, I);
```

5.7.5 READ/WRITE

Prozeduren für den Binär–Transfer von Daten. Aus Gründen der Kompatibilität zum alten PEARL80-Compiler existiert hier neben dem PEARL90-Statement noch die alte prozedurale Schnittstelle. Da polymorphe Prozeduren (solche mit variabler Parameterliste) in PEARL90 nicht zugelassen sind, wird der alte Aufruf durch Einbaufunktionen nachgebildet. Die im alten PEARL80 nötige Anweisung

```
SPC (WRITE,READ) ENTRY GLOBAL;
```

muß unbedingt entfernt werden. Für den nun veralteten Aufruf gilt ansonsten die bisherige Syntax:

`CALL WRITE (`*`dationame,Variablenliste`*`);` ..... Schreiben
`CALL READ (`*`dationame,Variablenliste`*`);` ..... Lesen

Das `CALL` ist, wie generell in PEARL90, nur noch optional.

Bei neuen Programmen sind die Anweisungen zu ersetzen durch die PEARL90-Anweisungen

`WRITE` *Variablenliste* `TO` *dationexpression*;
`READ` *Variablenliste* `FROM` *dationexpression*;

Die in der *Variablenliste* aufgeführten Variablen werden in der Reihenfolge ihres Auftretens auf die angegebene Datenstation „binär“ transferiert, bzw. von ihr gelesen. Beim Abspeichern binärer Daten in /ED–Files ist zu beachten, daß man diese Daten zwar nicht mit dem `COPY`-Befehl korrekt transportieren kann, die interne PEARL-Welt davon jedoch unberührt bleibt und richtig arbeitet.

Variablenliste:

- ARRAY–Typen werden komplett übertragen (total E/A)
- es sind alle Variablen–Typen aus PEARL erlaubt
- es können verschiedene Typen gemischt werden
- die Länge der Liste ist nicht begrenzt

Beide Operationen arbeiten mit „`WAIT`“’, d. h. auch beim `WRITE` wird auf das Ende der Schreiboperation gewartet. Die Prozeduren setzen am Ende der I/O-Operation das Status–Byte der `DATION`, welches mit der Funktion `ST(...)` abgefragt werden kann (siehe Abschnitt 5.7.2).

Beispiel:

```
MODULE TEST;
SYSTEM;
FLOPPY: /F0/MIST <->;
...
PROBLEM;
SPC FLOPPY DATION INOUT ALPHIC;
...
task1: TASK;
   DCL FELD (100,20) FIXED;
   DCL ARR(3)  FLOAT;
   DCL R    FLOAT(55);
   ....
   WRITE  FELD,ARR,R TO FLOPPY;
   ....
   READ   FELD,ARR,R FROM FLOPPY;
    ....
 END;
 ...
 MODEND;
```

Es ist darauf zu achten, daß beim Lesen und Schreiben der Daten die Datentypenfolge der Parameterlisten übereinstimmt, da sonst falsche Werte zugewiesen werden.

Bei einem Zugriff auf die Floppy/Winch–Laufwerke (erkennbar am Device-Parameter $40 im 2. Byte, siehe Seite 203) wird intern eine Blockgröße von maximal 32766 Byte benutzt, ansonsten wird mit dem TFU-Wert aus dem Systemteil gearbeitet. Dieser ist ohne TFU-Angabe meist 128.

Hinweis: Die Prozeduren führen kein `REWIND` durch, daher muß vor der Benutzung die Position im File durch ein `REWIND` oder `SEEK` eingestellt werden. Die Integration der Positionierformate aus PEARL90 in eine `BY`-Liste ist noch in Arbeit.

! → Die Funktionen `READ/WRITE` besetzen das Mode–Byte des CE's (Ein–/Ausgabeelemente) anders vor:

- `READ`: WAIT,BINÄR,SUPRESS CMMD
- `WRITE`: WAIT,OUTPUT,BINÄR,SUPRESS CMMD

Daher werden die Standardendekennzeichen Cr, Lf, Eot nicht berücksichtigt.

Mit NE sind alle Fehlermeldungen abschaltbar. Bei verschiedenartigen Fehlern werden die vom I/O-Dämonen angebotenen Meldungen präsentiert. Ist dieser alt und zu einem solchen Report nicht in der Lage, so werden ersatzweise folgende Meldungen ausgegeben:

```
>>...: I/O-Error_during_READ/WRITE                    ST=2
>>...: Time_out_during_READ/WRITE                     ST=7
```

Bei Fehlern, die mit dem File-Ende zusammenhängen, sind folgende Meldungen möglich:

```
>>...: End_of_File(READ/WRITE)                        ST=1
>>...: Incomplete-transfer_READ/WRITE                 ST=1
```

Bei der letzten Meldung wurde festgestellt, daß die vom I/O-Dämonen tatsächlich transferierte Anzahl Bytes kleiner ist als der Auftrag verlangte. Tritt auf, wenn während des Lesens das File-Ende ereicht wurde.

5.7.6 READ/WRITE mit S-Format

Seit der P90-Compilerversion 15.9-M ist bei den READ/WRITE-Anweisungen eine Steuerung und Kontrolle der übertragenen Datenlänge möglich. Dazu wird das S-Format der P90-Norm genutzt. Dieses Format muss nach dem Schlüsselwort BY z.B. in der Form S(lvar1) stehen. Als Argument ist nur eine beschreibbare FIXED(31)-PEARL-Variable zugelassen. Durch Vorbesetzen der Variablen kann die zu schreibende/lesende Anzahl der Bytes zu kleineren Werten hin begrenzt werden. Nach der Operation wird in der Variablen die Anzahl tatsächlich transferierter Bytes abgelegt. Beispiele:

```
DCL x(4000) CHAR(1);                 Feldgröße 4000 Byte
lvar1=1000(31);                      Max 1000 Byte sollen gelesen werden
READ x FROM Infile BY S(lvar1);      Legt die tatsächliche Länge in lvar1 ab

lvar1=1; lvar2=2;                    Zwei Vorbesetzungen
WRITE a,b TO Outfile BY              Liste von Objekten
          S(lvar1), S(lvar2);        Von a max 1 Byte, von b max 2 Bytes
                                     lvar1, lvar2 enthalten tatsächliche
                                     Anzahl der geschriebenen Bytes.
```

Stehen mehrere S-Formate hinter `BY`, so werden diese in der Reihenfolge der Objekte in der Ein-/Ausgabeliste zugeordnet. Überzählige S-Formate werden ignoriert. In den einzelnen S-Formaten der Liste müssen logischerweise jeweils individuelle Steuervariablen verwendet werden, wenn eine sinnvolle Aktion entstehen soll. Sind weniger S-Formate als Listenobjekte vorhanden, so werden die nicht zuzuordnenden Objekte automatisch mit dem ungesteuerten READ/WRITE bearbeitet.

5.7.7 Die Einbaufunktion NOW

Bei Verwendung des Symboles `NOW` in Ausdrücken, E/A–Listen etc. wird ein Objekt vom Typ `CLOCK` erzeugt, welches als Inhalt die aktuelle Uhrzeit erhält. Der Anschluß ist sehr schnell.

Der Bezeichner `NOW` darf nicht anderweitig deklariert oder spezifiziert werden, da der Compiler sonst die Nutzervereinbarung benutzt.

Beispiel:

```
PUT NOW TO A1 BY SKIP,LIST;
Starttime = NOW;
...
Elapsedtime = NOW - Starttime;
```

5.7.8 Die Funktion DATE zum Einlesen des Datums

Diese Prozedur muß spezifiziert werden:

```
SPC DATE ENTRY RETURNS(CHAR(10)) GLOBAL;
```

Der Aufruf lautet:

```
DCL datum CHAR(10);
...
datum = DATE;
```

Die Funktion weist der Zeichenkette `datum` das aktuelle Datum in der Form *dd-mm-jjjj* zu. Es bedeuten

dd: Tag, 1...31
mm: Monat, 1...12
jjjj: Jahr, 1984...2070

Ist das Datum nicht gesetzt, so wird eine nur aus Bindestrichen bestehende Zeichenkette zurückgegeben.

5.7.9 Die Einbaufunktion REFADD

Um die in PEARL nicht explizit vorgesehene Manipulation von Zeigervariablen zu ermöglichen, wurde die Einbaufunktion `REFADD` geschaffen. Es handelt sich allerdings keinesfalls um einen wirklichen Prozeduraufruf, die `CALL`–Konstruktion wurde nur aus Kompatibilitätsgründen gewählt! Der Compiler generiert hier einen geschwindigkeitsoptimierten Maschinencode ohne JSR etc.

`CALL REFADD(`*`Pointer`*`,`*`Shift`*`);`

Pointer: Eine `REF`–Variable beliebigen Typs.
Shift: Ein `FIXED(15)` oder `FIXED(31)` Ausdruck, bzw. Konstante.

Auf die Adreßvariable *Pointer* wird der Wert *Shift* · *Objektgröße* aufaddiert. *Objektgröße* ist die Anzahl Bytes, aus der der Datentyp, auf den *Pointer* zeigt, besteht.

Wenn *Shift* eine Konstante ist, so werden die notwendigen Rechnungen schon zur Compilezeit erledigt und lediglich ein einziger `ADD...` bzw. `SUB...`–Befehl entsteht. Daher für *Shift* nicht leichtfertig Ausdrücke oder Variablen verwenden!

Natürlich darf auch `REFADD` nicht anderweitig deklariert oder irgendwie spezifiziert werden.

5.7.10 Die Funktion ASSIGN zum Ändern der Datenstation

Nicht für Neuentwicklung!

Diese Prozedur muß spezifiziert werden:

```
SPC ASSIGN ENTRY(DATION ALPHIC IDENT, CHAR(24))
GLOBAL;
```

Man beachte, daß gegenüber dem alten PEARL80 nun das Attribut `INOUT` fehlt. Dies ist erforderlich, weil anderenfalls der Compiler beim Aufruf der Prozedur für nur lesefähige oder nur schreibfähige Datenstationen einen Parameterfehler anzeigen würde.

Für den Aufruf muß die folgende Syntax eingehalten werden:

```
CALL ASSIGN (Dation,newDation );
```

In PEARL werden den Datenstationen logische Namen zugewiesen, dieses erfolgt im `SYSTEM`–Teil mit der Anweisung:

```
logName : PhysName.Filename <->;
```

Beispiel dazu: `Plotter: /PP/DRUCK ->;`

Mit der Anweisung `OPEN logName BY IDF(newname)` kann der Datenstation ein neuer Filename zugewiesen werden, es ist erst in neueren Systemen möglich, die Ein–/Ausgabe ohne Änderung des Quelltextes zu einer anderen Datenstation zu schicken.

Hierzu wurde seinerzeit die Funktion `ASSIGN` eingeführt, sie erlaubt die Zuweisung einer neuen Ein–/Ausgabe–Datenstation aus einem PEARL–Programm heraus. Sie ist nur noch aus Kompatibilitätsgründen im System enthalten. Neuere Programme sollten das erweiterte „`OPEN BY IDF`“ benutzen.

Dazu betrachten wir ein Beispiel, in dem die Ausgabe der Datenstation `Plotter` auf ein Floppy–File umgelenkt werden soll:

```
SYSTEM;
   Plotter: /PP/PLOTT ->;
PROBLEM;
   SPC ASSIGN ENTRY(DATION INOUT ALPHIC IDENT,
         CHAR(24)) GLOBAL;
   DCL newDation CHAR(24);
   /* entweder : */
   GET newDation FROM TERMINAL BY A,SKIP;
   /* oder : */
   newDation = 'F0';
   OPEN Plotter BY IDF('/' CAT 'newdation');
         ! bei neuen Systemen
   ! CALL ASSIGN (Plotter,newDation);
         ! -> fuer aeltere Systeme
   /* die Ausgabe kann auf die angegebene */
   /* Datenstation umgelenkt werden.      */
```

Falls bei *newDation* der String `TY` angegeben wird, so wird als Ein–/Ausgabe–Station das eigene Terminal (von dem das Programm gestartet wurde) benutzt. Außer `TY` ist auch `TYD` für die entsprechenden Duplex–Kanäle zugelassen. Für *newDation* sind sämtliche dem System bekannten Datenstationsnamen, sowie die Bezeichner `L`*x* (*x* steht für die `LDN` der `DATION`) und das erwähnte `TY`, `TYD` erlaubt.

5.7.11 Die Funktionen RANF und DRANF zur Erzeugung von Zufallszahlen

Die Funktionen müssen im Systemteil spezifiziert werden:

```
SPC RANF ENTRY ( FIXED(31) IDENT, FIXED(31) IDENT)
     RETURNS(FLOAT(23)) GLOBAL;

SPC DRANF ENTRY ( FIXED(31) IDENT, FIXED(31) IDENT)
    RETURNS(FLOAT(55)) GLOBAL;
```

Nach Deklaration der Argumente mit dem korrekten Typ lautet der Aufruf:

```
DCL ZufallN FLOAT;
DCL ( ZufallNM1, ZufallNM2 ) FIXED(31);
...
ZufallN = RANF( ZufallNM1, ZufallNM2);
```

Die Funktionen berechnen die neue „Pseudo“–Zufallszahl mit dem Namen `ZufallN` aus den beiden vorangegangenen Zufallszahlen gemäß folgender Formel:

$X_n = k_1 \cdot X_{n-1} - k_2 \cdot X_{n-2} (mod 2^{31} - 1)$

mit $k_1 = 217828199$ und $k_2 = 314159269$

Die erzeugten Zahlen liegen gleichverteilt im Intervall [0,1).

Hat einer der beiden Parameter (oder beide Parameter) beim Aufruf den Wert 0, so wird er (oder werden beide) vor Ausführung der Rechnung auf die aktuelle Uhrzeit (in Millisekunden) gesetzt. Durch gezielte Vorbesetzung der Parameter lassen sich reproduzierbare Folgen von „Pseudo“–Zufallszahlen erzeugen.

5.7.12 Die Funktion TASKST zum Feststellen eines Taskstatus

Diese Prozedur muß spezifiziert werden:

```
SPC TASKST ENTRY (CHAR(24)) RETURNS (BIT(32)) GLOBAL;
```

Der Aufruf lautet:

```
DCL Stat BIT(32);
...
Stat = TASKST (Taskname);
```

Die Funktion gibt den Wert `'FFFFFFFF'B4` zurück, wenn eine Task mit dem angegebenen Namen dem System unbekannt ist. Ist eine derartige Task im Speicher vorhanden, so hat `Stat.BIT(1)` folgende Bedeutung:

`'0'B` : Task ist in irgendwelche Aktivitäten verwickelt.
`'1'B` : Task ist weder eingeplant noch in irgendwelche Aktivitäten verwickelt, jedoch dem System bekannt.

Genauere Information über den Taskstatus sind in den Bits 17...32 enthalten. Diese Informationen haben allerdings nur sehr eingeschränkte Aussagekraft, da in dem Zeitraum zwischen Funktionsaufruf und Auswertung des Funktionswertes durchaus Statusänderungen möglich sind. Die Bedeutung der Bitstellen finden Sie in Tabelle 5.9 auf Seite 346. Die Bits sind dabei in PEARL–Notation, d. h. vom höchstwertigsten Bit (=Stat.BIT(1)) bis zum niederwertigsten Bit (=Stat.BIT(32)) numeriert.

Bit-Nr.	Aussage
2...16	keine Bedeutung
17	reserviert
18	Waiting for I/O. Task wartet auf Ein–/Ausgabe
19	Waiting for Activation. Sonderstatus: Task wartet auf Aktivierung.
20	Suspend. Task ist suspendiert.
21	Waiting for CE. Task wartet auf ein CE (Kontingent erschöpft oder Speicher voll).
22	Waiting for Workspace. Task wartet auf Zusteilung von Workspace.
23	Waiting for Sema. Task wartet auf Sema (vergeblicher Request).
24	reserviert
25	Planned for Activation. Für die Task liegt eine Einplanung irgendeiner Art vor.
26	reserviert
27	Timed Activate. Task ist zeitlich zur Aktivierung eingeplant.
28	Cyclic–Activate. Für die Task liegt eine Einplanung zur zyklischen Aktivierung vor.
29	WHEN ... ACTIVATE. Task ist mit `WHEN` zur Aktivierung eingeplant.
30	Timed Continue. Task ist zeitlich zur Fortführung eingeplant.
31	WHEN ... CONTINUE. Task ist mit `WHEN` zur Fortsetzung eingeplant.
32	Idle–Task. Task ist Idle–Task des Systems.

Tabelle 5.9: Taskstatus

5.7.13 Prozeduren zum Lesen und Ändern der Taskpriorität

Diese Prozeduren müssen spezifiziert werden:

```
SPC SETPRI ENTRY(FIXED) RETURNS(FIXED) GLOBAL;
SPC GETPRI ENTRY(FIXED) RETURNS(FIXED) GLOBAL;
```

Der Aufruf lautet:

```
DCL (OWNPRIO,PRIOSET,OLDPRIO,NEWPRIO) FIXED;
...
OWNPRIO = GETPRI(0);       ! Default-Prioritaet lesen
OWNPRIO = GETPRI(1);       ! Aktuelle Prioritaet lesen
OLDPRIO = SETPRI(NEWPRIO);
```

Diese Prozeduren dienen zur vorübergehenden Änderung der Priorität. Sie sind für Tasks gedacht, die mit Semaphore arbeiten, die auch von Tasks höherer Priorität genutzt werden. Durch das vorübergehende Erhöhen der eigenen Priorität ist gewährleistet, daß die normalerweise niederpriorisierte Task weiterarbeiten kann, während sie die Semaphore hält.

`GETPRI` liefert mit 0 als Übergabewert die Default-Priorität, mit 1 als Übergabewert die aktuelle Priorität der eigenen Task.

`SETPRI` ändert die Priorität der eigenen Task auf `NEWPRIO` und gibt den Wert vor der Änderung zurück. Hat `NEWPRIO` den Wert 0, erhält die eigene Task ihre Default-Priorität.

Beispiel: Die niederpriore Task `AAT` teilt sich mit einer hochprioren einen Variablensatz, der durch das Semaphor `S1` geschützt ist:

```
DCL HOCHPIO FIXED;
AAT2:TASK PRIO 10;
   HOCHPRIO=GETPRI(1);
   REQUEST S1;
   ...                      ! Variablensatz bearbeiten
   RELEASE S1;
END;
AAT: TASK PRIO 200;
   DCL OLDPRIO FIXED;

   OLDPRIO=SETPRI(HOCHPRIO);! Hohe Prioritaet vor Request
   REQUEST S1;
   ...                      ! Variablensatz bearbeiten
   RELEASE S1;
   OLDPRIO=SETPRI(OLDPRIO); ! Alte Prioritaet wiederherstellen
END;
```

5.7.14 Die Prozeduren TOIEES und TOIEED zur Floatzahl–Wandlung

Diese Prozeduren müssen spezifiziert werden:

```
SPC TOIEES ENTRY(FLOAT IDENT) GLOBAL;
SPC TOIEED ENTRY(FLOAT(55) IDENT) GLOBAL;
```

Der Aufruf lautet:

```
DCL float   FLOAT;
DCL float55 FLOAT(55);
...
CALL TOIEES(float);
CALL TOIEED(float55);
```

Die Funktionen wandeln Gleitkommazahlen aus der Darstellung des **RTOS–UH**–Software–Float–Formats in die IEEE–Darstellung um. Der Einsatz dieser Funktionen ist insbesondere zur Konvertierung binär abgespeicherter Daten bei Systemerweiterung mit 68020/6888x–Prozessoren sinnvoll.

5.7.15 Die Prozeduren TORTOS und TORTOD zur Floatzahl–Wandlung

Diese Prozeduren müssen spezifiziert werden:

```
SPC TORTOS ENTRY(FLOAT IDENT) GLOBAL;
SPC TORTOD ENTRY(FLOAT(55) IDENT) GLOBAL;
```

Der Aufruf lautet:

```
DCL float   FLOAT;
DCL float55 FLOAT(55);
...
CALL TORTOS(float);
CALL TORTOD(float55);
```

Die Funktionen wandeln Gleitkommazahlen aus der Darstellung des IEEE–Formates in das **RTOS–UH**–Software–Float–Format um. Der Einsatz dieser Funktionen ist insbesondere zur Konvertierung binär abgespeicherter Daten bei Systemerweiterung mit 68020/6888x–Prozessoren sinnvoll.

5.7.16 PEARL-Unterprogramme für Shellfunktionen

Viele Operationen, die man als Bediener mit Hilfe der Shell ausführen kann, möchte man in ähnlicher Weise auch aus einem PEARL-Programm heraus zur Verfügung haben. Dafür gibt es zwar die Station `/XC` (siehe Seite 415), jedoch ist in vielen Fällen das Resultat nicht gut an die PEARL-Umgebung angepaßt. Oft stört es auch, daß ein Dämon und nicht die PEARL-Task selbst die Operation ausführt. Einige Prozeduren dieses Abschnittes schaffen da Abhilfe.

CMD_EXW — Bedienbefehl ausführen

Nicht für Neuentwicklung! Ersatz durch EXEC.

Bedienkommandos können über die Datenstation `/XC` (siehe Seite 415) an das Betriebssystem abgesetzt werden. Dabei erhält das PEARL-Programm nur eingeschränkt eine Rückmeldung über Erfolg oder Mißerfolg der ausgeführten Aktion. Mit der Prozedur `CMD_EXW` können Kommandos an das Betriebssystem weitergegeben werden, deren erfolgreiche Ausführung über ein Bit zurückgemeldet wird. Es können alle Kommandos auf diese Art abgesetzt werden, auch „PEARL-SHELLMODULE“ oder transiente Kommandos, die erst geladen werden müssen. Der Kommandostring hat den gleichen Aufbau wie eine Eingabe an die Shell. Wurde das Kommando erfolgreich ausgeführt, so wird der Status `'0'B` zurückgemeldet, im Fehlerfall ist es eine `'1'B`.

Die Prozedur muß spezifiziert werden:

```
SPC CMD_EXW ENTRY ( CHAR(255)) RETURNS(BIT(1)) GLOBAL;
```

Beispiel:

```
DCL kommando CHAR(80);
DCL status BIT(1);

kommando = 'P LO NO';
IF CMD_EXW(kommando) THEN
    status = CMD_EXW('UNLOAD mist*');
    status = status OR CMD_EXW('LOAD');
    IF NOT status THEN
        PUT 'fertig' TO ....
    FIN;
ELSE
    status = CMD_EXW('ED');
FIN;
```

Diese Routine führt die Kommandos im „WAIT“-Mode aus, siehe dazu Seite 223 die Beschreibung des `WAIT`-Befehles.

Environmentvariable abfragen **ENVGET**

Mit dieser Routine ist es möglich, einzelne oder mehrere Variablen des User-Environment aufzulösen und den erhaltenen Textstring weiterzuverwenden. So können PEARL-Programme oder PEARL-Shellmodule so kodiert werden, daß sie sich automatisch an ihre Umgebung anpassen.

```
SPC ENVGET(CHAR(255)) RETURNS(CHAR(255) GLOBAL;
```

Die Routine erzeugt aus einem Eingabestring einen Ergebnisstring. Der Eingabestring enthält Text, in dem ein oder mehrere Environment-Variablen mit vorangestelltem $-Zeichen vorkommen. Die Routine ersetzt diese jeweils durch ihre textlichen Werte und gibt den Resultatstring zurück.

Hatte man etwa irgendwann vorher

`ENVSET TEXDIR=/HO/TEX;` eingegeben, so wird nun bei

```
OPEN dation BY IDF(ENVGET('$TEXDIR/OUT/HELPFILE'));
```

die Environment-Variable `$TEXDIR` in den entsprechenden textlichen Wert umgewandelt. Es wird daher der File `/HO/TEX/OUT/HELPFILE` geöffnet.

Die Routine versucht, alle mit $ beginnenden Variablen aufzulösen. Das muß zwangsläufig scheitern, wenn die Variable im Environment nicht definiert ist. In diesem Fall liefert sie einen Ergebnistring zurück, der mit dem $-Zeichen beginnt und die erste nicht auflösbare Variable angibt.

```
z=ENVGET('Trallala $michgibtsgarnicht');
```

wird im Ergebnis zu

`z='$michgibtsgarnicht';` resultieren.

Im Sinne einer sicheren Programmierung sollte man das Ergebnis stets daraufhin überprüfen, ob das erste Zeichen nicht ein $ ist. Einige wichtige Environment-Variablen werden in normalen Systemen vom Hochlaufskript eingerichtet und von der Shell aktualisiert, etwa

`$WORKDIR`	Aktuelles Working Directory
`$EXEDIR1`	1. Exekution Directory
`$EXEDIR2`	2. Exekution Directory
`$STDIN`	Standard Input
`$STDOUT`	Standard Output
`$STDERR`	Standard Error
`$EDITOR`	Standard Texteditor

Geben Sie von der Shell kurz den Befehl `ENVSET` ein. Sie sehen dann, wie alle Variablen Ihres Arbeitsplatzes besetzt sind.

EXEC — Bedienbefehl ausführen

Bedienkommandos können über die Datenstation `/XC` (siehe Seite 415) an das Betriebssystem abgesetzt werden. Dabei erhält das PEARL-Programm nur eingeschränkt eine Rückmeldung über Erfolg oder Mißerfolg der ausgeführten Aktion. Mit der Prozedur `EXEC` können Kommandos an das Betriebssystem weitergegeben werden, deren erfolgreiche Ausführung über ein Bit zurückgemeldet wird. Es können alle Kommandos auf diese Art abgesetzt werden, auch „PEARL-SHELLMODULE“ oder transiente Kommandos, die erst geladen werden müssen. Der Kommandostring hat den gleichen Aufbau wie eine Eingabe an die Shell. Wurde das Kommando erfolgreich ausgeführt, so wird der Status `'0'B` zurückgemeldet, im Fehlerfall ist es eine `'1'B`. Man beachte diese vom sonstigen Shellkonzept abweichende Polarität, die ihre Ursache in der Kompatibilität zur veralteten Routine `CMD_EXW` hat.

Die Kommandos werden nicht im „WAIT“-Mode ausgeführt. Eine Rückmeldung über die korrekte Operation eines Sohnprozesses ist nur möglich, wenn ein expliziter `WAIT`-Befehl vorangestellt wird. Siehe Seite 223, Beschreibung des `WAIT`-Befehles.

Die Prozedur muß wie folgt spezifiziert werden:

```
SPC EXEC ENTRY ( CHAR(255)) RETURNS(BIT(1)) GLOBAL;
```

Beispiel:

```
DCL kommando CHAR(80);
DCL status BIT(1);

kommando = 'WAIT; P LO NO';
IF NOT EXEC(kommando) THEN
    status = EXEC('UNLOAD mist*');
    status = status OR EXEC('WAIT;LOAD');
    IF NOT status THEN
        PUT 'fertig' TO ....
    FIN;
ELSE
    status = EXEC('WAIT;ED');
FIN;
```

Wird `EXEC` aus einem PEARL-Shellmodul heraus aufgerufen, so wirken die Befehle `CD` und `CXD` nur lokal und beeinflussen damit lediglich weitere Aufrufe von `EXEC`.

Usernummer feststellen **GET_USER**

Mit GET_USER wird die Usernummer des aufrufenden Users zurückgeliefert.

```
SPC GET_USER ENTRY RETURNS(FIXED) GLOBAL;
```

Beispiel:

```
DCL usernr FIXED;

usernr = GET_USER;
```

Wurde das Programm von der Schnittstelle A1 vom User 1 gestartet, so ergibt der Aufruf von GET_USER eine 0, da die User intern ab 0 gezählt werden. Erfolgte der Start des Programms z. B. von Schnittstelle A3 / User 3, so liefert GET_USER eine 2.

GET_TASKNAME — Eigenen Tasknamen feststellen

Die Funktion `GET_TASKNAME` liefert den Namen der eigenen Task zurück. Wenn es sich um eine normale PEARL-Task handelt, ist der Name bekannt, diese Funktion also nicht nötig. Etwas anders ist die Situation bei PEARL-Shellmodulen, hier wird eine Subtask generiert, bestehend aus dem Kommando und einer 2 stelligen Hexzahl.

```
SPC GET_TASKNAME ENTRY RETURNS(CHAR(24)) GLOBAL;
```

Beispiel: Es gebe ein Shellmodul, in dem das Kommando „MORE“ enthalten ist. Die Prozedur, die das Kommado aufruft, soll „mp“ heißen.

```
mp : PROC ...
DCL taskname CHAR(24);

taskname = GET_TASKNAME; ! gibt 'MORE/xx' mit xx
                         ! zwischen 00 und FF
...
```

5.7.17 PEARL-Unterprogramme für Textstrings

In PEARL ist nur eine recht rudimentäre Stringbehandlung vorgesehen. Mit den hier zur Verfügung gestellten Stringprozeduren ist ein leistungsfähiges Werkzeug für die Stringbearbeitung gegeben. Die hier vorgestellten PEARL-Unterprogramme für Textstrings gehören nicht zum **RTOS–UH**-Standard, sind aber frei verfügbar und weit verbreitet. Testen Sie im zweifelsfall, ob diese Unterprogramme nicht vielleicht doch im Lieferumfang Ihrer Implementierung enthalten sind.

Alle Prozeduren benötigen keinen eigenen „PWSP“ und arbeiten nur mit den Registern und dem vom Aufrufer bereitgestellten Task- bzw. Procedure-workspace. Dadurch sind sie sehr schnell und natürlich dennoch reentrant.

Die Funktionen dürfen bei Verwendung des PEARL90-Compilers **nicht** spezifiziert werden. Die folgenden Spezifikationen dienen daher nur zur Information und zur Verwendung beim alten „PEARL80“-Compiler.

```
SPC BEG  ENTRY (CHAR(255) IDENT) RETURNS (FIXED(15)) GLOBAL;

SPC INSTR ENTRY (CHAR(255) IDENT, FIXED(15) IDENT,
      FIXED(15) IDENT, CHAR(255) IDENT, FIXED(15) IDENT,
      FIXED(15) IDENT)
      RETURNS (FIXED(15)) GLOBAL;

SPC LEN  ENTRY (CHAR(255) IDENT) RETURNS (FIXED(15)) GLOBAL;

SPC MID  ENTRY (CHAR(255) IDENT,FIXED(15) IDENT,
      FIXED(15) IDENT)
      RETURNS (CHAR(255)) GLOBAL;

SPC KON  ENTRY(CHAR(255) IDENT, FIXED(15) IDENT,
      FIXED(15) IDENT, CHAR(255) IDENT,  FIXED(15) IDENT,
      FIXED(15) IDENT)
      RETURNS (CHAR(255)) GLOBAL;

SPC INSER ENTRY(CHAR(255) IDENT, FIXED(15) IDENT,
      FIXED(15) IDENT, CHAR(255) IDENT, FIXED(15) IDENT,
      FIXED(15) IDENT)
      RETURNS (CHAR(255)) GLOBAL;
```

Die übergebenen Strings müssen nicht unbedingt eine Länge von 255 Zeichen haben. Die Prozeduren erkennen an Hand der übergebenen Daten die tatsächli-

che Länge und verwenden diese als Maximalindex für einen Zugriff. Ein Überschreiben der einem String benachbarten Variablen ist mit diesen Funktionen nicht möglich. Die verwendeten Variablen seien wie folgt deklariert:

```
DCL (pos, anf1, end1, anf2, end2) FIXED;
DCL string1 CHAR(x1);
DCL string2 CHAR(x2);
DCL string3 CHAR(x3);
     ! mit x1, x2, x3 = [1 ... 255]
```

AFORM

Diese Funktion ist nur noch aus Kompabilitätsgründen im System enthalten und für neue Programme nicht mehr zu verwenden.

BEG

Aufruf: `pos=BEG(string1);`

`pos` wird die Position des ersten Zeichens innerhalb von `string1` zugewiesen, welches kein Blank ist. Falls `string1` ein Leerstring ist, hat `pos` nach Aufruf von `BEG` den Wert 0.

LEN

Aufruf: `pos=LEN(string1);`

`pos` wird die Position des letzten Zeichens innerhalb von `string1` zugewiesen, welches kein Blank ist. Falls `string1` ein Leerstring ist, hat `pos` nach Aufruf von `LEN` den Wert 0.

INSTR

Aufruf: `pos=INSTR(string1, anf1, end1, string2, anf2, end2);`

`INSTR` dient zur Suche von Teilstrings innerhalb eines Strings.
Der zu suchende String steht in `string2` ab `anf2` bis `end2`. Der zu untersuchende String ist der Teilstring aus `string1`, auf der Position `anf1` beginnend und der Position `end1` endend. `pos` hat nach Aufruf von `INSTR` die Position des ersten erfolgreichen Suchens. Ist der zu findende String nicht im zu untersuchenden enthalten, wird `pos` der Wert 0 zugewiesen.

Fehler	Ergebnis
`end2-anf2 > end1-anf1`	`pos = -1`, der Suchstring ist länger als der zu analysierende
`anf1 < 1`	`string1` wird ab 1 untersucht
`end1 > x1`	`string1` wird bis `x1` untersucht
`end1 < anf1`	`pos = -1`
`anf2 < 1`	`string2` wird ab 1 verwendet
`end2 > x2`	`string2` wird bis `x2` verwendet
`end2 < anf2`	`pos = -1`, es wurde ein ungültiger Suchstring vorgegeben

MID

Aufruf: `string2=MID(string1,anf1,end1);`

Die Prozedur `MID` kopiert einen Teilstring.
`anf1` und `end1` sind Anfangs- und Endposition der zu kopierenden Zeichenkette in `string1`, der angesprochene Teilstring wird `string2` zugewiesen.

Fehler	Ergebnis
`anf1 < 1`	`string1` wird ab Position 1 ausgeschnitten
`end1 > x1`	`string1` wird bis `x1` ausgeschnitten
`end1 < anf1`	`string2` wird der Leerstring zugwiesen
`end1-anf1+1 > x2`	Die ersten `x2` Zeichen ab `anf1` werden aus `string1` ausgeschnitten

KON

Aufruf: `string3=KON(string1,anf1,end1,string2,anf2,end2);`

Mit der Prozedur `KON` lassen sich zwei Teilstrings zu einem dritten zusammenfügen.
`string1` wird von `anf1` bis `end1` ausgeschnitten, der von `anf2` bis `end2` aus `string2` extrahierte String angefügt und das Ergebnis `string3` zugewiesen. Umfaßt der Zielstring `string3` mehr Zeichen als die beiden ausgewählten Teilstrings aus `string1` und `string2`, wird `string3` mit Leerzeichen aufgefüllt.

Fehler	Ergebnis
`anf1 < 1`	`string1` wird ab 1 ausgeschnitten
`end1 > x1`	`string1` wird bis `x1` ausgeschnitten
`end1 < anf1`	nur der Teilstring aus `string2` wird verwendet
`anf2 < 1`	`string2` wird ab 1 ausgeschnitten
`end2 > x2`	`string2` wird bis `x2` ausgeschnitten
`end2 < anf2`	es wird nur der in `string1` angesprochene Teilstring verwendet
`end1-anf1+1 > x3`	`string3` ist der gewählte Teilstring aus `string1`, wobei von `string1` ab `anf1` die ersten `x3` Zeichen verwendet werden
`end2-anf2+end1-anf1+2 > x3`	ist der Teilstring aus `string1` nicht zu lang, wird dieser korrekt ausgeschnitten und anschließend der Teilstring aus `string2` solange extrahiert, bis `string3` aufgefüllt ist

INSER

Aufruf: `string3=INSER(string1,anf1,end1,string2,anf2,end2);`

Diese Prozedur fügt einen Ausschnitt von `string2` in `string1` ein und weist den neu gebildeten String der Variable `string3` zu.
Der einzufügende Teilstring aus `string2` wird hierbei durch die Positionen des ersten (`anf2`) und letzten Zeichens (`end2`) beschrieben und nach dem `anf1`.ten Zeichen in `string1` eingesetzt. Nach der eingeschobenen Passage wird `string3` mit den ab `end1` noch verbleibenden Zeichen von `string1` fortgesetzt. Ist die Länge von `string3` größer als die der selektierten Teilstrings aus `string1` und `string2`, wird der Zielstring mit Leerzeichen aufgefüllt. Sollte `string3` weniger Zeichen als die zusammengefügten Teilstrings umfassen, bricht der Zielstring nach der der Position `x3` ab.
Wenn sich aus den Marken `anf1` und `end1` in `string1` ein überschneidener Bereich ergibt, werden die hierin enthaltenen Zeichen sowohl vor als auch nach dem Einschub von `string2` im Zielstring berücksichtigt. Hierzu ein Beispiel:

```
DCL STR1 CHAR(7) INIT('ABCDEFG');
DCL STR2 CHAR(7) INIT('0123456');
DCL STR3 CHAR(30);
...
STR3=INSER(STR1,3,6,STR2,3,4); /* STR3 ist 'ABC23FG'       */
STR3=INSER(SRT1,6,3,STR2,3,4); /* STR3 ist 'ABCDEF23CDEFG' */
```

Fehler	Ergebnis
`anf1 < 1`	`string3` beginnt mit der einzufügenden Passage aus `string2`
`anf1 > x1`	`string1` wird bis zum `x1`-ten Zeichen angesprochen
`end1 > x1`	`string1` wird bis `x1` berücksichtigt
`anf2 < 1`	`string2` ist an Position 1 markiert
`anf2 > end2`	ungültiger Teilstring angegeben, kein Einschub aus `string2` in `string3`
`end2 > x2`	der einzufügende String aus `string2` endet bei `x2`

CMPW

Aufruf: `test=CMPW(string1, anf1, end1, string2, anf2, end2);`

`CMPW` dient zum Vergleich zweier Strings. Der erste String steht in `string1` ab `anf1` bis `end1`. Der zweite String steht in `string2`, auf der Position `anf2`

beginnend und der Position `end2` endend.

In jedem String sind die Wildcards „`*`“ (ASCII-Wert $1E im Gegensatz zun dem sonst üblichen Wert $2A für den Stern) und „`?`“ (ASCII-Wert $1F im Gegensatz zu dem sonst üblichen Wert $3F für das Fragezeichen) zugelassen. Die Bourne-Shell übergibt automatisch die ASCII-Werte der Wildcards, falls diese nicht von Apostrophs („'“) oder Gänsefüßchen („““) umrahmt sind.

Die Wildcard „`?`“ steht für genau ein beliebiges Zeichen an der Stelle des Auftretens der Wildcard. Die Wildcard „`*`“ dagegen läßt sich wesentlich flexibler gebrauchen: Sie steht in dem jeweiligen String für eine beliebige Zeichenkette beliebiger Länge an der entsprechenden Stelle. Die Länge kann sogar 0 sein; in diesem wäre es egal, ob der „`*`“ an dieser Stelle stünde oder nicht.

Sind beide Strings identisch, so wird `test` der Wert 0 zugewiesen; wenn sie ungleich sind, der Wert 1.

Fehler	Ergebnis
`anf1 < 1`	`string1` wird ab 1 untersucht
`end1 > x1`	`string1` wird bis `x1` untersucht
`anf2 < 1`	`string2` wird ab 1 untersucht
`end2 > x2`	`string2` wird bis `x2` untersucht
`end1 < anf1`	`string1` wird wie ein Leerstring behandelt
`end2 < anf2`	`string2` wird wie ein Leerstring behandelt

5.7.18 PEARL-Unterprogramme für Datenstationen

Device-Mnemo erzeugen **DEVMNEMO**

DEVMNEMO liefert den zu LDN und DRIVE gehörigen Mnemo.

```
SPC DEVMNEMO ENTRY (/* LDN   */ FIXED, /* DRIVE */ FIXED)
                    RETURNS(CHAR(24)) GLOBAL;
```

LDN enthält eine logische Device-Nummer.

DRIVE enthält die Untergliederungsnummer des Devices.

Sind LDN oder DRIVE dem System nicht bekannt, so gibt DEVMNEMO den allgemeine Form /LD/$x.y$ zurück, wobei „x“ der Inhalt von LDN und „y“ der Inhalt von DRV ist. Sind LDN oder DRV > 255, wird ein Leerstring zurückgegeben.

Beispiele:

```
DCL DEVICE CHAR(24);

DEVICE = DEVMNEMO(1,0);      ! DEVICE= '/ED'
DEVICE = DEVMNEMO(100,129); ! DEVICE= '/LD/100.129' (falls
                             ! nicht doch ein Mnemo im System)
DEVICE = DEVMNEMO(10,0);     ! DEVICE= '/PP'
DEVICE = DEVMNEMO(257,1);    ! DEVICE= ''
```

GET_DEVICE — Device-Mnemo erzeugen

GET_DEVICE liefert zu einer LDN und DRIVE den passenden Mnemonic zurück.

```
SPC GET_DEVICE ENTRY (/* LDN   */ FIXED IDENT,
                      /* DRIVE */ FIXED
                     ) RETURNS(CHAR(24)) GLOBAL;
```

LDN muß eine logische Device-Nummer enthalten.

DRIVE muß die Untergliederungsnummer des Devices enthalten.

Sind LDN oder DRIVE dem System nicht bekannt, so wird ein Leerstring zurück geliefert und LDN ist auf -1 gesetzt.

Beispiele:

```
DCL (ldn,drv) FIXED;
DCL device CHAR(24);

ldn = 0;      ! Schnittstelle USER 1
drv = 2;      ! B-Betriebsart

device = GET_DEVICE(ldn,drv);  ! device= '/B1/'

ldn = 10;     ! Druckerschnittstelle
drv = 0;      ! keine Untergliederung

device = GET_DEVICE(ldn,drv);  ! device= '/PP/'


ldn = 55;     ! unbekannte LDN
drv = 2;      !

device = GET_DEVICE(ldn,drv);  ! device= ' ',ldn=-1
```

Work- und Exe-Directory lesen GET_WORK/EXEC-DIR

Nicht für Neuentwicklung. Ersatz durch ENVGET!

Die folgenden Funktionen gestatten das Einlesen des Working- oder Execution-Directorys.

```
SPC GET_WORKDIR ENTRY RETURNS(CHAR(128)) GLOBAL;
SPC GET_EXECDIR ENRTY RETURNS(CHAR(128)) GLOBAL;
```

Es wird der komplette String inklusive Devicebezeichner und Pathlist zurückgegeben.

Beispiel: Es sei `CD = /H0/user` eingestellt und `CXD = /H1/cmd`

```
DCL (wdir,edir) CHAR(128);

wdir = GET_WORKDIR;    /* wdir = '/H0/user' */
edir = GET_EXECDIR;    /* edir = '/H1/cmd'  */
```

Soll das Directory nicht als kompletter String eingelesen werden, sondern nach `LDN`, `DRIVE` und Pathlist getrennt, so stehen die folgenden Funktionen zur Verfügung:

```
SPC GET_WORKPATH ENTRY (/* LDN   */ FIXED IDENT,
                        /* DRIVE */ FIXED IDENT
                       ) RETURNS(CHAR(128)) GLOBAL;

SPC GET_EXECPATH ENTRY (/* LDN   */ FIXED IDENT,
                        /* DRIVE */ FIXED IDENT
                       ) RETURNS(CHAR(128)) GLOBAL;
```

Beispiel: Es sei `CD = /ED/user` eingestellt und `CXD = /F0/cmd`

```
DCL (wdir,edir) CHAR(128);
DCL (wldn,eldn,wdrv,edrv) FIXED;

wdir = GET_WORKPATH(wldn,wdrv);
/* wdir = 'user' , wldn = 1, wdrv = 0  */

edir = GET_EXECPATH(eldn,edrv);
/* edir = 'cmd'  , eldn = 3, edrv = 0  */
```

IDF_DATION — Parameter einer Datenstation

Mit der Prozedur IDF_DATION ist es möglich, alle Parameter einer Datenstation zu erhalten oder zu manipulieren. Es muß aber sichergestellt werden, daß nur konsistente Datensätze verwendet werden, da es sonst zu unkontrollierten Systemabstürzen kommen kann!

```
SPC IDF_DATION ENTRY (/* dation */ DATION ALPHIC IDENT,
                      /* name   */ CHAR(128) IDENT,
                      /* AI     */ BIT(16) IDENT,
                      /* TFU    */ FIXED(15) IDENT,
                      /* LDN    */ FIXED(15) IDENT,
                      /* DRIVE  */ FIXED(15) IDENT
                     ) GLOBAL;
```

dation	bezeichnet die Datenstation, auf deren Parametersatz zugegriffen werden soll. Gegenüber PEARL80 fehlt hier das Attribut INOUT, da der PEARL90-Compiler hier bei reinen Lese- bzw. Schreibstationen sonst einen Parameterfehler anzeigt.
name	gibt Pathlist+Dateinamen ohne Device an (kann auch mit OPEN BY IDF ... geändert werden). Beim Lesen wird bis „maxpath“ mit Blanks aufgefüllt.
AI	enthält die Bits des AI-Parameters aus dem SYSTEM-Teil.
TFU	gibt die Transferlänge an.
LDN	liefert die logische-Device-Nummer (Nummer der Warteschlange). Wird für LDN eine negative Zahl eingesetzt, werden die Parameter der Dation ausgelesen und können zu einem späteren Zeitpunkt zurück geschrieben werden.
DRIVE	enthält die Untereinheit der Datenstation.

Die Prozedur IDF_DATION setzt den ST-Parameter, so daß eine Überprüfung der Operation erfolgen kann:

ST = 0	kein Fehler aufgetreten
ST = 2	Es wurde keine gültige Dation gefunden.
ST = 15	Die Pathlist incl. Dateinamen ist zu lang.

Fortsetzung **IDF_DATION**

Beispiel:

```
      DCL name CHAR(128);
      DCL AI BIT(16);
      DCL (tfu_alt,tfu,ldn,drv) FIXED;

      ldn = -1;        ! Datenstationsparameter lesen
      CALL IDF_DATION(term,name,AI,tfu,ldn,drv);
!     Die Parameter der Dation term sind ausgelesen
      tfu_alt = tfu;
      tfu = 1;         ! nur noch Einzelzeichen lesen
      CALL IDF_DATION(term,name,AI,tfu,ldn,drv);
!     ab jetzt werden Einzelzeichen ueber term gelesen
      ...
      CALL IDF_DATION(term,name,AI,tfu_alt,ldn,drv);
!     alte Transferlaenge wiederherstellen
```

Die Routine ist besonders hilfreich, wenn man sich mit `DCL` eine Station zur Laufzeit in einer Task oder Prozedur erzeugt hat und nun spezielle Parameter einstellen will oder von einer anderen Station übernehmen will.

SET_DATION — Datenstation neu setzen

Nicht für Neuentwicklung!

SET_DATION analysiert einen Textstring und macht daraus einen Parametersatz für eine Datenstation. Damit kann nicht nur der Dateiname, sondern auch die LDN und das DRIVE einer Dation geändert werden. Ist in dem Textstring kein Gerätename vorhanden, wird das eingestellte Working Directory berücksichtigt.

Auch diese Funktion ist in der PEARL90-Welt inzwischen überholt, da man sich mit der Funktion ENVGET (siehe Seite 351) das Working Directory holen und mit OPEN BY IDF einarbeiten kann.

```
SPC SET_DATION ENTRY(/* dation */ DATION INOUT ALPHIC,
                     /* string */ CHAR(128)
                    ) GLOBAL;
```

Die Prozedur SET_DATION setzt den ST-Parameter, so daß eine Überprüfung der Operation erfolgen kann:

ST = 0	kein Fehler aufgetreten
ST = 2	Es wurde keine gültige Dation gefunden.
ST = 15	Die Pathlist incl. Dateinamen ist zu lang.

Beispiel: Es sei ein Working-Directory = /H0/user eingestellt.

```
DCL string CHAR(128);

string = 'lib/myfile';

CALL SET_DATION(dat,string);

OPEN dat;     /* jetzt kann auf /H0/user/lib/myfile */
              /* zugegriffen werden.                */
```

5.8 Aufruf von C-kodierten Unterprogrammen

In der PEARL–Welt werden mit dem GNU-C Compiler übersetzte C-Unterprogramme nicht anders aufgerufen als PEARL-Unterprogramme, die mit dem PEARL–Compiler übersetzt wurden – es gibt allerdings massive Einschränkungen bei den transferierbaren Parametern. Die aufzurufende Prozedur wird außerdem anders spezifiziert, wobei zwangsläufig das Attribut „`GLOBAL`“ erforderlich ist.

Darüberhinaus können auch andere globale Objekte zwischen dem C-Modul und dem PEARL-Modul ausgetauscht werden, z. B. `FIXED`–Variablen. Dazu sind Werkzeuge von anderen Anbietern zu verwenden, die die Objektcode-Dateien aus der C-Welt in S-Rekords der **RTOS–UH**- Welt umsetzen, z. B. etwa OBJ2SR von *esd*.

Ein Aufruf von PEARL-Unterprogrammen aus der C-Welt ist dagegen nicht möglich, weil die C-Compiler nicht das besondere Umfeld (Index-Test, Parametertest zur Laufzeit, kein Stack etc.) für PEARL-Unterprogramme bereitstellen.

Bei der Spezifikation wird statt der Schlüsselworte `PROC`, `PROCEDURE` oder `ENTRY` ein um das Anhängsel `_C` erweitertes Schlüsselwort, also `PROC_C`, `PROCEDURE_C` oder `ENTRY_C` verwendet. Der PEARL-Compiler generiert nun das für diese C-Programme erforderliche (wegen des Stacks wie bei allen C-Programmen sehr unsichere!) Umfeld zum Aufruf der C-Prozedur bei jedem Aufruf.

Beispiel:

```
MODULE ... ; SYSTEM; ... ;
PROBLEM;
 SPC Hilf ENTRY_C(FIXED(31) IDENT) RETURNS(FLOAT) GLOBAL;
 ...
 task1:TASK; ...
 ...
 x=Hilf(i);
 ...
 END;
MODEND;
```

Das C-programm `Hilf` wird übersetzt, in S-Records umgesetzt und kann danach normal mit dem Linker eingebunden werden oder durch den Lader hinzugenommen werden. Ein Ladebefehl könnte hier etwa

```
LOAD /ED/SR+/ED/Hilfe
```

lauten, wenn im File `/ED/Hilfe` das in S-Records übersetzte C-programm abgelegt wurde.

Warnung!

Vergewissern Sie sich, daß das C-Programm mit genügend Stack-Speicher ausgestattet wird! Es ist – wie leider in der C-Welt üblich – keine Instanz da, die eine Stacküberschreitung verhindern kann. Man kann lediglich mit Hilfe eines entsprechend überdimensionierten Prozedurarbeitsspeichers der ausführenden Task (/*+R-Kommentar, siehe Seite 303) die Wahrscheinlichkeit für einen Stacküberlauf beliebig verringern. Eine weitere Unsicherheit: es ist keine Überprüfung der Parameter zur Laufzeit möglich! Stimmt die Spezifikation nicht mit der Definition in C überein, so sind schwer erkennbare Fehlfunktionen denkbar.

Die Parameterübergabe sollte sich auf die Typen `FLOAT(23)`,`FLOAT(55)`, `FIXED(31)`, deren Zeiger (`IDENT`) oder Zeiger auf Strukturen beschränken. Bei Strukturen ist auf ggf. unterschiedliche *Padding*-Modes zu achten. *Padding* wird z.B. benutzt, wenn nach einer Komponente vom Typ `CHAR(1)` eine Komponente vom Typ `FLOAT(55)` folgt. Je nach System werden dabei 1 oder 3 Bytes als Füllung eingefügt. Eine gewisse Anpassung des PEARL-Compilers ist möglich, siehe Seite 294.

5.9 Aufruf von Assembler–Unterprogrammen

In der PEARL–Welt werden Assembler-Unterprogramme nicht anders behandelt als solche, die mit dem PEARL–Compiler in anderen Modulen übersetzt wurden. Das Objekt ist korrekt zu spezifizieren, wobei der Zusatz „`GLOBAL`“ erforderlich ist.

Darüberhinaus können auch globale Objekte im *Nicht–PEARL–Modul*, z. B. `FIXED`–Variablen im Assemblerprogrammodul, durch das Linking des Laders aus der PEARL–Welt adressiert werden.

Versuchen Sie stets, die Einbeziehung von Assemblerunterprogrammen auf Ausnahmefälle zu beschränken. In jedem Fall sollten Sie nur transferassemblierbaren Maschinenkode benutzen, damit eine Hardwareabhängigkeit weitgehend vermieden wird. Assemblerprogramme können in unserem System ja bekanntlich ohne Emulation sowohl auf dem 68k als auch auf dem PowerPC laufen.

Hinsichtlich der korrekten Codierung des Assemblerprogrammes wird auf das Kapitel über den Assembler und Transferassembler verwiesen.

Beispiel:

```
MODULE ... ; SYSTEM; ... ;
PROBLEM;
 SPC Hilf ENTRY(FIXED IDENT) RETURNS(FLOAT) GLOBAL;
 ...
 task1:TASK; ...
 ...
 x=Hilf(i);
 ...
 END;
MODEND;
```

Das Assemblerprogramm `Hilf` kann mit dem Linker eingebunden werden oder durch den Lader hinzugenommen werden. Ein Ladebefehl könnte hier etwa

```
LOAD /ED/SR+/ED/Hilfe
```

lauten, wenn im File `/ED/Hilfe` das übersetzte Assemblerprogramm abgelegt wurde.

Warnung!

Vergewissern Sie sich, daß das Assemblerprogramm wiedereintrittsfest ist! Im Gegensatz zu den Unterprogrammen, die der Compiler generiert, sind Assemblerprogramme keineswegs automatisch für eine Multitaskingumgebung geeignet.

Wenn allerdings gesichert ist, daß nur eine Task das Unterprogramm benutzt, brauchen Sie sich theoretisch nicht um dessen Wiedereintrittsfestigkeit zu kümmern. Dennoch sei davon abgeraten: es zeigt sich immer wieder, daß oft uralte Unterprogramme wiederverwendet werden und man dabei leicht deren Wiedereintrittsrestriktionen übersieht. Die in diesem Handbuch beschriebenen systemeigenen Assemblerunterprogramme sind wiedereintrittsfest – es sei denn, daß ausdrücklich etwas anderes angegeben wird.

5.10 Ausnahmebehandlung und Signale

5.10.1 Vorgänge im Systemkern

Zunächst soll die betriebssystemseitige Bearbeitung von Ausnahmesituationen betrachtet werden. Dabei wird von einem normalen System mit dem Standard Error-Dämon `#ERRDM` ausgegangen. Es sind 3 unterschiedliche Ausnahmesituationen zu unterscheiden:

1 Hardwareinduzierte Ausnahmen: Die Rechnerhardware wurde mit einer nicht korrekt lösbaren Aufgabe betraut. Klassisches Beispiel: Es soll von einer nicht existenten Speicherzelle gelesen werden oder der gelesene Maschinenbefehl ist illegal kodiert. Aus der Sicht der gerade laufenden Software kommt das Ereignis unvorbereitet, wenngleich es an einen Maschinenbefehl gekoppelt ist - im Gegensatz zu von der Außenwelt getriebenen Interrupts. Der Prozessor wird bei dieser Ausnahmesituation wie bei einem Interrupt der höchsten Priorität nun zu einem speziellen Stück Code (im Supervisormode) geführt, welches den individuellen Errorcode erstellt und darin vorgibt, ob der verursachende Prozess anzuhalten ist. Der Verursacher kann hier sowohl eine Task als auch ein Supervisorprozess (kernel mode) sein. Supervisorprozesse dürfen aber keinesfalls angehalten werden sondern müssen zu einem funktionserhaltenden Ende geführt werden.

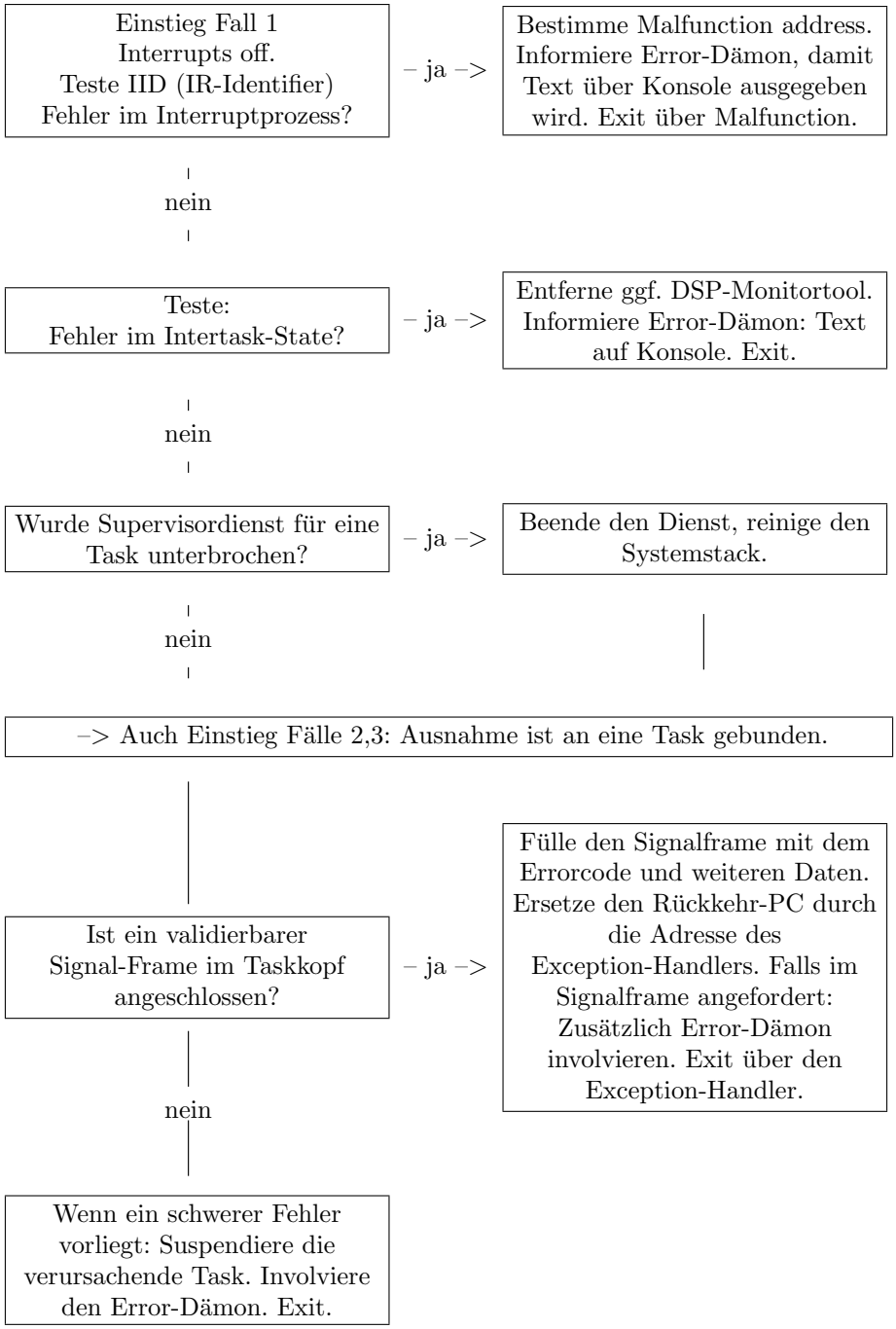
2 Softwareinduzierte Ausnahmen: Der Rechner funktioniert hardwareseitig korrekt, aber es ist eine (typischerweise datenabhängige) Fehlersituation aufgetreten. Klassisches Beispiel: die Software soll die Wurzel aus einer negativen Zahl bestimmen. Die Ausnahmesituation wird von der Software durch einen Betriebssystemaufruf (`ERROR`-Trap) eingeleitet. Durch die Kodierung des dem System mitgegebenen Error-Codes wird ein informeller Text erstellt. Auch hier wird dem System damit mitgeteilt, ob der laufende Prozess angehalten werden muss: Während man die Wurzel aus einer negativen Zahl ersatzweise mit Null beantworten kann (und die Task weiterlaufen kann), ist es nicht möglich, einer Prozedur übergebene (falsche) Zeiger (wrong parameterlist) zu reparieren. Im letzteren Fall muss die Task angehalten werden und sollte nicht entblockiert sondern beendet werden – eventuell durch einen Bedienerprozess oder manuell. Der Fehler kann stets einer Task angelastet werden.

3 Mischformen: Es gibt in fast allen Prozessoren Maschinenbefehle, die abhängig vom inspizierten Datum eine Ausnahmesituation auslösen oder auch nicht. Klassisches Beispiel: Der Indextester des PEARL-Compilers überprüft im Test-Mode mithilfe des `CHECK`-Traps ob der errechnete lineare Feldindex eine Zelle innerhalb des Feldes adressiert. Liegt das Element außerhalb, feuert der Maschinenbefehl eine Ausnahme. (Nebenbei: Beim PEARL-Indextester repariert das Ausnahmebehandlungsprogramm den Fehler und addressiert das erste Element des Feldes - Fortsetzung möglich!). Der Fehler kann typischerweise immer einer Task angelastet werden, wird daher wie eine softwaregetriggerte Ausnahme behandelt.

Im Nukleus ist für alle drei Ausnahmesituationen eine Behandlungsroutine integriert. Diese kann man – für alle Fälle, in denen der Fehler einer Task zugeordnet werden kann – so parametrieren, dass die verursachende Task gezwungen wird, selbst auf den Fehler zu reagieren. Dabei kann das zentrale Fehlermeldesystem (`#ERRDM` = Error-Dämon) ausgeschaltet werden. So arbeitet z.B. die Shell mit einem eigenen Handler und verwendet das Fehlermeldesystem nicht.

Das Involvieren des Error-Dämonen erfolgt über einen Ringpuffer, der in der Shell des verantwortlichen Users angelegt ist. Bei den Ausnahmesituationen, die nicht einer Task zugeordnet werden können, erfolgt die Fehlermeldung auf der Konsole – der Anschlusss eines eigenen Exception-Handlers ist für diese Situation nicht möglich und auch nicht sinnvoll. Aber: alle Interruptprozesse bringen zwangsweise (über den Malfunction-Exit) einen eigenen Handler mit. Dies ist eine markante Eigenschaft von **RTOS–UH**, die die Überlebenswahrscheinlichkeit erhöht.

Im folgenden Flussbild wird der Ablauf mit einer Ausnahme vom Typ 1 (Hardwarefehler) begonnen. Die Fälle 1,2 und 3 treffen sich an einem Sammelpunkt – immer dann, wenn eine Task als Verursacher feststeht.


Einstieg Fall 1
Interrupts off.
Teste IID (IR-Identifier)
Fehler im Interruptprozess?
– ja –>
Bestimme Malfunction address.
Informiere Error-Dämon, damit
Text über Konsole ausgegeben
wird. Exit über Malfunction.
nein
Teste:
Fehler im Intertask-State?
– ja –>
Entferne ggf. DSP-Monitortool.
Informiere Error-Dämon: Text
auf Konsole. Exit.
nein
Wurde Supervisordienst für eine
Task unterbrochen?
– ja –>
Beende den Dienst, reinige den
Systemstack.
nein
–> Auch Einstieg Fälle 2,3: Ausnahme ist an eine Task gebunden.
Ist ein validierbarer
Signal-Frame im Taskkopf
angeschlossen?
– ja –>
Fülle den Signalframe mit dem
Errorcode und weiteren Daten.
Ersetze den Rückkehr-PC durch
die Adresse des
Exception-Handlers. Falls im
Signalframe angefordert:
Zusätzlich Error-Dämon
involvieren. Exit über den
Exception-Handler.
nein
Wenn ein schwerer Fehler
vorliegt: Suspendiere die
verursachende Task. Involviere
den Error-Dämon. Exit.

5.10.2 Exception-Händler in PEARL

Die Signalbehandlung ist über den Anschluss sogenannter `ON`-Blöcke möglich. Bei der Definition des Bearbeitungsmodes kann auf Wunsch der normale Error-Dämon neben dem eigenen Handler seine Meldung machen, allerdings wird die Task dann in keinem Fall mehr suspendiert.

Neben dem Bearbeitungsmode muss eine Variable (die RST-Struktur) angegeben werden, in die das Betriebssystem im Fehlerfall Daten einschreiben kann. Dazu gehört die letzte überlaufene Zeilennummer, die letzte registrierte Modulnummer, der Errorcode, der Program Counter und ggf. der Induceparameter.

Es gibt zwei Stufen der Auslösung, die angewählt werden können.

- Nur schwere Fehler:
  Der PEARL-Code im ON-Block wird nur angesprungen, wenn es sich um eine Ausnahme mit gesetztem Suspend-Bit handelt. In diesem Fall kann die Task nicht an der Fehlerstelle fortgesetzt werden. Es ist aber möglich, per `EXIT`-Anweisung auf den Task-Grundlevel zurückzukehren oder Prozeduren abzubrechen. Über die Shell per `INDUCE`-Befehl gefeuerte Ausnahmen werden wie schwere Fehler behandelt, wenn der Induceparameter negativ ist. Auch wenn der Exception Handler bei einem leichten Fehler nicht angesprungen wird, so gibt es dennoch alle üblichen Einträge in die RST-Struktur. Nach Aufruf einer Prozedur kann man also abfragen, ob in ihr Fehler aufgetreten sind – auch wenn diese nicht zum Abbruch geführt haben.

- Alle Fehler:
  Jede Auslösung wird wie ein schwerer Fehler behandelt.

Sinnvollerweise wird die RST-Struktur als statisch alloziertes Objekt definiert. Damit kann sie am einfachsten auch von Prozeduren aus erreicht werden.

```
#DEFINE X_ERRDM 1; ! Invoke #ERRDM
#DEFINE X_SEO   2; ! Invoke Exception handler Severe Errors Only
#DEFINE X_ALL   4; ! Catch all errors

MODULE EXCDEMO;    ! A simple demo prog
SYSTEM; A1;
PROBLEM;
SPC A1 DATION OUT ALPHIC;
DCL RS STRUCT(/Lino FIXED,       ! Last registered line number
               Mono FIXED,       ! Last registered module number
               Ecount FIXED,     ! Exception counter (all)
               Errcode BIT(16),  ! 16-Bit error code
               PC BIT(32),       ! Program counter to exception
               Indpar FIXED,     ! Induce parameter
               Buffer(100) FIXED ! Reserve for later extensions
                /);

/* +P  */
TA:TASK;
DCL I FIXED;

ON E_(X_ALL+X_ERRDM) RST(RS);     ! Catch all, invoke #ERRDM too
! Task will go here if exception handler is executed
  PUT 'Exception fired at PC:', RS.PC TO A1 BY SKIP,A,B4(8);
  SUSPEND;
  ! Do whatever may be necessary
  EXIT -1;  ! Step one procedure level down if not on task level
  END;

!.... Task may be here for the first time or after exception

IF RS.Ecount > 0 THEN
  PUT 'Restarted', RS.Ecount TO A1 BY SKIP,A,F(4);
ELSE
  PUT 'Initial Start' TO A1 BY SKIP,A;
FIN;

I=2//0; ! Diese Barriere wird nicht ueberwunden

END; ! TASK

MODEND;
```

5.11 Fehlermeldungen zur Compile–Zeit

Obwohl die Meldungen durchweg selbsterklärend sind, soll im Folgenden speziell dem Systemneuling eine Hilfestellung gegeben werden. Es werden die Meldungen des PEARL90-Compilers mit der Version P15.4-A beschrieben. Die älteren Compiler erzeugen ähnliche Meldungen. Diese Beschreibung kann auch für sie benutzt werden.

Zu unterscheiden sind „lokal detektierbare“ Fehler und „bilanz–detektierbare“ Fehler.

5.11.1 Lokal detektierbare Fehler

Der Compiler bettet in das Übersetzungsprotokoll einen Fehlerprompt `*` in einer nach der falschen Zeile stehenden Zusatzzeile ein, diese Zeile enthält am linken Rand die Kennzeichnung `<ERROR>`, um ein Auffinden im Listing zu erleichtern. Der Prompt steht in der unmittelbaren Nähe der Stelle, an der die Abweichung endgültig — auch unter Ausnutzung möglicher anderer Interpretationen — festgestellt werden kann. Auch bei abgeschaltetem Übersetzerprotokoll erscheint die fehlerbehaftete Zeile zusammen mit der Zusatzinformation.

In der Regel gilt eine fehlerhafte Anweisung als insgesamt nicht vorhanden. Dadurch können Folgefehler entstehen. Mit der Vereinbarung von `MAXERR` kann man einen frühen Abbruch der Kompilation erzwingen.

/Syntax violation/ Es wurde keine PEARL–Produktionsregel gefunden. Das kann z. B. durch ein falsches Zeichen, etwa in einer Konstanten `3.14.2`, oder falsch geschriebenes PEARL Schlüsselwort (`TUSK` statt `TASK` etc.) verursacht werden. Dabei gelten gewisse Schlüsselworte nur in der richtigen Blockumgebung als bekannt, z. B. wird bei der Sequenz `MODULE; PUT` *`xyz`* `TO ...` das `PUT` nicht akzeptiert, weil es nur auf Task/Prozedur–Ebene benutzt werden kann.

/Data- or object-types do not match/ Bei der Verknüpfung von Daten oder Zeigern wurden Objekte, die nicht verknüpfbar sind, benutzt. Zum Beispiel wurde der *DURATION*–Variablen `x` der Wert der *FLOAT*–Variablen `z` durch `x=z;` zugewiesen. Meist steht der Fehlerprompt hier am rechten Ende der Anweisung, weil der Compiler erst bei der Zuweisung die Zulässigkeit nach Auswertung der rechten Seite feststellen kann.

Es ist auch möglich, daß eine Zeigervariable oder ein adresslieferndes Objekt erwartet wird und vom Compiler nun nicht vorgefunden wird, z. B. bei der Operation `CONT`, `IS` etc.

/Undefined/ Die Variable bzw. der Prozedurname etc. wurden dem Compiler nicht durch `DECLARE` oder `SPECIFY` bekannt gemacht.

/Double-defined/ Der Identifier ist bereits im Gebrauch und kann nicht neu verwendet werden.

/Limit/ Der Wert einer Konstanten liegt außerhalb der zulässigen Grenzen, z. B. `X = 45196(15)` etc.

/Number of subscripts incorrect/ Beim Zugriff auf ein Feld stimmt die Anzahl der Indizes nicht mit denen der Felddefinition bzw. -Spezifikation überein, z. B. `DCL A(2,3) FIXED; A(I,J,K) = 5`.

/Dation use or direction incorrect/ Die Datenstation ist per Definition/Spezifikation oder auf Grund von Kenntnissen des Compilers nicht in der Lage, wie im Text beabsichtigt zu funktionieren. Beisp: `SPC xyz DATION OUT ...; ...; GET ... FROM xyz`.

/Excessive INIT-data/ Es wurden im `INIT` mehr Objekte gefunden als in der zugehörigen Deklarationsliste. Der Compiler kann die überzähligen Elemente nicht zuordnen, sie werden ignoriert.

/You cannot modify invariant objects!/ Einem Objekt mit `INV`–Attribut soll zur Laufzeit ein Wert zugewiesen werden. Das kann auch durch Einsetzen als Prozedurparameter entstehen, wenn im `IDENT`-Mode transferiert wird. Der Compiler weiß dann natürlich nicht, ob die Prozedur das Objekt im konkreten Fall wirklich verändert. Immerhin könnte sie es tun.

/Blockstructure/ Block–Struktur verletzt, z. B. `IF ... THEN ... END;` oder `BEGIN; FIN;`. Tritt häufig als Folgefehler auf.

/INIT-list too short/ Es soll eine benamte Konstante mit DCL definiert werden, jedoch fehlt in der `INIT`-Liste ein zugehöriger Wert.

/Parameter(list) incorrect/ Eine Prozedur kann mit der angegebenen Parameterliste nicht aufgerufen werden. Oft steht der Marker am rechten Ende der Anweisung und zeigt nicht auf den verursachenden Parameter, weil der Fehler erst in einer späten Phase beim Feinabgleich von Aktual- und Formalparametern (Strukturinnenleben etc.) erkannt wurde.

/Parameterlist: mismatch with earlier SPECIFY/ Eine Prozedur wurde mit `SPC` vorab bekannt gemacht. Bei der jetzt erfolgenden Prozedurdefinition stimmen Parameterliste oder Ergebnistyp nicht mit der Vorabspezifikation überein. Der Fehler tritt bei der Übertra-

gung von PEARL80-Programmen häufig auf, weil der alte Compiler die Vorabspezifikation nicht wirklich benutzte.

/Missing parameterlist/ Die Benutzung der Prozedur oder des Operators erfordert an dieser Stelle eine Parameterliste, z.B. muß ein Zeiger auf eine Prozedur dereferenziert werden, es fehlt jedoch die Parameterliste.

/You must write 'ACTIVATE..'!/ Ein Taskname oder ein Zeiger auf eine Task wurde als Instruktion hingeschrieben, wie es zwar bei Prozeduren, nicht aber bei Tasks erlaubt ist.

/Size of object undefined for compiler/ Der Compiler benötigt für die Operation die Anzahl Bytes, die für das Objekt erforderlich ist, hat diese Information hier aber nicht verfügbar. Wird z.B. ein virtuelles Feld in einem `DCL ..` benutzt oder die Totalzuweisung eines solchen Feldes versucht, so erscheint diese Fehlermeldung.

/Cannot store result/ Es wurde eine Funktion aufgerufen, die einen Wert oder einen Zeiger zurückgibt. Durch die Art des Aufrufes kann jedoch kein Ergebnis abgespeichert werden.

/Too many large parameters by value/ In der Praxis nie zu sehen, aber prinzipiell möglich. Beim Aufruf einer Prozedur oder Funktion wurden zu viele platzverbrauchende Parameter mit knapp weniger als 256 Byte Größe per value übergeben, sodaß der dafür vorgesehene „Parameterspace“ von ca. 16 kByte nicht ausreicht. Tritt auf, wenn z.B. mehr als 63 Parameter vom Typ `CHAR(255)` per value übergeben werden. Sollte das tatsächlich einmal ein Problem sein, so sollten Sie zunächst versuchen, diese gigantische Parameterliste zu verkürzen. Vielleicht können Sie ja auch einige Parameter in Strukturen zusammenfassen oder per `IDENT` übergeben. Große Objekte (Strukturen mit Feldern darin etc.) mit mehr als 256 Bytes belasten den Parameterspace in beiden Übergabemodes nur mit 4 Bytes. Gleiches gilt für kleinere Objekte im `IDENT`-Mode. Der Compiler fertigt im „per value“ Mode bei großen Objekten nämlich auf der Aufruferseite eine Kopie im Variablenraum des Aufrufers an und übergibt lediglich einen 4 Byte Zeiger an die Prozedur.

/REF by IDENT: actual no REF./ Die aufgerufene Prozedur erwartet auf einem Platz eine Zeigervariable by IDENT doch das aktuell angebotene Objekt ist keine echte Zeigervariable. Kann es nur eine Adresse liefern, so reicht dies für eine REF by value zwar aus, nicht jedoch für den hier erforderlichen Zeigerbezug auf eine

Zeigervariable.

/Sorry! Compilerlimit exceeded/ In der Praxis nie zu sehen, aber prinzipiell möglich. Zuviele kleine lokale Objekte überlasten den 32 kByte Raum für sehr schnell adressierbare kleine lokale Prozedur- oder Taskobjekte. Platz für ein weiteres Objekt dieser Art ist nicht mehr vorhanden. Abhilfe durch Verlagerung von Aufgaben auf Prozeduren, die dann ja weiteren eigenen schnellen lokalen Speicherraum haben. Auch die Zusammenfassung zu Datenstrukturen kann helfen, ist aber meist weniger schnell: Lokale Felder und größere Strukturen legt der Compiler stets in einen besonderen lokalen Adressraum, der zwar (speziell bei den RISC-Prozessoren) nicht ganz so schnell erreichbar, dafür aber nur durch den vorhandenen Speicher begrenzt ist.

/Not(yet)Implemented/ Der Compiler hat ein Konstrukt zwar erkannt, dieses ist in der aktuellen Version aber noch nicht implementiert. (Baustelle der PEARL90-Norm)

/Internal Compiler-Error/ Der komplexe interne Selbsttest des Compilers hat einen Fehler entdeckt. Wir hoffen, daß Sie diese Meldung nicht zu sehen bekommen. Bleibt sie bei einem neu gebooteten System bestehen, so sollten Sie uns informieren.

5.11.2 Bilanzdetektierbare Fehler

MISSING PROC/TASK Dem Compiler wurde durch `SPECIFY` vorgeschwindelt, daß die angegebenen Prozedur/Task weiter unten im Modul noch definiert werden. Nun — zum `MODEND` — wird dem Compiler klar, daß er nicht mehr hoffen darf.

MISSING LABELS Die angegebenen Marken wurden zwar angesprochen, aber nicht oder nicht blockkonform definiert.

SIZE_LIMIT–ERROR Die tatsächliche Größe des Modules übersteigt den durch `S=...` (bzw. default) festgelegten Kopfeintrag. Das Modul kann nur mit zusätzlichem `SZ`–Parameter beim `LOAD` geladen werden.

5.11.3 Nicht sprachbedingte Abbruchkonditionen

Maxerr: Limit exceeded. Mit dem letzten aufgetretenen Fehler wurde die gesetzte Grenze überschritten. Die Kompilation wird vorzeitig abgebrochen.

Local scalars >32kB In der Praxis nie zu sehen, aber prinzipiell möglich. Der Übersetzungslauf wurde abgebrochen, weil sich erst bei der Codegenerierung zum letzten PEARL-Statement herausgestellt hat, daß der kleine schnelle lokale Prozedur- oder Task-Workspace infolge weiterer intern benötigter Hilfsobjekte nicht ausreicht. Die Abhilfe ist die gleiche wie beim oben beschriebenen „/Sorry! Compilerlimit exceeded/“.

INCLUDE-file ended inside statement Mitten in der aktiven Übersetzungsphase einer Anweisung wurde das Ende des zu „includenden“ Files erreicht. Vielleicht fehlt ein Semikolon am Ende oder die letzte Anweisung ist falsch.

Cannot open INCULE-file Der einzufügende File ist nicht vorhanden oder steht in einem anderen Verzeichnis.

Outp. failed Der Compiler kann nicht schreiben, typischerweise wegen einer Irregularität beim Code-Output (Platte voll?). Beim List-Output unterbleibt meistens diese Meldung, gelegentlich kann sie über den Standard Error Kanal ausgegeben werden.

Cannot read from input-file Der Eingabefile produzierte einen Lesefehler.

Premature end of input-file File zu Ende, bevor `MODEND` akzeptiert wurde.

Internal Compiler-error Bitte schreiben Sie uns, wenn klar ist, daß Ihr System ansonsten in Ordnung ist!!

Insufficient memory. SZ-Para? Nicht genügend Listenplatz, möglicherweise war der Parameter `SZ` beim `P`–Kommando zu klein. Man beachte, daß es beim Compiler keine eigene Limitierung der Länge einer Anweisung gibt. Eine Formel, die sich über hunderte von Zeilen hinzieht, kann bearbeitet werden, solange genügend Listenplatz verfügbar ist. Ein Aufbrechen von solchen Riesenstatements entspannt logischerweise die Situation.

5.11.4 Warnungen

Wird nach einem öffnenden Kommentar ein weiterer öffnender Kommentar gefunden, bevor der erste Kommentar geschlossen wurde, gibt der Compiler eine entsprechende Warnung aus. Sind mehrere solcher Stellen in einem Modul, wird die letzte gefundene Zeile angegeben. Solche Warnungen sind zu beherzigen und das Konstrukt sollte elimiert werden!

Beispiel: `/* ....* / ALT x=5; ... /* ........ */;`

Im Beispiel wurde der erste Kommentar versehentlich nicht beendet. Ohne die Warnung des Compilers würde die komplette Alternative einfach verschluckt, und der Nutzer müßte eine sehr langwierige Fehlersuche starten.

5.11.5 Abschlußmeldungen

In der Compilerschlußbilanz werden folgende Informationen ausgegeben:

`TASKS:`

> `(INT)` im Modul vereinbarte Task
> `(EXT)` extern angesprochene Task
> `(***)` angesprochen, aber nicht als globale Task spezifiert.

`Internal Procedures/Functions:`

> `(FUN)` im Modul vereinbarte Funktion
> `(PRO)` im Modul vereinbarte Prozedur

`Extra Devices:`

> Einige wenige Standardgeräte sind dem Compiler bekannt, so etwa /ED, /A1 etc. Unter dieser Rubrik werden alle Datenstationen des Systemteiles aufgelistet, die ihm nicht bekannt sind. Man muß später beim Laden dafür Sorge tragen, daß diese im Zielsystem unter genau dem Gerätebezeichner vorhanden sind oder aber vor dem Laden mit Hilfe des Linkers die notwendigen Informationen über LDN und DRIVE anfügen.

`VAR(RAM):...`

> Länge des für RAM übersetzten `VARi`ablenbereiches

`VAR(ROM): ...`

Für EPROM übersetzter `VAR`iablenbereich, mit Angabe des Adreßbereiches in dem beim Systemstart aus dem EPROM der `VAR`iablenbereich als Modul angelegt wird.

`CODE(RAM): ...`

Für RAM übersetzter CODE, mit Angabe des relativen Offsets in den S-Records.

`CODE(ROM): ...`

Für EPROM übersetzter CODE, mit Angabe des absoluten Adreßbereiches im EPROM, auf den der Code abgelegt werden muß (Ausnahme siehe Ausgabe `SHIFTABLE`).

`$... BYTES`

Gesamtlänge der erzeugten `S`–Records (VAR+CODE).

`(FOR 68...[+68881])`

Angabe des Prozessortypes, für den der Code übersetzt worden ist.

`...ERRORS`

Angabe der bei der Übersetzung ermittelten Fehler.

`SIZE_LIMIT_ERROR`

Falls mehr Platz für VAR+CODE bei der Übersetzung ermittelt wurde, als in der `S[C]`–Option angegeben wurde, wird diese Meldung ausgegeben (zählt als 1 Fehler in der Bilanz).

Wenn der Compiler ROM-Code erzeugen soll, so gibt es eine weitere Angabe darüber, ob der erzeugte Code frei verschieblich ist. Dann kann er an beliebiger Stelle im EPROM und nicht nur ab der Adresse, die bei `CODE= ...` angegeben wurde, abgelegt werden. In der Schlußmeldung wird dann

```
SHIFTABLE
```

ausgegeben. Falls der Code nicht verschieblich ist, wird die Meldung

```
HARDWARE ADRESS
```

in der Schlußbilanz erscheinen.

Bei der Übersetzung von Shell–Modulen wird ebenfalls angegeben, ob die erzeugte Shellerweiterung später an feste Adressen gebunden ist oder nicht. Bei der Schlußmeldung:

```
>>USABLE_ALSO_INSIDE_RTOS_KERNEL_...
```

kann das übersetzte Modul nach einem Linkerlauf oder mit Hilfe des PROM-Befehles in (Programmier-) gerätegeeignete S-Records verwandelt werden. Der Code kann danach auf beliebigen Stellen im EPROM oder im Boot-Memory des Systemes plaziert werden.

Lautet dagegen die Schlußmeldung:

```
>>USE_ONLY_AS_LOADED_RAM_SHELL_....
```

so ist eine freie Ablage innerhalb des EPROM oder Boot-RAM nicht möglich. Mit Hilfe des `PROM`-Befehles werden die Ablageadressen (`CODE`, `VAR`) des Quellfiles fest eingearbeitet. Mit dem Linker an Stelle des `PROM` können diese zwar auch später noch verändert werden, doch sind die erzeugten S-Records nur an den exakten physikalischen Adressen in den EPROMS bzw. dem Boot-Memory verwendbar.

5.12 Fehlermeldungen zur Laufzeit

Bei Auftreten eines der unten aufgelisteten Fehler wird die entsprechende Meldung ausgegeben und die verursachende PEARL–Task läuft nach den beschriebenen Aktionen weiter. Einige Fehler führen zu einer Terminierung oder Suspendierung der PEARL–Task, dies ist bei den entsprechenden Meldungen vermerkt. Wenn das Programm mit der Markierungsoption `/*+M*/` übersetzt wurde (zumindest teilweise), so wird vor der Fehlermeldung die letzte *registrierte* Hochsprachzeilennummer ausgegeben. In Programmbereichen mit abgeschalteter Markeroption werden keine neuen Nummern registriert!

DV/0 Divided by zero. Es wurde versucht, durch Null zu teilen. Für das Resultat wird die größte mögliche Zahl eingesetzt.

END–OF–FILE Das Ende eines Files ist beim Lesen überschritten worden. Der File wurde vor dem Lesen nicht auf den Anfang gesetzt (`REWIND(`*`dation`*`)` vergessen?).

FL.OV Floating overflow. Betrag des Resultates ist größer, als es der verwendete Gleitkommadatentyp zuläßt.

INPUT–SYNTAX Input–character violates format–syntax. Das Zeichen paßt nicht zum Eingabeformat, z. B. Buchstabe bei `FIXED`–Format.

ILL) Illegal character ')' in FORMAT → destructed machine-code. Die Anzahl rechter Klammern stimmt nicht. Kann nur bei falscher Hyperprozessor–Benutzung im Assemblerprogramm oder zerstörtem Programmcode passieren.

IOFM I/O–FORMAT does not conform to data–type in list. Das Format paßt nicht zum Datentyp des Elements.

IONS I/O not set up → Destructed machine–code. I/O nicht eröffnet! Kann nur bei zerstörtem Code oder fehlerhaftem Assemblerprogramm passieren. Terminiert die Task.

NDSF No data–spec. in FORMAT found. (`PUT X TO ... BY SKIP;`) Die Formatanweisung für das Datum fehlt völlig.

NDUR Negative duration (`AFTER ...`, `ALL ...` etc.). Die Zeitdauer ist negativ.

NIEX Negative input–exponent for `FIXED` number. Der Exponent für die Eingabe einer `FIXED`–Zahl ist negativ.

NIM–D0 Not implem. hyperproc–instruction. Opcode in register D0 oder r0. Die V–Number in D0 ist nicht implementiert. Die geladenen

S–Records stammen wahrscheinlich von einem neuen Compiler, für dessen Code das veraltete Laufsystem nicht mehr ausreicht. Bitte ggf. Systemupdate besorgen.

OBIN Overflow `B`–formatted input. Die Binärziffernfolge ist länger als das angegebene Format.

OEXI Overflow exponent on (numeric) input. Der Exponent ist zu groß oder zu klein.

OPNDIF Open by `IDF ...` Syntax or length error. Die Zeichenkette beim `IDF` ist fehlerhaft.

PNUM Tritt nur noch beim veralteten PEARL80 Compiler auf: Parameter–numbers on caller–/proc–side not equal. Die Anzahl der Parameter auf der Aufruferseite stimmt nicht mit Anzahl der Prozedurseite überein. Terminiert die PEARL–Task. In PEARL90 wird der Test zur Compilezeit gemacht.

PTYP Tritt nur noch beim veralteten PEARL80 Compiler auf: Parameter–types on caller–/proc–side: no match. Der Parametertyp stimmt nicht überein. Terminiert die Task. In PEARL90 wird der Test zur Compilezeit gemacht.

PXFR Tritt nur noch beim veralteten PEARL90 Compiler auf: Parameter „xfer“ illegal. (`IDENT`–proc + `INV`–call) Der Aufruf stimmt nicht. Terminiert die PEARL–Task. In PEARL90 wird der Test zur Compilezeit gemacht.

X/PAG– X or PAGE –count negative. (`A=--3 ... X(A)`,`PAGE(A)`, ...). Die Anzahl Blanks oder `PAGE` ist negativ.

PATHLT: TOO LONG In der angegebenen Pathlist wurde ein zu langer Bezeichner gefunden, d.h. der im Systemteil angelegte Platz reicht nicht aus.

READ Can't read from inputfile. (`GET` + file empty etc.). Es kann nicht vom Eingabefile gelesen werden.

RND/EN `ROUND/ENTIER` overflow. Result > `FIXED(15)`. Das Resultat einer Rundung paßt nicht in eine `FIXED(15)` Zahl.

SKP– `SKIP`–count is negative. (`A=--3; ... SKIP(A),...`). Die Anzahl `SKIP`s ist negativ.

SUSP: TASK NOT FOUND Extern zu suspendierende Task wurde nicht gefunden. Tritt bei einem Fremd–Suspend auf.

TIME–OUT Time out error. Nur bei bestimmten I/O–Treibern: Die Zeitüberwachung hat angesprochen, Gerät antwortet nicht.

XIOV `FIXED`–number input–overflow. Die einzulesende Zahl paßt nicht in die `FIXED`–Variable.

(OVF Left bracket or „`R`“–FORMAT nesting overflow. Zuviele öffnende Klammern beim Format.

5.12.1 Fehlermeldungen der implementierten mathematischen Einbaufunktionen

Die folgenden Fehlermeldungen werden von den mathematischen Einbaufunktionen zur Laufzeit eines Programmes generiert. Die Task, die den Fehler verursacht, läuft weiter. Es wird versucht, ein möglichst sinnvolles Resultat der entsprechenden Operation zu liefern.

ASIN/ACOS OVERFLOW Das Argument eines `ASIN` oder `ACOS` ist größer `1.0`. Als Ergebnis der Operation wird das Argument zurückgeliefert.

EXP OVERFLOW Das Ergebnis der Funktion e^x wird zu groß. Als Ergebnis wird die größtmögliche Zahl zurückgkeliefert.

LOG OVERFLOW Das Argument der `LOG`–Funktion ist ≤ 0. Als Ergebnis wird die größte mögliche negative Zahl zurückgeliefert.

DSC OVERFLOW Das Argument einer `SIN`– oder `COS`–Berechnung ist größer als 2^{23}. Es wird eine 0 als Ergebnis zurückgeliefert.

SQR OVERFLOW Das Argument einer Quadratwurzel ist negativ. Es wird die positive Wurzel berechnet.

TAN OVERFLOW Das Argument einer `TAN`–Berechnung ist größer als 2^{23}. Das Ergebnis ist eine 0.

Wird ein 68881/68882 Coprozessor zur schnelleren Rechnung eingesetzt, können folgende Fehlermeldungen auftreten:

ZERO–DIV FPU–68881 Es wurde versucht, durch 0 zu teilen.

WRONG OPERAND FPU–68881 Der Operand ist z. B. zu groß/klein, keine Zahl o. ä.

OVERFLOW FPU–68881 Das Ergebnis der Operation ist zu groß.

! → Bei einem FPU–Fehler wird die „schuldige“ Task suspendiert!!!

Die Meldungen gelten für den PowerPC ganz analog.

(Leere Seite vor neuem Kapitel)

Kapitel 6: Datenstationen

6.1 Datenstationen A*x*, B*x*, C*x*, UL

Bei diesen Stationen handelt es sich entweder um emulierte Terminals oder um serielle Schnittstellen (RS 232, RS 458 etc.). Für `/A`*x*, `/B`*x*, `/C`*x* gilt — bei gleichem x — stets eine gemeinsame `LDN`, die Buchstaben `A`, `B`, `C` werden also lediglich in eine Untergliederungsnummer (=Betriebsart) umgewandelt. Die Datenstation `/UL` kennzeichnet eine Sonderbetriebsart der Station mit x=2.

!

Da für beide Übertragungsrichtungen nur eine Warteschlange zur Verfügung steht, ist ein echter Vollduplexbetrieb mit diesen Stationen nicht möglich (siehe `/D`*x*-Station). Solange also etwa eine Eingabe „hängt“, tritt in der Warteschlange Stillstand ein. Um wenigstens die Operation „mal sehen, ob was da ist“ für die Eingabe machen zu können, wurde die Betriebsart `C` geschaffen.

Ausgabe

Es gibt zwischen `/A`*x*, `/B`*x*, `/C`*x* und `/UL` keinen funktionellen Unterschied. Die Ausgabe kann vom Empfänger jederzeit mit Hilfe eines ausgesendeten X_{off} (`Ctrl S`) angehalten werden. Mit Hilfe des X_{on} (`Ctrl Q`) wird dann die Wiederaufnahme der Sendung angefordert (X_{on}/X_{off}–Protokoll). Auch emulierte Terminals, etwa beim Apple Performa oder innherhalb von Textfenstern des Window-Managers, können auf diese Weise angehalten werden.

In den meisten Systemen kann über besondere Bedienbefehle die Baudrate verändert werden.

Zusätzlich zum Softwareprotokoll (X_{on}/X_{off}), das bei binärer Betriebsart der Schnittstelle abgeschaltet ist, wird ein Hardwareprotoll mit RTS/CTS unterstützt. Diese beiden Leitungen sind zu brücken, wenn kein Hardwareprotokoll erwünscht ist.

!

Die zugehörige I/O–Task „merkt sich“, ob sie als letzte Anforderung über `A` oder über `B/C` angesprochen wurde. Dies steuert ihr Verhalten hinsichtlich des unten beschriebenen Eingabekanales.

Eingabe ohne Initiative der I/O–Task

1. Die Station war zuletzt als `/A`*x*–Station in Benutzung.

   Die unaufgefordert empfangenen Daten werden in einen implementierungsabhängigen Eingabepuffer (Ringpuffer) geschrieben. Es wird keine Reaktion zum Sender ausgesendet.

2. Die Station war zuletzt als `/B`*x* oder `/C`*x* in Benutzung.

   Auch hier werden die unaufgefordert empfangenen Daten in den begrenzten Ringpuffer ($\geq$ 31 Zeichen) genommen, allerdings wird nun ab dem Moment, in dem der Ringpuffer zur Hälfte gefüllt ist, für jedes Zeichen, das auf den Puffer geht, ein X_{off} zum Sender zurückgesendet, um diesen zu stoppen. Außerdem wird der Sender auch über die Hardware–Handshake Leitung aufgefordert, seine unerwünschte Datensendung zu unterbrechen.

Eingabe mit Initiative der I/O–Task

1. Expliziter Lesebefehl für die Station `/A`*x*.

   Die evtl. im Eingaberingpuffer befindlichen Daten werden eliminiert (Puffer wird gelöscht) und sind damit verloren. Wurde ein X_{off} ausgeschickt, so wird jetzt wieder ein X_{on} geschickt bzw. über die Handshake Leitung wird der „gegnerische“ Sender wieder freigegeben. Die I/O–Task hängt sich jetzt auf (`SUSP`), bis die angefordete Anzahl von Daten eingetroffen ist. Beim Lesen mehrerer Sätze und ständig laufendem Eingabestrom gehen bei dieser Betriebsart in den `DORM`–Phasen der I/O–Task zwangsläufig Daten verloren, denn es wird kein Protokoll erzeugt. Die `/A`*x* sind als Eingabestationen praktisch nur für die Terminaleingabe des Bedieninterface geeignet, weil es dort gerade erwünscht ist, daß evtl. in der inaktiven Phase angeschlagene Zeichen (Affe, der auf die Tasten hämmerte...) keine Auswirkungen haben.

2. Expliziter Lesebefehl für die Station `/Bx`, `/UL`.

   Die im Eingaberingpuffer befindlichen Zeichen werden übertragen. Falls diese bereits die Anforderung erfüllen konnten, so ist die Operation — ggf. mit aufgehobenem Rest im Eingaberingpuffer — beendet.

   Beim Lesen von der Station `/UL`, entsprechend der Station `/B2`, werden empfangene LF's nicht übernommen. Diese Option ist hauptsächlich zum Laden von S–Records von einem Host–Rechner geeignet (`/UL` = Up–Load), kann aber auch immer dann eingesetzt werden, wenn vom Host zusätzliche LF's generiert werden („Aufblähen" des CR zur Kombination CR/LF). Reichen die im Puffer befindlichen Daten nicht aus, so geht die I/O–Task in den Zustand `SUSP` bis zu dem Zeitpunkt, zu dem die erforderliche Anzahl Daten erreicht wird. Beim Lesen mehrerer Sätze und einem fortlaufenden Eingabestrom kommt es hier nicht zu einem Datenverlust, da die Betriebsart `B` das X_{on}/X_{off} bzw. RTS/CTS Protokoll benutzt und damit auch in den `DORM`–Phasen der I/O–Task keine Daten verloren gehen (sofern der Sender rechtzeitig mit Senden aufhört!!).

3. Expliziter Lesebefehl für die Station `/Cx`.

   Wenn keine Daten im Eingaberingpuffer stehen, so wird dem Auftraggeber nur das Zeichen NUL (`$00`) mit der Satzlänge `1` (`RECLEN = 1`) übermittelt, und der Auftrag wird als erledigt behandelt.

   Stehen im Eingaberingpuffer weniger Zeichen als angefordert, so werden diese übertragen, und die erreichte Satzlänge wird dem Auftraggeber mitgeteilt. Der Auftrag gilt danach als erledigt.

   Stehen im Eingaberingpuffer ausreichend Zeichen, so wird — unter Erhalt eines evtl. Restes — der Auftrag daraus erledigt.

   Eine Steuerung des Senders erfolgt auch auf dem C-Port mit dem Softwareprotokoll über X_{off} und X_{on} bzw. mit den Hardwaresignalen RTS und CTS. In diesem Punkt gibt es keinen Unterschied zur Betriebsart `B`.

   Die Betriebsart `C` ist also gedacht, um ohne Risiko des Aufhängens das Eingabeport abzufragen, ob denn „Daten da sind". Beim Lesen aus Hochsprachprogrammen über `C` muß der Programmierer selbst auf die NUL abprüfen — dann waren noch keine Daten eingetroffen. Danach kann ruhig ein kleines Päuschen (`AFTER x SEC RESUME`) eingelegt werden, um den Rechner nicht nur mit der Abfrage der Datenstation zu beschäftigen! Besser ist aber der Verzicht auf die `C`–Station und Benutzung der

Time–Out Funktion.

Die C–Station kann genutzt werden, um den Eingaberingpuffer definiert zu leeren und trotzdem keine Zeichen zu verlieren. Es wird ein GET vom C–Port mit einer TFU von 128 gemacht.

Das Bedienkommando COPY (s. Seite 115) ist auf das NUL–Zeichen und die Satzlänge 1 abgerichtet: Nach Einlesen des NUL–Zeichens legt sich COPY für 8 Millisekunden „aufs Ohr“, um danach die Abfrage zu wiederholen.

Mit `COPY.W /C1/>/B2/; COPY.R /C2/>/B1/`

kann man seinen Rechner in so etwas wie ein Terminal für den angeschlossenen Host verwandeln. Mit Ctrl A wird dann bei Bedarf die Verbindung zum **RTOS–UH** wiederhergestellt.

Auch das Lesen vom /C*x* kann zum Aufhängen führen, nämlich dann, wenn die Schlange durch eine liegengebliebene Ausgabe oder durch eine Eingabe der Betriebsart A bzw. B „verstopft“ ist.

Time–Out

Für die seriellen Schnittstellen steht eine Time–Out Funktion zur Verfügung. Das Time–Out bezieht sich immer auf ein komplettes Auftragselement (CE). Das Zeitraster für ein Time–Out kann in 512 msec Schritten eingestellt werden. Die maximale Anzahl Schritte ist 127, so daß das längste mögliche Time–Out auf 65,024 sec gesetzt werden kann.

Wenn das Time–Out eintritt, wird die Fehlermeldung TIME-OUT (mit NE unterdrückbar) ausgegeben und ST=7 gesetzt. Das betroffene CE wird vom Treiber an den Aufrufer zurückgegeben oder — bei gesetztem Verschrottungsbit (z. B. Ausgabe ohne Wait) — in freien Speicher verwandelt. Die evt. bei einer Eingabe schon gelesenen Daten sollten nicht mehr verwendet werden. Das Setzen des Time–Outs in PEARL wird im Systemteil des erledigt. Der Assembler–Programmierer muß die Anzahl Schritte im linken Byte des DRIVE–Wortes eintragen (siehe CE-Beschreibung ab Seite 559).

Bedieninterface

Stationsname ist /A*x*, /B*x*, /C*x*, z. B. /A1, /C2, /B2. Ein eventuell angegebener Pfadname wird voll in die Verwaltung übernommen (S–Befehl), hat aber

keinerlei Funktion.

Beispiel:
```
O /B2; DM 1000 2000
P >/B2; COPY /B3>/B1
```

PEARL–Programm

Systemname ist `/Ax`, `/Bx`, `/Cx` z. B. `/A1; /C2; /B2;`. Eine eventuell nachgestellte *pathlist* wird voll in die Verwaltung übernommen (`S`–Befehl), hat aber keine funktionelle Bedeutung.

Beispiel:
```
SYSTEM; Output:/B3− >;
Inputdevice:/B2< −;
A1; /* Kurzform username=systemname*/;
...
PROBLEM;
...
SPC A1 DATION INOUT ALPHIC;
SPC Inputdevice DATION IN ALPHIC;
SPC Output DATION OUT ALPHIC CONTROL(ALL);
...
GET ... FROM Inputdevice BY ...
PUT ... TO Output BY ...
```

Weitere Einzelheiten sind beim Sprachumfang des PEARL–Compilers zu finden.

6.2 Datenstation BU

Die Datenstation `/BU` läßt den direkten Zugriff auf Prozeßperipherie über allgemeine Peripherieadressen oder über den Unterbus (P–bus, nicht bei allen Systemen implementiert) zu. Zu dieser Station existiert keine Betreuungstask, da über die Peripherieadressen bzw. den Unterbus nur ungepuffert und unquittiert ein-/ausgegeben werden kann. Das Betriebssystem ist an dem Transport nicht beteiligt. Die `/BU`–Stationen können also nicht vom Bedieninterface angesprochen werden (Ausnahme: `DM-P` für P–bus). In Systemen, die Mehrrechnerbetrieb zulassen (z. B. VMEbus), können Synchronissationsmaßnahmen unterstützt werden, die wie globale Semaphore wirken.

PEARL–Programm

Der Systemname ist `BU(`*`hexadr8, Zugriffscode`*`)`, alternativ (bei 68000–er Rechnern, die nur 6–Byte–Adressen verarbeiten können) `BU(`*`hex8`*`)`, wobei der Zugriffscode zweistellig vor der Peripherieadresse angegeben werden muß. Die Datenstation gilt als vom Typ `BASIC`. Die Übertragungsrichtung muß angegeben werden. Das Bitmuster *hexadr8* muß in Kodierung und Adresse der aktuell angeschlossenen Hardware entsprechen.

Um möglichst viele unterschiedliche Peripheriekarten verwenden zu können, wurden folgende Zugriffsmöglichkeiten geschaffen:

In der Anweisung

`VENTIL: BU(`*`y`*`,`*`x`*`) − >;`

bedeuten: y = Adresse des Peripheriebausteins in *hex8*.
x = Kodierung der Zugriffsart, einstellig dezimal.

Folgende Kodierungen sind zugelassen:

CODE	Übertragung	wird abgelegt auf
0	P–Bus, WORD oder LSB	
1	8 bit, MSB (MOVE.B)	y
2	16 bit, MSB (MOVE.W)	y, y+1
3	32 bit, MSB (MOVE.L)	y, y+1, y+2, y+3
4	16 bit, MSB (MOVEP.W)	y, y+2
5	32 bit, MSB (MOVEP.L)	y, y+2, y+4, y+6
6	8 bit, LSB (MOVE.B)	y
7	P–Bus, WORD oder MSB	
8	GLOBAL SEMA (TAS)	

Alle einfachen Datentypen sind zugelassen (Ausnahme: Kodierung `$08`). Bei Float–Variablen wird normalisiert bzw. denormalisiert. `CHAR`-Variablen werden linksbündig beschrieben und ggf. mit Leerzeichen aufgefüllt.

Bei Kodierung `$08` ist eine `BIT(1)`–Variable zu verwenden. Die `TAKE`–Anweisung benutzt den Assembler–Befehl `TAS` und liefert den Wert `'1'B1`, wenn der Pseudo–Request erfolglos war (dann stand bereits eine `'1'` an der getesteten Speicherstelle) oder `'0'B1`, wenn der Pseudo–Request erfolgreich war (dann stand bisher eine `'0'` an der Speicherstelle, die vom `TAS`–Befehl mit einer `'1'` überschrieben worden ist). Die Kodierungen `$04` und `$05` sind für einige Bausteine aus 8–Bit–Prozessor–Familien erforderlich.

Beispiel 1: Zugriff über Peripherieadressen

```
SYSTEM;
   FUEHLR: BU(0240FFFC) <-;
   /* MC 68000: Adr=$40FFFC, Zugriffscode=2 */;
   ...
PROBLEM;
   SPC FUEHLR DATION IN BASIC;
   ...
   R: TASK;
      DCL TEMP BIT(16);
      TAKE TEMP FROM FUEHLR;
```

Beispiel 2: Zugriff über den Unterbus

```
SYSTEM;
   VENTIL: BU(42) ->;
   ...
PROBLEM;
   SPC VENTIL DATION OUT BASIC;
   ...
   T: TASK;
      SEND '008F'B4 TO VENTIL;
```

Beispiel 3: Globale Pseudo-Semaphore

```
SYSTEM;
   S1: BU(FF0F0006,8) <->;
   /* MC 68020, Adr=$FF0F0006, Zugriffscode=8 */
   ...
PROBLEM;
   SPC S1 DATION INOUT BASIC;
   ...
   INIT: TASK;
      SEND '01'B1 TO S1;
      ...
   X: TASK;
      DCL FLAG BIT(1);
      ...
      FOR I TO MAX WHILE FLAG REPEAT;
         TAKE FLAG FROM S1;
      END;
      ...       (geschuetzter Bereich)
      SEND '0'B1 TO S1;    (Peudo-Release)
```

Warnung!

Da sich die Peripherieadressen nicht von Speicheradressen unterscheiden, können durch falsche Adreßangaben in der `/BU`–Anweisung auch Zugriffe auf den von **RTOS–UH** benutzten Speicherbereich erfolgen. Mit `SEND` auf diesen Bereich ist das Überschreiben von System– und Anwendersoftware möglich.

6.3 Eigene BU–Datenstation

Sind die vom Compiler zur Verfügung gestellten /BU–Datenstationen einmal nicht ausreichend, z. B. weil ein Zugriff im Supervisormode notwendig ist, so kann eine eigene Datenstation generiert werden. Man hat dazu im Prinzip alle Register mit Ausnahme von A4, A5, A6 und A7 zur freien Verfügung. Beim PowerPC dürfen die Register r12, r13, r14 und r15 nicht verändert werden. Das Register D0 (r0 beim PowerPC) dient zum Datentransfer. Der Stack hat nur noch Platz für einen weiteren BSR-Level.

Bei den RISC-Prozessoren (PowerPC) legt der Compiler zur Geschwindigkeitssteigerung die Rückkehradresse zunächst nicht auf den Stack sondern in das Link-Register. Daher ist als Rücksprungbefehl nicht RTS sondern XRTS zu verwenden, der vom 68K-Assembler wie ein RTS übersetzt wird. Bei den RISC-Transferassemblern jedoch wird aus dem XRTS ein sehr schneller Sprung mit dem Link-Register als Zeiger. Das Link-Register wird leider schnell zerstört, sowohl durch PC-relative Operationen des Transferassemblers als auch durch Systemtraps und Unterprogrammaufrufe. Mit dem XSL-Befehl (Xtended Save Link) kann sichergestellt werden, daß die Rückkehradresse anschließend in jedem Fall auf dem Stack steht: die Rückkehr kann dann mit dem normalen RTS korrekt erfolgen. (Da bei den 68K-Prozessoren die Rückkehradresse immer auf dem Stack steht, wird der XSL vom 68K-Assembler ignoriert.)

Bei der PEARL-Verwendung einer benutzerdefinierten BU-Station wird deren Beschreibung im SYSTEM–Teil weggelassen, die Station wird lediglich im Problemteil GLOBAL spezifiziert. Mit Hilfe eines kleinen Maschinenprogrammes in Transferassemblersprache kann der Zugriffscode realisiert werden. Im folgenden sind die Code–Sequenzen angegeben, die der Compiler bei den Standard BU-Stationen generiert:

PEARL–System Definition: `sta1: BU(adr1,1);`

(Transfer-) Assembler Realisierung:

```
>sta1    BRA.S    IN         Einsprung für Einlesen
         ASR.W    =8,D0      rechtsschieben um ein Byte
         _MOVE.B  D0,adr1    Zugriff auf Peripherie unter adr
         XRTS                Rücksprung (by stack or linkreg)
IN       CLR.L    D0         löschen
         MOVE.B   adr1,D0    Wert einlesen
         _ASL.W   =8,D0      in high-Byte schieben
         XRTS                Rücksprung (by stack or linkreg)
```

PEARL–System Definition: `sta2: BU(adr2,2);`

(Transfer-) Assembler Realisierung:

```
>sta2   BRA.S   IN        Einsprung für Einlesen
        _MOVE   D0,adr2   Zugriff auf Peripherie unter adr
         XRTS             Rücksprung (by stack or linkreg)
IN       CLR.L  D0        löschen
        _MOVE   adr1,D0   Wert einlesen, no condition code
         XRTS             Rücksprung (by stack or linkreg)
```

PEARL-System Definition: `sta3: BU(adr3,3);`

(Transfer-) Assembler Realisierung:

```
>sta3   BRA.S   IN        Einsprung für Einlesen
         SWAP   D0        Wort tauschen
        _MOVE.L D0,adr3   Zugriff auf Peripherie unter adr
         XRTS             Rücksprung (by stack or linkreg)
IN      _MOVE.L adr3,D0   Wert einlesen
        _SWAP   D0        Wort tauschen
         RTS              Rücksprung (by stack or linkreg)
```

PEARL-System Definition: `sta4: BU(adr4,4);`

Assembler Realisierung (nur 68K-Prozessoren, wegen `MOVEP`):

```
>sta4   BRA.S    IN         Einsprung für Einlesen
        LEA      adr4,A0    Adresse laden
        MOVEP.W  D0,0(A0)   Zugriff auf Peripherie
        RTS                 Rücksprung
IN      LEA      adr4,A0    Adresse laden
        CLR.L    D0         löschen
        MOVEP.W  0(A0),D0   Wert einlesen
        RTS                 Rücksprung
```

PEARL-System Definition: `sta5: BU(adr5,5);`

Assembler Realisierung (nur 68K-Prozessoren, wegen `MOVEP`):

```
>sta5   BRA.S     IN          Einsprung für Einlesen
        LEA       adr5,A0     Adresse laden
        SWAP      D0          Wort tauschen
        MOVEP.L   D0,0(A0)    Zugriff auf Peripherie
        RTS                   Rücksprung
IN      LEA       adr5,A0     Adresse laden
        MOVEP.L   0(A0),D0    Wert einlesen
        SWAP      D0          Wort tauschen
        RTS                   Rücksprung
```

PEARL-System Definition: `sta6: BU(adr6,6);`

(Transfer-) Assembler Realisierung:

```
>sta6    BRA.S    IN          Einsprung für Einlesen
        _MOVE.B   D0,adr6     Zugriff auf Peripherie unter adr
         XRTS                 Rücksprung (by stack or linkreg)
IN       CLR.L    D0          löschen
        _MOVE.B   adr1,D0     Wert einlesen
         XRTS                 Rücksprung (by stack or linkreg)
```

PEARL-System Definition: `sta7: BU(adr7,8);`

Assembler Realisierung (nur bei 68K-Prozessoren):

```
>sta7   BRA.S     IN          Einsprung für Einlesen
        LSR.W     =8,D0       rechtsschieben um ein Byte
        ANDI.B    =$0080,D0   maskieren
        MOVE.B    D0,adr7     Zugriff auf Peripherie
        RTS                   Rücksprung
IN      TAS       adr7        testen und ggf. setzen
        SNE       D0          Ergebnis übertragen
        ANDI.B    =$80,D0     maskieren
        LSL.W     =8,D0       in high-Byte schieben
        RTS                   Rücksprung
```

Beispiel: Es soll ein Wort–Zugriff auf die Adresse `$FE0000` im Supervisor–Mode des Prozessors stattfinden. Dies entspricht der Zugriffsart 2. Die Datenstation soll den logischen Namen `DIGO` bekommen. Wir codieren die Station in Transferassemblersprache wie folgt:

```
OFF      OPD     $4E4F          Trap-Definition
DPC      OPD     $4E43
         DC.L    0,0            Modul-Kopf
         DC      $0010          Type = Module
         DC.B    'Dation'       Name = Dation
>DIGO    BRA.S   IN             Einsprung für Lesen
         XSL                    save return link
         OFF                    In Supervisor
         MOVE    D0,$FE0000     Daten schreiben
         DPC                    Dispatcher Start
         RTS                    Rücksprung
IN       XSL                    save return link
         OFF                    In Supervisor
         MOVE    $FE0000,D0     Daten lesen
         DPC                    Dispatcher Start
         RTS                    Rücksprung
         END                    Ende
```

Der `XSL`-Befehl ist bei den 68K-Prozessoren ein Leerbefehl, d.h. es wird nichts generiert. Bei den RISC-Prozessoren sichert er die Rückkehr-adresse auf den Stack. Dies ist notwendig, weil in der obigen Sequenz Systemtraps aufgerufen werden, die bei den RISC-Prozessoren das Linkregister zerstören.

In einem PEARL–Programm könnte die neue Datenstation jetzt wie folgt benutzt werden:

```
PROBLEM;
   SPC DIGO DATION INOUT BASIC GLOBAL;
   ...
   TAKE wert FROM DIGO;
   ...
```

Das übersetzte PEARL-Modul muß nun nur noch mit dem assemblierten (bzw. transferassemblierten) obigen Maschinenprogramm gelinkt werden.

Hinweis:

Die Kodierung von Supervisormodesequenzen ist ein sicherheitsrelevanter Bereich und sollte nur im Notfall und mit äußerster Vorsicht erfolgen! So darf keinesfalls der `DPC` vergessen werden, da sonst eine Rücksprungadresse vom System–Stack des Prozessors geholt wird, die aber nicht dort, sondern auf dem Userstack abgelegt wurde. U. u. kann es zu einem Systemabsturz kommen. Die Zeit zwischem dem `OFF` und dem `DPC` sollte nicht länger als wenige μsec betragen, da sonst die Echtzeiteigenschaft des ganzen Systemes merkbar leidet. U.a. könnten Einplanungen verschlafen werden, auch der Verlust hochfrequenter Interrupts wäre sonst möglich!

6.4 Datenstation D*x*

Die Datenstation `/Dx` ist im System eingerichtet, um mit den Schnittstellen einen Voll–Duplex–Betrieb fahren zu können. Sie stellen einen zusätzlichen Ausgabekanal für die seriellen Schnittstellen dar. Bisher wurde mit einer nicht erfüllten Eingabeanforderung die Warteschlange der seriellen Schnittstelle blockiert, und es konnten keine Ausgaben auf die Schnittstelle gemacht werden, solange die Eingabe nicht erfüllt wurde. Da für die Datenstation `/Dx` eine getrennte Warteschlange genutzt wird, kann ihre Ausgabe jederzeit bearbeitet werden.

Die `/Dx` Datenstationen sind nur für die Ausgabe vorgesehen, eine Eingabe wird mit einer Fehlermeldung (`WRONG I/O`) quittiert.

Bedieninterface:

Stationsname ist `/Dx` z. B. `/D1`, `/D2`. Eine dem Systemnamen zugefügte Pathlist wird in die Verwaltung übernommen (`S`–Befehl), hat aber keine funktionelle Bedeutung.

Beispiel: `COPY /HO/bla>/D2`

PEARL–Programm:

Der Systemname entspricht dem des Bedieninterfaces.

Beispiel:

```
SYSTEM;
   TYdup: /D1 ->; TY:/A1;
   ...
PROBLEM;
   SPC TYdup DATION   OUT ALPHIC;
   SPC TY    DATION INOUT ALPHIC;
   ...
   ALARM: TASK;
      PUT 'Alarm' TO TYdup;
   END;
   ...
   INPUT: TASK;
      DCL in CHAR(1);
      GET in FROM TY BY SKIP,A;
   END;
```

In diesem Beispiel wird die Funktion der Datenstation `/Dx` verdeutlicht. Trotz nicht erfüllter Eingabeanforderung der Task `INPUT`, kann die Task `ALARM` ihre Alarmmeldung auf die Schnittstelle ausgeben.

6.5 Datenstationen ED/EDB

Zu dieser Station gehört die Betreuungstask `#EDFMN`. Damit können — solange genügend Speicher vorhanden ist — beliebig viele benannte Textdateien ähnlich wie auf einem Massenspeicher verwaltet werden. Die Betreuungstask `#EDFMN` kennt keine Wartephasen, da ja kein physikalisches Gerät in dem Transport verwickelt ist.

Der Text wird im Speicher in untereinander vernetzten Blöcken (`EDTF`) abgelegt. Es wird, soweit möglich, eine Verdichtung durch eine Sonderform des „Run-length-Encoding“ durchgeführt und eine Zeilennumerierung mitgeführt. Bei binären Daten ist die Verkürzung allerdings meist eher bescheiden. Die Station `/EDB` unterscheidet sich heute nicht mehr wirklich von `/ED`, lediglich die „Laufwerksnummer“ ist „1“ statt „0“.

Man beachte, daß man binäre Daten nicht mit dem Bedienbefehl `COPY` zwischen Platte/Floppy und `/ED` oder `/EDB` kopieren kann. Dieser Befehl bereitet den File für den Editor auf und stoppt das Kopieren beim ersten EOT-Zeichen. Dagegen ist die Ablage von binären Daten aus PEARL- oder Assemblerprogrammen heraus möglich.

Der in den `/ED`–Dateien abgelegte Text kann durch Einloggen in den bildschirmorientierten Texteditor (siehe `ED`–Kommando) verändert werden – natürlich nicht, wenn er binäre Daten enthält.

Erlaubte Operationen sind:

`DIR`, `ERASE`, `FILES`,`FIND`, `READ`, `REWIND`, `RM`, `SAVEP`, `SEEK`, `TOUCH`, `WRITE`.

Bedieninterface:

Name der Station ist `/ED` oder `/EDB`. Die Angabe eines Filenamen (bis zur implementierungsabhängigen maximalen Länge, meist 64 oder mehr Zeichen) ist notwendig, da andernfalls der Ersatzname „–“ eingesetzt wird. Nur bei `FILES` oder `DIR` kann auf einen Filenamen verzichtet werden.

Beispiel:
```
ERASE /ED/Test /ED/Kopie
REWIND /ED/Data
TOUCH -R /ED/Test
COPY ... > /ED/Program
```

PEARL–Programm:

Systemname ist `/ED` oder `/EDB`, wenn man die binäre Benutzung damit dokumentieren möchte. Wird kein Filename zugefügt, so wird der Filename „`-`“ eingesetzt.

Beispiel 1: Station `/ED` normale ASCII–Zeichen

```
MODULE M;
   SYSTEM;
      POOL: /ED/SAVE ->;
      ...
   PROBLEM;
      SPC POOL DATION OUT ALPHIC;
      ...

   AA: TASK;
      ...
      OPEN POOL;
      CALL REWIND (POOL);
      PUT x,x**2 TO POOL BY (2)E(20,10);
      ...
      CLOSE POOL;
      ...
   END;
MODEND;
```

Beispiel 2: Station /EDB binäre Daten

```
MODULE M;
   SYSTEM;
      DATA: /EDB/BIN ->;
      ...
   PROBLEM;
      SPC DATA DATION OUT ALPHIC CONTROL(ALL);
      SPC WRITE ENTRY GLOBAL;
      ...
   AB: TASK;
      DCL SSS(1000) FLOAT;

      OPEN DATA;
      CALL REWIND(DATA);
      /* Daten binaer speichern */
      CALL WRITE (DATA,SSS);
      ...
      CLOSE DATA;
   END;
MODEND;
```

Hinweise

Wird nach einer REWIND–Operation erneut in den File geschrieben, so wird die Länge des Files bis auf den Stand des Schreibzeigers gekürzt, ggf. früher eingeschriebene Daten (Text) gehen somit verloren.
Dies gilt jedoch nicht, wenn der File mit SEEK vorher positioniert wurde. In diesem Fall ersetzt der /ED-Handler die Zeichen genauso, wie es ein Handler für die Platte tut.

Falls bei einer Schreiboperation kein Platz für die Anforderung eines weiteren Blockes mehr zur Verfügung steht, so wird eine Fehlermeldung und der Returncode RECLEN=0 (wichtig für Assemblerprogrammierer) abgesetzt. Die zu schreibenden Daten gehen dabei verloren.

Die Datenstationen /ED und /EDB verfügen über einen „auto-close“-Mechanismus. D. h. nach jedem Filezugriff, außer bei einer „Exclusiv-Öffnung“, wird die angesprochene Datei selbstständig geschlossen, so daß praktisch keine geöffneten Dateien zurückbleiben können.

6.6 Datenstationen F*x*/H*x*

Der Fileaufbau ist hierarchisch, es besteht aber kein Zwang zur Benutzung einer solchen Baumstruktur. Insbesondere können uralte Disketten des nichthierarchischen Filehandlers immer noch einwandfrei gelesen werden.

Die Platte kann scheinbar in verschiedene Laufwerke aufgeteilt sein (Partitionierung). Zu diesen scheinbaren Laufwerken gehören dann auch eigene Directories. Grundsätzlich gibt es keinen Unterschied beim Umgang zwischen Disketten und Festplatten.

Studieren Sie bitte die Kommandos `FORM`, `FILES`, `FREE`, `CF` (evtl. auch für Festplatte), `SYNC`, `REWIND`, `RM/ERASE`, `MKDIR`, `RMDIR`, `RETURN`, `MSFILES`, `RTOSFILES` und `DIR`.

Warnung

> Wichtig ist, daß nicht versehentlich Files geöffnet zurückbleiben oder bei geöffneten Files die Diskette gewechselt wird, weil das den Verlust der gesamten Daten nach sich ziehen kann!!! Dies wird u. U. erst später nach außen sichtbar.

Unsere Floppy- und Plattenhandler erlauben einen wahlfreien Zugriff von PEARL–Programmen aus mit Hilfe der Einbaufunktionen `SEEK` und `SAVEP`.

Wird bei einer Datei bis zum Dateiende gelesen, schließt der Filemanager diese Datei automatisch. Wird danach noch ein `CLOSE` versucht, ist die Datei schon geschlossen, und es gibt eine Fehlermeldung. Wenn das automatische Schließen der Datei stört, da in jedem Fall noch eine `CLOSE`–Operation durchgeführt werden soll, kann es unterdrückt werden: Im `SYSTEM`–Teil des PEARL–Programms muß mit Hilfe des `AI/MB`–Parameters das Suppress–Command–Bit (= `$0400`) gesetzt werden.

Mit dem Kommando `MSFILES` kann auf eine DOS–kompatible Dateiverwaltung umgeschaltet werden. Am Zugriff auf die Diskette oder Festplatte ändert sich nach außen nichts, Sie können also weiterhin `DIR /F0` eingeben, obwohl im entsprechenden Laufwerk eine DOS–Diskette steckt. Der DOS–kompatible Filemanager unterstützt alle Features des **RTOS–UH**–Filemanagers. Zum Formatieren von DOS–Disketten ist das Format `C5` (9 Sektoren je 512 Bytes) einzugeben.

Sollen zwischen **RTOS–UH** und DOS Dateien kopiert werden, so ist zu beachten, daß unter **RTOS–UH** ein Record mit CR, unter DOS hingegen mit

CR/LF endet.

Bedieninterface:

Stationsname ist `/Fx` (x=0, 1, ...) für die Disketten und `/Hx` (x=0, 1, ...) für die Festplatte.

Beispiele: `LOAD /H0/usr/games/kalaha`
`SYNC /F1; CF /F1/FORGET;` (bei eiligem Aufbruch)

PEARL–Programm:

Der PEARL–Compiler übergibt diese Geräte als Extra-Devices an den Lader bzw. Linker.

Beispiel:
```
SYSTEM;
   Wfile1:/H0/platzhalter12345<->;
   ...
PROBLEM;
   SPC Wfile1 DATION INOUT ALPHIC;
   ...

   OPEN Wfile1 BY IDF('mueller/daten');
   CALL REWIND(Wfile1);
   PUT x,y TO Wfile1;
   CALL SAVEP(Wfile1,Pos);
   ...
```

6.7 Stationszugriff über „LD“

Wenn eine Station keinen mnemotechnischen Namen besitzt, so kann sie über das Bedieninterface durch Angabe ihrer Warteschlangennummer (LDN) adressiert werden. Es kann ebenso wie beim PEARL–Compiler sowohl LDN als auch das DRIVE eingegeben werden.

Beispiel: COPY /ED/test > LD/5.3/abc/xyz

PEARL–Programm:

Die Station wird über das Schlüsselwort LD angesprochen. Dahinter erfolgt, durch / abgetrennt, die Angabe von Warteschlangen– und Drivenumber. Ein Beispiel:

```
SYSTEM;
   Flop2:LD/5.2/xxxxxxxxxx<->;/*LDN=5,DRIVE=2*/
PROBLEM;
   SPC Flop2 DATION INOUT ALPHIC CONTROL(ALL);
   ...
   PUT data1,data2 TO Flop2 BY (2) LIST;
```

Natürlich können über diesen Weg Warteschlangennummern angegeben werden, zu denen gar keine Betreuungstasks existieren (systemintern ist dann kein „Task–Identifier“ TID in der LDN--TID Tabelle eingetragen). In solchen Fällen meldet sich das System beim verantwortlichen User mit

>> *taskname*: WRONG LDN (XIO)

und es findet keine Operation statt.

6.8 Datenstation NIL

Die Datenstation `/NIL` ist die ideale Datensenke und Datenquelle. Sie kann behilflich sein bei einem funktionellen Test eines PEARL–Programmes, um den Ablauf eines Programmes zu testen.

Alle Eingaben von der Datenstation `/NIL` werden mit einem CR (ASCII `$0D`) beanwortet. Somit können alle Eingabeanforderungen eines PEARL–Programmes erfüllt werden, da ein CR von allen Eingabeformaten akzeptiert wird. Alle Ausgaben auf die Datenstation `/NIL` werden verschrottet, so daß sie eine ideale Datensenke darstellt.

Soll zum Beispiel der Datenstrom von `/B2` verschrottet werde, so gebe man ein:

```
COPY /B2>/NIL
```

Will man einen unendlichen Strom von Carriage-Returns auf die Schnittstelle `/B2` senden, so gelingt das mit

```
COPY /NIL>/B2
```

PEARL–Programm

Systemname ist `/NIL`. Eine dem Systemnamen zugefügte Pathlist wird in die Verwaltung übernommen (`S`–Befehl), hat aber keine funktionelle Bedeutung.

Beispiel:

```
SYSTEM;
   dummy: /NIL<->;
PROBLEM;
   ...
   SPC dummy DATION INOUT ALPHIC CONTROL(ALL);
   ...
   ALARM: TASK;
      PUT 'Alarm' TO dummy;
   END;
   ....
   INPUT: TASK;
      DCL in CHAR(1);
      GET in FROM dummy BY SKIP,A;
   END;
```

In diesem Beispiel wird die Funktion der Datenstation `/NIL` verdeutlicht. Beide Task laufen bis zu ihrem `END` durch, da die Datenstation `/NIL` die Ein–/Ausgaben der Tasks befriedigt.

6.9 Parallel–Port

Die parallele Druckerschnittstelle des Rechners wird angesprochen, und die Daten werden im Handshakemode interruptgesteuert übertragen. Die Druckerbereitschaft kann geprüft werden, wenn ein `PUT` mit `WAIT` generiert wird. In diesem Fall erhält man eine Fehlermeldung oder kann über `ST` den Status abfragen. Andernfalls, wenn also kein Gerät angeschlossen oder dieses nicht bereit ist, bleibt die Betreuungstask `#PPORT` so lange hängen, bis ein Data–Acknowledge empfangen wird.

Bedieninterface:

Name der Station für den normalen Texttransfer ist `/PP`. Ein evtl. angegebener Filename bleibt ohne Wirkung. Das Gerät ist dem System nur als Ausgabeeinheit bekannt. Mit Hilfe des `SD`–Befehles kann das Auto–Linefeed den Erfordernissen entsprechend parametriert werden.

Zur Übertragun binärer Daten, etwa zur Ausgabe von Grafikbildern, ist in vielen Systemen noch der Stationsname `/PN` installiert. Hier unterbleibt eine Veränderung des Datenstromes durch Anfügen von Line–feeds etc.

Beispiel: `COPY /ED/Test>/PP/`

PEARL–Programm:

Der Systemname ist `/PP`. Dazu ein Beispiel:

```
SYSTEM;
   PRINT: /PP ->;
   ...
PROBLEM;
   SPC Print DATION OUT ALPHIC CONTROL(ALL);
   ...
   PUT message TO Print BY ...;
```

6.10 Datenstationen VI, VO

Die Stationen `/VI` (Virtual Input) und `/VO` (Virtual Output) besitzen jeweils eigene prioritätengeordnete Warteschlangen, aber nur eine Betreuungstask `#VDATN`, die deren Betreuung übernimmt. Alle Ausgabe–Communication–Elements (CE), die nach `/VO` geschrieben werden, sowie alle Eingabe-CE's, die von `/VI` gelesen werden, werden in eine betreuungstask–eigene Warteschlange übernommen. Die Übertragung der Daten von den `/VO`–CE's in die `/VI`–CE's erfolgt, sobald auf beiden Seiten CE's vorhanden sind. Die Stationen arbeiten nach dem FIFO–(First In/First Out) Prinzip, d. h. die zuerst erfolgte Ausgabe an eine `/VO`–Datei wird als erste Eingabe von der gleichen `/VI`–Datei gelesen.

`/VI` und `/VO` bilden ein Instrument zur synchronisierten Task-Kommunikation. Erzeuger können ihre Daten zu einem beliebigen Zeitpunkt in die Datenstation `/VO` schreiben und ohne Unterbrechung weiterarbeiten; auch eine Terminierung des Erzeugers führt nicht zum Datenverlust. Verbraucher können jederzeit Daten von `/VI` anfordern, werden aber bei gesetztem Waitbit (das ist die Regel) bis zur Erfüllung der Eingabeanforderung angehalten.

Z. B. können Compiler und Lader über `/VI` und `/VO` verbunden werden, um das Compilat nicht als (speicher– oder floppy–) residente Datei abzulegen.

Bedieninterface:

Die Stationen heißen `/VI` und `/VO`, der Dateiname kann einen Path enthalten. Es können beliebig viele Dateinamen gleichzeitig verwendet werden. Dazu 2 Beispiele:

```
LOAD /VI/Loader; PEARL ...>/VO/Loader
RM /VO/Mist (wenn /VO/Mist überflüssig ist)
```

PEARL–Programm:

Systemnamen sind `/VI` (nur Eingabe) und `/VO` (nur Ausgabe). Wird kein Filename angegeben, so wird der am weitesten rechts stehende Nutzername verwendet; wird kein Nutzername angegeben, so wird der Filename „–“ eingesetzt.

Ein Beispielprogramm:

```
SYSTEM;
   PRODUCE: /VO/mypipe ->;
   CONSUME: /VI/mypipe <-;
PROBLEM;
   SPC PRODUCE DATION OUT ALPHIC CONTROL(ALL);
   SPC CONSUME DATION IN ALPHIC CONTROL(ALL);
   T1: TASK;
      ...
      PUT ... TO PRODUCE BY ...;
      ...
   END;
   T2: TASK;
      ...
      GET ... FROM CONSUME BY ...;
      ...
   END;
```

6.11 Datenstation XC

Zu dieser Station gehört die Task `#XCMMD`, die die Betreuung der Ausgabeschlange (prioritätsgeordnet) übernimmt. Mit dem an dieser Station eintreffenden Text wird das Bedieninterface beschickt, die Kommandos werden also auf der Prioritätsebene der Task `#XCMMD` ausgeführt, sofern nicht Subtasks gebildet werden. Die an der Station eintreffenden Sätze sollten jeweils mit CR oder LF abgeschlossen werden. Leere Sätze haben keine Wirkung.

Bedieninterface:

Name der Station ist `/XC`. Ein hinzugefügter Filename wird zwar in die Verwaltung übernommen und erscheint beim `S`–Befehl des Bedieninterfaces, er nimmt aber auf den Ablauf keinen Einfluß.

Beispiel: `COPY ...>/XC/`

PEARL–Programm:

Systemname ist `/XC`. Wird kein Filename zugefügt, so wird der Filename „`-`“ eingesetzt. Wie beim Bedieninterface hat der Filename jedoch keine Wirkung auf den Ablauf. Ein Beispiel:

```
SYSTEM;
   BEDIEN: XC;
   ...
PROBLEM;
   SPC BEDIEN DATION OUT ALPHIC CONTROL(ALL);
   ...
   TT: TASK;
      ...
      PUT 'UNLOAD TEST' TO BEDIEN BY A,SKIP;
      ...
```

Zum Thema „ausführen von Bedienbefehlen“ siehe auch die Anmerkungen zur Shellsprachanweisung `EXEC` auf Seite 90 oder die Erläuterungen zum PEARL-Unterprogramm `CMD_EXW` im Abschnitt 5.7.16 auf Seite 350.

6.12 Prozeßinterrupts

RTOS–UH verwaltet maximal 32 Prozessinterrupts, die eine hohe zeitliche Auflösung ermöglichen, da die Interruptroutine typischerweise in weniger als 100μs durchlaufen werden kann. Für einige Implementierungen von **RTOS–UH** gibt es fest zugeordnete Leitungen zu einzelnen Bits des Interruptbitmusters. Diese Zuordnungen entnehme man den Hardwareunterlagen.

Ein Interrupt wird in **RTOS–UH** nur wirksam, wenn sein zugehöriges Bit in der „ENABLE“–Maske des Systemes gesetzt ist. Bei einem Kaltstart von **RTOS–UH** werden zunächst alle Prozessinterrupts abgeschaltet.

Bedieninterface:

Siehe Anweisungen ENABLE, DISABLE, TRIGGER und WHEN.

PEARL–Programm:

Systemname ist EV(*hexnum8*). Das Bitmuster *hexnum8* gibt an, auf welche Bits reagiert werden soll. Ein Beispiel:

```
SYSTEM;
   Limit: EV(00000006);
   ...
PROBLEM;
   SPC Limit INTERRUPT;
   ...
   TS: TASK;
      ...
      WHEN Limit ACTIVATE XYZ;
```

Sowohl das Interruptbit 00000002 als auch das Bit 00000004 führen jetzt zur Aktivierung von XYZ (logisches 'oder').

Tip

Mit Hilfe der TRIGGER–Anweisung kann die Wirkung eines äußeren Ereignisses exakt simuliert werden, da systemintern die gleichen Programmteile angestoßen werden.

6.13 Einbindung eigener Prozeßinterrupts

Der Nutzer schreibt eine Interruptroutine und versorgt den Interruptvektor sowie die sogenannte „Malfunction“ (s. dazu Extrabeschreibung ab Seite 611) mit Hilfe einer Task oder des GO–Befehles. Dabei müssen genau die angegebenen Register gerettet werden, da sonst der Rückfallmechanismus zum Absturz führen kann.

Der Anschluß könnte bei einem **68k-Prozessor** etwa wie folgt aussehen:

```
      DC.L   0,0        Fuer RTOS-Lader         !   *
      DC     $0010      Modulkopf   ''              *
      DC.B   'Prozir'   Name des Modules  ''        *
*-------------------------------------------------*
IID   EQU    $7FE       Interrupt-identifier        *
IRVEC EQU    ????       Vector-link je nach Hardware*
TERV  OPD    $A010      Terminate and vanish        *
*.... Hier Einstieg fuer 'GO'-Befehl ..............*
      MOVE.L =IRP,IRVEC Einsetzen                   *
      TERV              'GOTO'-Subtask killen.      *
*-------------------------------------------------*
      DC     IRPE-IRP   Anschluss zu Fehlerrueckfall*
IRP   MOVE   IID,-(A7)  Save old Interruptidentifier*
      MOVE   =IRVEC,IID For any malfunction-process *
      MOVEM.L D1/D6/D7/A1,-(A7) ist vorgeschrieben!!*
*                                                   *
*      Interruptbitmuster vom Coupler lesen         *
*      und den Interruptrequest des Couplers        *
*      zuruecksetzen (IR-Ursache beseitigen).       *
*      Triggerbitmuster nach D1.L schaffen.         *
*                                                   *
      MOVE.L =$00001000,D1 Bitmuster fuer EV        *
      MOVEA.L $80E,A1   Zieladresse innerhalb RTOS  *
      JMP    (A1)       Zum Systemkern              *
*.... Malfunction ..................................*
IRPE  MOVEM.L (A7)+,D1/D6/D7/A1 Reload registers    *
      MOVE   (A7)+,IID  Rueckladen des Interrupt-ID *
      RTE               Rueuckfallabschluss         *
```

Durch einmalige Exekution von GO auf den Platz (Ladeadresse+$10) wird die neue Prozessinterruptbehandlung aktiviert. Dies ist die platzsparendste Lösung, bequemer wäre es, eine richtige Anschlußtask zu schreiben und die-

se namentlich zu starten.

Hinweis

Nach Abort/Reset wird dieser Anschluß wieder aufgehoben, es muß also ein erneuter `GO`–Befehl abgesetzt werden.

Vor dem Entladen des angeschlossenen Interruptcodes muß umgekehrt das alte Link wieder hergestellt werden, eben z. B. mit Abort. Für dauerhafte Sonderbehandlung wird eine Einbettung in den EPROM–Code empfohlen. Siehe dazu Beschreibung der Scheibe 14, Seite 654. In diesem Fall ändert sich am Code der Interruptroutine nichts, lediglich der Vektoranschluß wird durch die Scheibe automatisiert.

Bei den RISC-Versionen von **RTOS**–**UH** (PowerPC) sieht der Mechanismus in der Struktur ganz ähnlich aus. Dort gibt es jedoch keine Vektorinterrupts. Außerdem ist die Hardwareumgebung vor dem einen (!) Interrupteingang des Prozessors je nach Prozessorboard sehr unterschiedlich. Die Vektorinterrupts der 68k-Welt werden in einem Implementierungsmodul nachgebildet. Eine universelle Darstellung an dieser Stelle ist zur Zeit wenig sinnvoll, bitte informieren Sie sich zu gegebener Zeit bei uns.

Kapitel 7: Der RTOS–UH Assembler

7.1 Allgemeine Eigenschaften

RTOS–UH verwendet seit der Integration der RISC-Prozessoren (PowerPC) eine prozessorunabhängige Maschinensprache, der wir den Namen „T-Code“ gegeben haben. Niemand sollte über diese neue Philosophie erschrecken: Bisherige 68k-Maschinenprogramme können natürlich in der 68k-Version unverändert weiterbenutzt werden! Erst wenn man diese 68k-Programme auf den PowerPC portieren möchte, sollte man sich mit dem T-Code beschäftigen. Das ist kein großer Aufwand, denn der T-Code unterscheidet sich nur minimal von der 68k-Assemblersprache. Oft kann man komplette 68k-Programme sogar ohne Änderungen als T-Code verwenden.

Zur Zeit existieren für den T-Code 2 Übersetzer:

- Der T-Code-Übersetzer für den 68k ist durch kleine Erweiterungen aus dem 68k-Assembler entstanden und ersetzt diesen vollständig. Der alten Gewohnheit folgend nennen wir ihn weiterhin „68k-Assembler“.
- Der T-Code-Übersetzer für den PowerPC ist ein neues Produkt, ein echter „Transferassembler“. Der zugehörige Bedienbefehl ist `TAPP`, ein Acronym für **T**ransfer-**A**ssembler **P**ower**P**C.

Alle Assembler sind virtuell codiert, sie verhalten sich daher unabhängig von der Gastmaschine in allen Implementierungen in gleicher Art und Weise. Den 68k-Assembler gibt es in zwei unterschiedlich leistungsfähigen Varianten: Die „MAXI“- ist gegenüber der „MINI“-Version um die Befehle des 68020/68881 erweitert. Der Transferassembler TAPP entspricht bezüglich Befehlsumfang und Adressierungsarten der Mini-Version.

Ausgegeben werden S–Records in einem gegenüber dem Original (Motorola) erweiterten Format. Die Ausgabe ist abgestimmt auf das PEARL–System **RTOS–UH** und dort bindefähig. Bei entsprechender Selbstbeschränkung (keine relativen Langadressen, keine globalen Symbole) ist jedoch Motorola–Kompatibilität erreichbar.

Bei der Codierung richte man sich nach dem Hardwarereference Manual von Motorola für den 68k. Man beachte dabei, daß die Immediateadressierung durch „`=`“ statt „`#`“ spezifiziert werden muß.

Der Assembler benötigt 2 Durchläufe. Das Quellprogramm muß zweimal angeboten werden, wenn die Quelldatei nicht „rückspulbar“ ist, es sei denn, der Assembler arbeitet im „automatic scratch–pad“–Mode, bei dem er selbst eine Zwischendatei anfertigt. Die Betriebsparameter werden über das Bedieninterface besetzt:

`CO /Dev/File` Senke für die S–Records.

`LO /Dev/File` Senke für die Liste mit Zeilennummer und Hexcode.
Auch bei `LO NO` werden die fehlerhaften Zeilen aufgelistet.

`SC /Dev/File` Scratch–Device. Ist die Input–Datei rückspulbar, so muß `SC` nur angegeben werden, wenn eine neue Quellendatei erstellt werden soll. Wenn `SC` angegeben ist, so wird es in jedem Fall benutzt. Bei `SC NO` muß bei nicht rückspulbaren Dateien der Quellcode zweimal angeboten werden.

`SI /DEV/FILE` Eingabe–Datei.

Bei Betrieb unter **RTOS–UH** kann zusätzlich noch der Speicherbereich und/-oder die Bearbeitungspriorität angegeben werden.

7.2 Programmzeilenaufbau

Jede Zeile enthält entweder eine Anweisung an den Assembler oder beschreibt einen Maschinenbefehl. Sie zerfällt in 4 Sektionen:

Labelfeld Operationsfeld Operandenfeld Kommentarfeld.

Mehrere Anweisungen pro Zeile sind nicht zugelassen. Als Feldtrennung wird die Lücke (mind. 1 Blank), bei Befehlen, die mit und ohne Operand auftreten können, ein Block $>$ 10 Blanks benutzt.

7.2.1 Labelfeld

Das Feld ist leer (mind. 1 Blank) oder enthält ein max. 24 Zeichen langes „Labelsymbol“. Solche „Labels“ können auf vier Arten definiert werden:

1. In Spalte 1 bedeutet „∼“: Das folgende Symbol wird als globales PEARL90-Symbol definiert und ist dem Lader später in dieser Kategorie namentlich bekannt. Weitere Behandlung wie unter 3.
2. In Spalte 1 bedeutet „>“: Das folgende Symbol wird als allgemeines globales Symbol definiert und ist dem Lader später namentlich bekannt. In der alten PEARL80-Welt wurden auch vom Compiler generierte PEARL-Globals dieser Kategorie zugeordnet. Weitere Behandlung wie unter 3.

3. In Spalte 1 steht ein Buchstabe oder das Buchstabenersatzzeichen `#`. Es wird ein bis zu 24 Zeichen langes Symbol definiert und ihm der Wert des Location–Counter (`REL` oder `ABS`) zugewiesen. Danach darf ein `:` folgen, muß aber nicht — eine Feldtrennung genügt. Zwischen Groß- und Kleinschreibung wird bei den Symbolen unterschieden.

4. Spalte 1 enthält ein Leerzeichen. Das Symbol darf weiter hinten im Feld beginnen, aber es muß das Zeichen `:` folgen, damit erkennbar ist, daß kein OP–Code gemeint sein kann.

   Innerhalb des Symboles sind Ziffern erlaubt.

7.2.2 Operationsfeld

Im Operationsfeld wird zwischen Groß- und Kleinschreibung nicht unterschieden. Es enthält eine der folgenden Anweisungen:

1. Semikolon

   Die Zeile dient nur zur Definition eines Labels und enthält keinen Operationscode. Wegen des freien Anweisungsaufbaues würde bei einfachem Weglassen des Operationscodes sonst ein eventueller Kommentar als Befehl interpretiert. Das Semikolon ist nicht erforderlich, wenn die Zeile dahinter leer ist.

2. Bedingungsanweisung für Assembler

   Diese beginnen mit einem Punkt, dahinter folgt einer der Mnemos `IF`, `ELSE` oder `FIN`. Nur wenn der Ausdruck hinter `IF` zur *Assemblierzeit*(!) einen Wert ungleich Null ergibt, wird der folgende Text vom Assembler bearbeitet. Mit dem `ELSE` Zweig kann der komplementäre Fall kodiert werden.

   Mit der bedingten Assemblierung können zur Übersetzungszeit Sourcetextabschnitte ausgeblendet werden, dieses geschieht mit den angegebenen Schlüsselworten. Die ausgeblendeten Sourcetextabschnitten werden auch in der *LIST*–Ausgabe unterdrückt. Die auszulassende Länge des Sourcetextes ist nicht begrenzt.

   Syntax: `.IF` ***`expr.`***
   ... Anweisungsfolge 1
   `.ELSE`
   ... Anweisungsfolge 2
   `.FIN`
   ...

Die Expression wird zur Übersetzungszeit ausgewertet und in Abhängigkeit von ihrem Wert die entsprechende Anweisungsfolge bei der Übersetzung berücksichtigt. Falls der `.ELSE`–Teil leer ist, kann das Schlüsselwort `.ELSE` weggelassen werden.

expr. $= 0$ der `.ELSE`–Zweig wird bei der Übersetzung berücksichtigt.
expr. $\neq 0$ der `.IF`–Zweig wird bei der Übersetzung berücksichtigt.

2 Beispieltexte dazu:

```
        ...
source EQU 1
        ...
       .IF source
        MOVE.B (A0)+,D0
       .ELSE
        MOVE.B D0,(A0)+
       .FIN

        ...
S1      EQU 1
S2      EQU 1
        ...
       .IF S1-S2
        MOVE.B (A0),D0
       .FIN
        ...
```

Nur die Anweisung `MOVE.B (A0)+,D0` wird hier für die Codeerzeugung berücksichtigt, weil `source`=1 ist.

`MOVE.B (A0),D0` wird nicht berücksichtigt, da die Expression gleich 0 ist.

Eine Schachtelung dieser Strukturen ist zulässig. Am Ende des Programmes muß natürlich die Anzahl der `IF` mit der Anzahl der `FIN` übereinstimmen. Vorsicht: Liegt die `END`-Direktive im abgeschalteten Bereich, so bricht der Übersetzer mit „end of input-file“ ab!

3. Prozessorswitch

Um Anweisungen optimal an den jeweiligen Zielprozessor anpassen zu können, kam mit dem T-Code eine Umschaltemöglichkeit, die in etwa den Bedingungsanweisungen entspricht, jedoch als Argument spezielle Schlüsselworte verwendet. Es gibt folgende Umschalter:

`.IF_PROCTYPE` *processorname*
`.IF_TATYPE` *processorname*

Der Wirkungsbereich kann wie bei den normalen `IF` mit `.ELSE` untergliedert werden und endet wie diese mit dem `.FIN`.

Als *processorname* sind zur Zeit folgende Strings zugelassen:

`MPC601 oder MPC604`	identisch, ganze PowerPC-Familie
`M68K`	ganze 68xxx-Familie

`IF_PROCTYPE`:

Wenn beim `.IF_PROCTYPE` vom Transferassembler festgestellt wird, daß er für die fragliche Maschine *processorname* kodiert, so erfolgt eine Umschaltung in den „native mode“ des Zielprozessors mit der zu dessen Assemblersprache gehörender Syntax. Erst beim passenden `.ELSE` bzw. `.FIN` wird wieder in die T-Code-Sprache zurückgeschaltet. Stimmt der Prozessorname nicht mit der tatsächlichen Zielmaschine überein, so wird der `.IF_..-`Zweig ignoriert.

Mit dieser Option kann man hochoptimierten prozessorspezifischen Kode erzeugen. Sie wird an einigen Stellen des Systemkernes, etwa beim Prozessumschalter, benutzt. Ein Umschalten von der ersten Programmzeile bis zum Ende verwandelt z.B. den PowerPC-Transferassembler in einen reinen PowerPC-Assembler.

`IF_TATYPE`:

Wenn der ausführende Transferassembler für den angegebenen Zielprozessor kodiert, so wird das im `.IF-`Zweig Stehende transferassembliert, sonst wird es ignoriert. Eine Mode-Umschaltung findet nicht statt. Damit kann man gewissen Prozessoreigenheiten Rechnung tragen, dabei jedoch weiterhin in T-Code kodieren.

4. `INCLUDE`-Anweisung

   Mit einem Statement der beispielhaften Form

   ```
   .INCLUDE /HO/SOURCE/X1.AS
   ```

   kann ein Quellfile an Stelle der Zeile eingefügt werden. Es ist nur die Substitution kompletter Zeilen möglich. Hinsichtlich der Möglichkeiten beim Filezugriff wird hier auf die Beschreibung beim Compiler verwiesen. Lesen Sie dazu bitte ab Seite 288 nach. Da der gleiche Unterbau (VCP) auch beim Assembler benutzt wird, gelten alle Angaben (relative Positionierung etc.) einschließlich der Markierung der Zeilennummern auch beim Assembler.

5. Hardware–Instruktion

   Alle Mnemos der Motorola 68k Hardwarebeschreibung sind zugelassen. Eine Längenspezifikation (`.L`, `.B`, `.W`) ist nur hinter solchen Mnemos zugelassen, bei denen überhaupt eine Wahlfreiheit besteht. Fehlt die Längenangabe, so wird die dem Mnemo eingeprägte Länge — z. B. `.L` bei `LEA` — oder `.W` bei Wahlfreiheit substituiert. Bei dem Befehl `MOVE` muß daher im

Ausnahmefall `MOVE ...,USP` die Länge `MOVE.L ...,USP` explizit angegeben werden, da sonst ein Längenfehler diagnostiziert wird. Bei Sprungbefehlen (`BRA`, `BGE`, `BPL` etc.) ist die Angabe `.L`, `.S` oder `.B` möglich. Sie nimmt dem Assembler die Wahlfreiheit bei der Codierung dieser Befehle.

Alle Hardware–Befehle werden auf geraden Positionszählerstand gesetzt, so kann z. B. nach einem `DC.B` vom Assembler automatisch ein nicht besetztes Byte eingefügt werden, wenn der nachfolgende Befehl eine gerade Adresse verlangt.

6. Virtueller PEARL–Laufzeit–Befehl (Hyperprozessor)

   Diese Instruktionen haben den folgenden Aufbau

   ```
   V... OP1,OP2,OP3,OP4
   ```

   Dabei steht `...` für eine max. 3–stellige Dezimalzahl im Bereich 0...255. Es sind 0...4 Operanden möglich, deren Syntax z. T. von der der realen Befehle abweicht (s. u.). Gleiches gilt bei Verwendung eines mit `OPD.V` definierten virtuellen Benutzermnemos.

7. Assembler–Direktive

   Dies sind Anweisungen an den Assembler, zur Assemblierzeit etwas bestimmtes zu tun. Solche Anweisungen erzeugen in der Regel keinen ausführbaren Maschinencode. Die Direktiven finden Sie ab Seite 427 genauer beschrieben.

8. Aufruf eines vorher definierten Formates

   Statt der in den meisten Assemblern zu findenden Makrodefinition gibt es in den **RTOS–UH**-Assemblern eine sogenannte Format-definition. Formate beschreiben Bauschablonen für Bytestrings. Am Ort des Formataufrufes können bis zu 9 Parameter mitgegeben werden, die nicht als Text (wie bei den primitiveren Makros) sondern mit ihren numerischen Werten den Bau des Bytestrings steuern. Eine allgemein verfügbare Format-Bibliothek ermöglicht z.B. das Generieren von Prozedurköpfen von assemblercodierten PEARL90-Unterprogrammen usw. Die Kodierung der Formatdefinitionen wird auf Seite 432 beschrieben.

7.2.3 Operanden–Feld

Die Adressierungsarten werden kleinlich mit den Hardwaremöglichkeiten verglichen. Fehlerdiagnose: `MODE-ERROR`. Es gibt einige Besonderheiten bei den Adressierungsarten:

Zur Verwendung in Hyperprozessorbefehlen und zur besseren Lesbarkeit normaler Befehle wurden folgende Abkürzungen eingeführt:

Die Adressierungsart label(A4) ist ersetzbar durch label.T
Die Adressierungsart label(A5) ist ersetzbar durch label.X

Bei `MOVEM` ist eine Ordnung der Registerliste von `D0` in Richtung `A7` zwingend vorgeschrieben (Schutz vor Tippfehlern).

	Adr.Art	Syntax	Bemerkungen
	Absolut–Short	*AE*	Absolut Expression
	Absolut–Long	*AE*	Wenn *AE* in Pass 1 > $FFFF
x		*AE*`.L`	Wertunabhängig Long
x	Extern global	*EG*	Wie Abs.–Long, kein Ausdruck
x	Relativ–Long	*RE*`.L`	Rel. durch Lader. Wie *AE*`.L`
	Reg. direkt	*RG*	*RG*=Register–Symbol.
	Indirekt Reg.	`(`*AR*`)`	*AR*=Adreßregistersymbol.
	Predecrement	`-(`*AR*`)`	—" " —"
	Postincrement	`(`*AR*`)+`	—" " —"
	Ind.Reg+Disp.	*AE*`(`*AR*`)`	*AE* muß in 16 Bit passen
	Ind. A4+Disp.	*AE*`.T`	Task–Workspace–Adressierung
x	Ind. A5+Disp.	*AE*`.X`	Prozedur–Workspace–Adr.
v	Ind(Ind.A5+Disp)	*AE*`.Z`	32 Bit–end–adr. nur bei V!
	Disp+AR+Ind.	*AE*`(`*AR*`,`*RG*`)`	`.W`–Index, *AE* 8 Bit!
	"	*AE*`(`*AR*`,`*RG*`.W)`	—" " —"
	"	*AE*`(`*AR*`,`*RG*`.L)`	`.L`–Index, "
x	PC–Relativ	*RE*	*RE*=REL.Expression, s. u.
	PC–Rel+Ind.	*RE*`(`*RG*`)`	`.W`–Index
	"	*RE*`(`*RG*`.W)`	— " —
	"	*RE*`(`*RG*`.L)`	`.L`–Index
	Label	*AE*	Im ORG–Mode. `Bcc`, `BRA`, `DBcc`
	"	*RE*	Im RORG–Mode. " " "
	Immediate	tt =*AE*,*RE*	Achtung: „`=`" statt „`#`"!

Die mit „x" gekennzeichneten Adressierungen sind bei realen und bei virtuellen Befehlen zulässig. Die mit „v" gekennzeichnete Adressierung ist nur bei virtuellen (Hyperprozessor-) Befehlen erlaubt.

7.2.4 Ausdrücke

AE ist ein Ausdruck, dessen Wert lageunabhängig ist

EG ist ein Bezug auf ein extern definiertes Symbol:

EG := >*symbol* oder
EG := ∼*symbol* oder
EG := >*symbol*+*offset*. *EG* := ∼*symbol*+*offset*.

symbol repräsentiert das übliche max. 24 Zeichen lange Symbol.

offset ist ein nicht lageabhängiger Ausdruck, der zur Assemblierzeit berechnet werden kann. Er darf auch einen negativen Wert liefern.

Beispiel: `JMP >TEST+$200`

Im eigenen Programmodul definierte Globalsymbole können zwar, sollten aber nicht über diese Konstruktion angesprochen werden, um den Lader zu entlasten. Um Verwirrungen mit den globalen PEARL90-Symbolen (∼-Zeichen am Anfang) zu vermeiden, empfiehlt sich die Verwendung von internen Aliasnamen.

RE ist ein Ausdruck, dessen Wert von der Lage des Programmes abhängig ist, wobei im Adressausdruck die spätere Ladeadresse nur einfach additiv eingehen darf. Damit ist z.B. X+Y verboten, wenn beide Symbole relativ positionierte Objekte sind.

In beiden Typen von Ausdrücken sind Klammerungen erlaubt, ebenso die Voranstellung monadischer Operatoren (`+` oder `-`). Ausdrücke werden grundsätzlich mit 32–Bit Arithmetik berechnet und erst in Pass 2 auf Einhaltung der zulässigen Grenzen überprüft.

Bei *AE* in Immediate und `DC`–Anwendungen wird zusätzlich zwischen „logisch“ und „arithmetisch“ unterschieden. Ein *AE* ist „logisch“, wenn in ihm mindestens eine Sedezimalzahl auftritt. Eventuell muß man also die Zahl `$0` addieren. Damit ist z. B. `40126+$0` auch bei Beschränkung auf 16 Bit legal, da „logisch“.

Elemente in Ausdrücken:

Dezimalzahlen	24,108637 (immer als AE)
Hexadezimalzahlen	$0,$AFFE,$2CDE3
Textstrings	'a','AB','xyz','Mist' liefern 8,16 oder 32 bit, bei Bedarf links mit Nullen aufgefüllt.
Symbol (kein Register!)	X,AB23 (Max. 24 Zeichen) Können RE oder AE sein.
Loc. counter = $	Ohne folgende Ziffern, ist RE im RORG–Mode, sonst AE.

Dyadische Operatoren:

+ Addition	$ABS+ABS = ABS$, $ABS+REL = REL$, $REL + ABS = REL$, $REL + REL$ nicht erlaubt!
− Subtraktion	$REL-REL = ABS$, $REL-ABS = REL$, $ABS - ABS = ABS$, $ABS - REL$ nicht erlaubt!
∗ Multiplikation	nur bei zwei AE erlaubt. Es wird in signed 32 Bit Arithmetik gerechnet.
/ Division	Wie bei Multiplikation, nur AE etc.

Die Prezedenz der Operatoren ist wie üblich, d. h. ∗, / geht vor +, −. Durch Klammerung wird die Prezedenz übersteuert.

7.2.5 Die Assemblerdirektiven

Etliche der oben bereits erwähnten Assemblerdirektiven benutzen Ausdrücke. Die Bedeutung von AE und RE in der folgenden Aufstellung wurde oben bereits erläutert.

`DC ...` Datenablage wie bei Maschinenbefehlen, aber bei `DC.B` keine Positionsrundung.

`DC` *`AE`*`,...` Ablage der 16–Bit Daten (Wert AE).

`DC.W` *`AE`*`,...` Wie oben. (`.W` ist Defaultbesetzung).

`DC.L` *`AE`*`,...` Ablage von 32 Bit Daten mit Wert AE.

`DC.L` *`EG`*`,...` Adresse des externen globalen Symbol ablegen.

`DC.B` *`AE`*`,...` Ablage der 8 Bit Daten ohne vorherige Positionsrundung.

	DC.B	*'str',..*	Ablage des Textstrings *'str'* mit fortlaufenden Adressen. Das Zeichen ' ist durch '' zu ersetzen. Bsp: DC.B 'AB''CD','''' → AB'CD' Mischung '*str*','*AE*',' str',... ist erlaubt.
	DS	*AE*	Define storage, mit *AE* angegebene Zahl von Bytes freihalten. Inhalt der Bytes nach dem Laden undefiniert.
	END		Ende dieser Assembliereinheit. Zwischen END und Kommentar mind. 10 Blanks!
	END	*RE*	Rückbezug für **RTOS–UH** herstellen.
LBL	EQU	*AE*	Definition von *LBL* mit Wert *AE* absolut. VORSICHT: Die Eigenschaft „logisch“ wird nicht durch das EQU übertragen.
LBL	EQU	*RE*	Definition von *LBL* mit Wert *RE* relativ.
LBL	EQU	*RG*	Definition von *LBL* als Registersymbol.
LBL	EQU	SR	Definition von *LBL* als Statusregister.
LBL	EQU	CCR	Definition von *LBL* als Condition–Coderegister.
LBL	EQU	USP	Definition von *LBL* als USER–A7.
MNE	FORMAT		Beginn Formatdefinition. Alle folgenden Zeilen, die mit „/“ beginnen, werden bis zur Endekennung zur Definition herangezogen. Der genauere Aufbau ist auf Seite 432 beschrieben.
	LOCK	*AE*	Sperren bestimmter Register für den Transferassembliervorgang, bei 68k-Zielmaschine ignoriert.
MNE	OPD	*AE*	Definition des Nutzer–Mnemos (ohne Operanden) *MNE* durch das 16 Bit–Wort *AE*.
MNE	OPD.V	*AE*	Definition des Nutzer–Mnemos *MNE* als virtueller Laufzeitbefehl. *AE* muß im Bereich 0...255 liegen, da sonst bei der späteren Benutzung ein *LIMIT*–Error erzeugt werden kann. Der so definierte Befehl kann 0...4 Operanden haben und und unterliegt der bei den V–Befehlen üblichen Syntax.
	ORG	*AE*	*AE* muß in Pass 1 definiert sein, der Wert wird in den Positionszähler geladen. Das Programm ist nun im ABS–Mode und kann vom **RTOS–UH**–Lader nicht korrekt geladen werden, sofern nicht ausschließlich relativ adressiert wird.

	`PAGE`		New Page. Es wird ein Seitenvorschub ausgegeben.
	`PRINT`	*AE*	Ist in Pass 2 der Wert von *AE* ungleich Null, so wird das Übersetzungsprotokoll ab dieser Zeile eingeschaltet, anderenfalls wird das Listing unterdrückt. Fehlermeldungen erscheinen aber auch dann noch
	`RORG`	*AE*	Wie bei `ORG` jedoch ist das Modul jetzt verschiebbar
	`RORG`	*RE*	und kann korrekt vom Lader geladen werden.
MNE	`UNLOCK`	*AE*	Hebt selektierte Registersperren für den Transferassembler auf. Gegenteil von `LOCK`. Die Selektion erfolgt mit Hilfe der Maske in *AE*.

7.3 Besonderheiten des T-Code

7.3.1 Problematische 68k-Befehle

Adressierungsarten:

Im T-Code sind nur die Adressierungsarten des 68000 erlaubt. Die erweiterten Möglichkeiten, die mit dem 68020 hinzugekommen sind, wurden nicht in den T-Code aufgenommen. Sie werden vom Mini-Assembler und vom PowerPC-Transferassembler als Fehler erkannt und angezeigt.

Es ist wahrscheinlich, daß PC-relative Bezüge, die in der 68k-Welt so eben noch mit einem 16-bit Verschiebungswort auskommen, auf einem RISC-Prozessor wegen der Codeverlängerung nicht mehr in 16 Bit passen. Der `.V`–Zusatz (V = Very far) hinter den Befehlen

`Bcc.V, BSR.V, LEA.V` und `PEA.V`

zwingt den Transferassembler, statt eines 16-bit langen Displacements – wie es in der 68k-Welt genügen würde – schon im Pass 1 Platz für ein 32-bit langes Displacement vorzusehen. Damit der Transferassembler nicht unnötigerweise den längeren Code erzeugt, sollte man nur die zuvor von ihm angemahnten Befehle und wacklige Kandidaten mit der Option versehen. Der 68k-Assembler ignoriert den `.V`-Anhang.

Maschinenbefehle:

Es sind nur die Maschinenbefehle des Nutzerprogrammiermodelles des 68000 erlaubt. Supervisorinstruktionen wie z.B. `RTE` werden von den reinen T-Code-Übersetzern als Fehler angezeigt und nicht umgesetzt.

Strukturelle Restriktionen:

Wenn Programme implizit von der Länge bestimmter Maschinenbefehle Gebrauch machen, weil sie z.B. PC-relativ mit Displacement im Code-Bereich adressieren, so kann dies oft nicht vom Transferassembler erkannt werden und es entsteht ein falsches Umsetzergebnis. Werden derart dubiose Codierungen vom Übersetzer erkannt (Meldung „peculiar coding“), so erfolgt keine Umsetzung. Ansonsten ist es leider Aufgabe des Programmierers, solche Programmierverfehlungen aufzuspüren und zu eliminieren.

7.3.2 Optimierter T-Code

Die 68k-Hardware führt bei Befehlen zum Transport sowie zur arithmetischen oder logischen Verknüpfung automatisch ein Update des Condition-Code-Registers (CCR) aus. Dies ist bei den RISC-Prozessoren jedoch nicht der Fall.

Bei der Transferassemblierung eines alten unveränderten 68k-Programmes muß der Transferassembler diesen Update mit einer Folge von Extrabefehlen nachbilden. Im Transferassembler ist ein Mechanismus eingebaut, der prüft, ob der nachfolgende Maschinenbefehl nicht vielleicht selbst wieder einen neuen Inhalt in das CCR schreibt. Ist dies der Fall, unterbleibt beim aktuellen Befehl die Generierung von Extrabefehlen.

Dennoch wird bei üblichen Programmen schnell eine Fülle im Grunde nutzloser CCR-Updates erzeugt, insbesondere vor allen Sprüngen und Rücksprüngen, aber auch vor Befehlen wie `LEA`, `ADDQ`, `SUBQ`, `MOVEA`, `DBcc` usw. Gleiches gilt vor allen Traps, Formaten und Assemblerdirektiven. Die T-Code-Syntax sieht darum eine Möglichkeit vor, um bei jedem Maschinenbefehl explizit die Generierung des CCR-Updates unterdrücken zu können. Dies geschieht durch Voranstellen des Underscore-Zeichens „`_`“ vor den Befehlsmnemo:

```
_MOVE  D3,D7      Unterdrücke CCR-Update
 BSR   SUBRO      Weil in subro nicht gebraucht
```

Man kann im Protokoll (Listing) des Transferassemblers leicht erkennen, ob bei der Generierung der Befehlssequenz der CCR-Update unterdrückt war oder nicht: Zwischen der hexadezimalen relativen Ablageaddresse und dem Hexcode, der dort abgelegt wurde, wird ein „`_`“ eingestreut, wenn die Unterdrückung aktiv war. Dabei wird nicht unterschieden, ob die Unterdrückung durch Eigenintelligenz des Übersetzers oder durch Befehl des Programmierers ausgelöst wurde.

Im Sinne einer kompakten und schnellen Kodierung sollte man seine T-Code-Programme mit Hilfe dieser Option optimieren. Der 68k-Assembler (der ein Transferassembler für den 68k ist) ignoriert den Unterdrückungsbefehl.

7.3.3 Zielmaschinenkonditionierte Befehle

Diese Gruppe von Maschinenbefehlen entstand neu bei der Definition des T-Codes. Je nach Zielprozessor wird eine hinsichtlich des Datenflusses unterschiedliche Maschinenbefehlssequenz (bzw. evtl auch nur ein oder kein Befehl) erzeugt. Diese Option soll den unterschiedlichen Stackphilosophien der RISC- und CISC-Prozessoren beim Unterprogrammaufruf Rechnung tragen.

`XBSR` (Zielmaschinenkonditioniert) Branch to subroutine. Bei Übersetzung für die 68k-Familie wird hier ein normaler BSR-Befehl erzeugt. Bei Übersetzung für die PowerPC-Familie entsteht ein reiner (sehr schneller) „Branch and link“. Die Rückkehradresse steht dann ausschließlich im Link-Register und es ist Aufgabe des Programmierers, dafür zu sorgen, daß sie dort sicher ist. Mit XSL kann sie notfalls später noch gerettet werden.

`XJSR` (Zielmaschinenkonditioniert) Branch to subroutine. Bei Übersetzung für die 68k-Familie wird hier ein normaler JSR-Befehl erzeugt. Bei Übersetzung für die PowerPC-Familie entsteht ein reiner (sehr schneller) „Branch and link“. Die Rückkehradresse steht dann ausschließlich im Link-Register und es ist Aufgabe des Programmierers, dafür zu sorgen, daß sie dort sicher ist. Mit XSL kann sie notfalls später noch gerettet werden.

`XRTS` Return from Subroutine. Bei 68k Prozessoren entsteht hier ein normaler RTS-Befehl. Bei Übersetzung für den PowerPC wird ein Branch by Link-Register generiert, d.h. der Stack ist nicht involviert (sehr schnell).

`XSL` Save link on stack. Bei Übersetzung für die 68k-Familie wird hier nichts generiert, da die Rückkehradresse bereits auf dem Stack steht. Beim Prozessor PowerPC wird der aktuelle Wert des Link-Registers auf den Stack geschrieben.

Vorsicht!

Die Ersetzung von BSR-Befehlen durch XBSR etc. darf nur in besonderen Fällen erfolgen, bei denen das Linkregister weder explizit noch implizit (z.B. durch den Transferassembler bei PC-relativer Adressierung) zerstört wird. Sicherer – aber langsamer – ist die Beibehaltung der Original-68k Befehle. Der PEARL-Compiler für den PowerPC springt dennoch alle Unterprogramme mit XJSR an. Folglich muß man selbstgeschriebene Maschinenunterprogramme für PEARL90 typischerweise mit XSL beginnen und mit normalem RTS verlassen, wenn sie in beiden Prozessorfamilien korrekt laufen sollen.

7.3.4 Formatdefinition

Die Formatdefinition entspricht einer Prozedurdefinition, jedoch mit der Besonderheit, daß diese Prozedur zur Assemblierzeit – also bei der Übersetzung des Assemblerquellfiles – an einer oder mehren Stellen zur Ausführung kommt. Das Innenleben der Formatprozedur besteht aus einer einfachen Folge von Befehlen einer sehr einfachen 32-Bit Akkumulatormaschine. Diese beherrscht Befehle zum Addieren, Subtrahieren, Segmentieren von Bitsequenzen und zur Ablage von Daten.

Formatnamen dürfen maximal aus 6 Zeichen bestehen, die bei der Definition aus Großbuchstaben bestehen müssen und beim Aufruf wahlweise – wie Befehlsmnemos – in Groß- oder Kleinschreibung ansprechbar sind.

Formate können bis zu 9 Parametern haben, davon können die letzten beiden – oder nur der letzte – sogenannte „Defaultparameter“ sein. Das sind Parameter, die nur bei der Definition und später nicht mehr bei der Benutzung des Formates als Aktualparameter angegeben werden. Erkennbar sind die Deafultparameter an den umschließenden runden Klammern bei der Formatdefinition (siehe Beispiel unten). Als Aktualparameter können nur zur Assemblierzeit berechenbare 32-Bit Ausdrücke benutzt werden. Diese können entweder absolut *AE* oder relativ *RE* sein. Bei der Definition des Formates muß dies für jeden einzelnen Parameter durch die Buchstaben „a“ oder „r“ angegeben werden. Beim Aufruf des Formates muß die Anzahl der Aufrufparameter und deren Typ exakt mit der Definition übereinstimmen.

Formate müssen stets so gestaltet werden, daß sie insgesamt einen Bitstring erzeugen, dessen Länge ein Vielfaches der Zahl 8 ist, weil Assembler und Transferassembler das Byte als kleinste Ablageeinheit verwenden. Man kann bei der Definition durch Nachstellen von `.B`, `.W` oder `.L` an das `FORMAT`-Mnemo angeben, ob vor Beginn der Bytegenerierung durch das Format der relative Ablage-PC auf eine Byte-, Wort- oder Langwortgrenze positioniert werden soll. Beim Aufruf des Formates nimmt der Assembler dann die nötige Einstellung der Ablageadresse selbständig vor. Wird keine Adressjustage vordefiniert, so verwendet der Assembler oder Transferassembler diejenige, die für die Maschinenbefehle seiner Zielhardware vorgesehen ist. Bei Datentabellen – die in den verschiedenen Prozessorwelten gleich aussehen müssen – wird daher die explizite Angabe von .B, .W oder .L dringend angeraten. In diesem Fall ist zusätzlich noch eine Verschiebung der Zuordnung des eventuell vor dem Format stehenden Labels um eine maximal 2 (dezimal-)stellige Anzahl von Bytes nach hinten möglich:

```
ABCD  FORMAT.L+16 ...
      ...
Test  ABCD   ...      Test liegt 16 byte oberhalb Ablage-PC
```

Die mit unserem System mitgelieferten Dateiein `PROCS.FOR`, `SUPERVIS.FOR` und `GENERAL.FOR` enthalten zahlreiche Formatdefinitionen, die man zum Kennenlernen dieser Assembleroption verwenden kann. Hier studieren wir eine hypothetische Formatdefinition zur Generierung eines Tabelleneintrages für eine Tabelle, deren Einträge jeweils aus einer 16-bit Konstanten (`$AFFE`), einer 5-bit Zahl, einer 27-bit Zahl, einer 16-bit Differenz dieser Zahlen und einer relativen 32-bit Adresse bestehen:

```
TABLX  FORMAT.W  a,a,r,($AFFE) definiert Format TABLX, Wortgrenze
/      #4(16:31)               lege das Bitmuster $AFFE ab.
/      #1(27:31)#2(5:31)       Ablage par1 5 bit, par2 27 bit
/      #A=#2#A-#1#A(16:31)     Ablage (par2-par1) in 16 bit
/      #3(R)                   Ablage par3 in 32 Bit relativ
/      e                       oder E, Ende Formatdefinition
```

Der Wert des in Klammern stehenden Defaultparameters kann natürlich statt als Konstante auch durch einen zur Assemblierzeit berechenbaren Ausdruck dargestellt werden. Eine normale Benutzung dieses Formates wäre etwa:

```
LABL1  TABLX 25,30,Label5      Defaultparameter wird nicht angegeben!
```

Dieses Beispiel benutzt den bereits erwähnten 32-Bit Akku. Mit ihm wird die Differenz (Parameter2-Parameter1) berechnet, um anschließend die Bits No. 16 bis 31 abzulegen. Mit der Definitionszeile wird das Format geöffnet. Anschließend wird mit den Folgezeilen, die mit „/“ beginnen, der Formatkörper beschrieben. Kommentarzeilen sind erlaubt, andere Assembleranweisungen sollten nicht benutzt werden.

Im einzelnen sind zur Zeit folgende Operationen implementiert:

`01001100 ...`	Eine beliebige Ziffernfolge bestehend aus 0-en und 1-en wird als Binärstring abgelegt: 10101111 legt das Byte `AF` ab.
`#7(14:22)`	Ein Teilstring (hier Bits No. 14 bis 22) des Parameters (hier 7) wird abgelegt. Statt der 7 kann jeder der Parameter 1 ... 9 benutzt werden. Das Bit mit der No. 0 ist das höchswertige, das Bit mit der No. 31 das niederwertigste.Sollen alle 32 Bit abgelegt werden, so ist `#7(0:31)` zu verwenden.
`#3(R)`	Der Parameter (hier No. 3, muß „r“ spezifiziert sein) wird als 32 bit langer relativer Wert abgelegt. Ein Zerschnippeln von relativen Parametern ist nicht zugelassen.
`#A=#4`	Wertzuweisung: Dem Akku wird der Wert des Parameters (hier No. 4) zugewiesen.
`#A+#5`	Der Parameter (hier No. 5) wird auf den Akku addiert.
`#A-#1`	Der Parameter (hier No. 1) wird vom Akku subtrahiert.
`#A=12`	Lade Konstante (hier 12) in den Akku. Es sind nur Werte von 0 ... 31 zugelassen!.
`#A+4`	Addiere Konstante (hier 4) zum Akku. Es sind nur Zahlen von 0 ... 31 zugelassen.
`#A-6`	Subtrahiere Konstante (hier 6) vom Akku. Es sind nur Zahlen von 0 ... 31 zugelassen.
`#A(12:29)`	Ein Teilstring (hier Bits No. 12 bis 29) des aktuellen Akkuinhaltes wird abgelegt. Sollen alle 32 Bit abgelegt werden, so ist `#A(0:31)` zu verwenden.
`#A>6`	Es wird sichergestellt, daß der Akku eine Zahl enthält, die größer ist als (hier) 6. Statt der 6 sind Zahlen von 0 ... 31 zugelassen. Ist die Bedingung nicht erfüllt, so wird bei der Übersetzung eine Fehlermeldung (Limit-Error) generiert.
`#A-#$`	Der aktuelle Wert des Ablage-PC wird vom Akku subtrahiert.

`#A%3`	Das Bitmuster `$80000000` wird um 3 Plätze nach rechts geschoben auf den Akku aufaddiert. Als Argument sind Zahlen von Null bis 31 – letzteres entspricht der Addition einer 1 – zugelassen. Hinweis: Man kann mit mehreren solcher Befehle ein beliebiges bis zu 32 Bit langes Bitmuster addieren.
`#A?16`	Der Inhalt des Akkus wird daraufhin geprüft, ob er in ein Vorzeichenbehaftetes 16 Bit-Wort passt. Statt der 16 können Zahlen von 1 ... 31 benutzt werden. Achtung, Zweierkomplement: das vorderste Bit des gewählten rechten Endes des Akkus muss identisch zu allen links daneben stehenden Bits sein. Ist die Bedingung nicht erfüllt, so wird bei der Übersetzung eine Fehlermeldung (Limit-Error) generiert.

Die Befehle können unmittelbar hintereinandergeschrieben oder auf mehrere Zeilen verteilt werden. Ein Leerzeichen unterbricht den Befehlskode und schaltet auf das Kommentarfeld der Zeile um.

Man kann relative Ausdrücke ebenfalls in den Akku laden, das Relativ-Attribut geht dabei zunächst verloren. Allerdings kann nach einer Rechnung der Akku mit `#A(R)` als relativierter Zeiger (32 bit) abgelegt werden. Das Ablegen von Schnipseln eines solchen lageabhängigen Akkuinhaltes macht jedoch im Normalfall keinen Sinn, da später beim Laden keine Korrektur durch den Lader erfolgen kann. Dagegen kann es sehr wohl sinnvoll sein, mit Hilfe des Akkus die Differenz zweier relativer Parameter oder die Distanz einer relativen Adresse zum Ablage-PC (`#$`) zu berechnen und als Kurzbitzahl abzulegen, weil das Ergebnis tatsächlich lageunabhängig ist.

Die Formatmaschine kodiert assemblerintern extrem kompakt. Selbst längere Formatdefinitionen belasten den Assembler kaum. Leider ist diese Kompaktheit auch der Grund für die Beschränkungen bei den Konstanten etc. Der Befehlssatz wird in zukünftigen Versionen sicher noch erweitert.

7.4 PowerPC-Assembler

Der PowerPC-Assembler ist im Transferassembler `TAPP` enthalten und wird durch die Anweisung

```
.IF_PROCTYPE  MPC601 oder
.IF_PROCTYPE  MPC604
```

aktiviert. Seine Syntax richtet sich nach dem PowerPC User's Manual von IBM und Motorola. Man beachte die dort definierte gegenüber der 68k-Welt andersartige Reihenfolge von Quellen- und Senkenangaben.

Soweit sinnvoll, werden auch die vom 68k-Assembler bekannten Direktiven unterstützt, z.B. `EQU`,`FORMAT` und `DC`. Fast alle Kurzmnemos aus dem User's Manual (z.B. für bedingte Sprünge) wurden ebenfalls implementiert. Zusätzlich akzeptiert der Assembler – wo sinnvoll – auch eine 2-Register Notation:

```
    add   r5,r30      ist gleichwertig zu
    add   r5,r5,r30
*                     ebenso ist
    addi  r6,=12      gleichwertig zu
    addi  r6,r6,=12   etc.
```

Nicht implementiert wurden die Befehle der POWER-Architektur aus dem MPC601-User's Manual.

7.5 Tabellenkapazität

Unsere Assembler bzw. Transferassembler benötigen für jedes Symbol mit einer Länge von bis zu 6 Zeichen 14 Bytes Listenplatz. Längere Symbole verbrauchen entsprechend mehr. Von dem angebotenen Workspace (z. B. `SZ=`*xx* unter **RTOS–UH**) gehen zusätzlich noch einmal ca. 600 Bytes für Pufferung etc. verloren. So kann z. B. mit `SZ=6000` auf der 68k-Variante ein Programm mit bis zu 1700 Symbolen übersetzt werden.

Der Transferassembler für den PowerPC verbraucht deutlich mehr Listenplatz als die 68k-Version. Defaultmäßig fordert er darum stets volle 64 kByte an. Bis heute reichte dieser Speicher noch bequem selbst für unsere größten maschinenkodierten Module, wie z.B. Window-Manager und Multiwindow-Editor.

7.6 FPU–Befehle und Maxi–Version

Die „MAXI"–Version des 68k-Assemblers kann zusätzlich zur kleinen Version und im Gegensatz zu den Transferassemblern den vollständigen Befehlsumfang inklusive der FPU-Befehle der MC680xx-Familie verarbeiten. Auch dabei wurde die Befehls–Syntax der Motorola Handbücher (mit den bekannten Abweichungen) zu den einzelnen Prozessoren zu Grunde gelegt.

Will man die FPU-Befehle des PowerPC benutzen, so muß der Transferassembler `TAPP` in den native PowerPC-Mode geschaltet werden.

Achtung

Bei Benutzung der FPU–Befehle in Assemblercodierten Tasks muß in jedem Fall eine Hilfszelle im Taskkopf gesetzt werden, damit der Prozeßumschalter des Betriebssystemes die benutzten FPU–Register bei einem Taskwechsel rettet!

Diese Hilfszelle „FPUSFL“ enthält in besonderer Codierung die Anzahl (PowerPC) oder Selektion (68k) der benutzten FPU–Register. FPUSFL befindet sich im Taskkopf und ist ein Byte lang. Den Offsetwert von FPUSFL wird durch Inkluden der Datei COMEQU automatisch richtig gesetzt. (Nur falls man die Datei nicht zur Hand hat: zur Zeit der Drucklegung hatte sie für beide Prozessorfamilien den Wert $45).

FPUSFL-Belegung in der 68k-Familie.

Für jedes zu rettende FPU–Register muß ein Bit in dieser Hilfszelle gesetzt werden. Zuordnung der einzelnen Bits zu den FPU–Registern:

Bit7 = FP0, Bit6 = FP1 ... Bit0 = FP7

FPUSFL-Belegung in der PowerPC-Familie.

Die Rechenformel lautet $8 + 4 * Anzahl$

FPUSFL=$00	FPU wird nicht benutzt.
FPUSFL=$0C	FR0 wird benutzt.
FPUSFL=$10	FR0 und FR1 werden benutzt.
...	
FPUSFL=$88	FR0 ... FR31 werden benutzt.

Werte außerhalb des obigen Bereiches sind unbedingt zu vermeiden, da sie zu Prozeßumschalterfehlern führen können!

Die Hilfszelle darf in einer assemblercodierten Task dynamisch verändert werden, es muß nur darauf geachtet werden, daß die Besetzung immer der aktuell benutzten Anzahl FPU–Register entspricht.

Werden in assemblercodierten PEARL–Unterprogrammen FPU-Befehle verwendet, so muß in dem PEARL–Modul, in dem die aufrufende Task definiert ist, der Übersetzungsmode des Compilers so eingestellt sein, daß die FPU eingeschaltet ist. Dabei müssen alle irgendwie benutzten FPU-Register freigegeben wurden. Sonst können sporadische und damit sehr schwer auffindbare numerische Fehler wegen der Nichtwiederkehr von Registerinhalten nach Kontextswitchen auftreten!

Benutzen Sie die FPU–Befehle nur bei Vorhandensein einer FPU in ihrem System, sonst wird die Task an der entsprechenden Stelle mit einem Fehler angehalten.

Beispiel: assemblercodierte Task

```
        .....
      .INCLUDE COMEQU           Symbolische offsets laden
      .if_proctype  MPC604      ggf. in PowerPC mode
FPMSK EQU  $10                  fr0 .. fr1
      .else                     ende PowerPC Zweig
FPMSK EQU  $C0                  FP0 und FP1
      .fin
        .....
*... Transferassemblierbare Anweisungen ....
      MOVEA.L TID,A1            Hole Task-pointer
     _MOVE.B =FPMSK,FPUSFL(A1) Setze FPU-Zelle
*... Die ersten 2 FPU-Register sind nun benutzbar
*     .....
     .if_proctype  MPC604       ggf. in PowerPC mode
      fadd  fr0,fr1             2 register freigegeben
     .else                      ende PowerPC Zweig
      FADD  FP0,FP1             2 register 68k
     .fin
```

Feststellen der Systemkonfiguration (nur 68k):			
Name	Adresse	Wert	Bedeutung
`FPUFLG`	$8CE.B	0	68881/2 nicht vorhanden
		$FF	68881/2 im System vorhanden
`F68020`	$8CF.B	0	68000, 68010, 68008 Prozessor
		$FF	68020/30/40/60

Wichtiger Hinweis

Bitte benutzen Sie möglichst die aus dem file `COMEQU` stammenden Symbole statt der Konstanten! Auch wenn wir eine Lageveränderung zentraler Objekte wenn irgend möglich vermeiden, ist diese Vorgehensweise erheblich sicherer und vereinfacht die Verwendung Ihrer Software auf den verschiedenen Hardwareplattformen.

7.7 S–Records

Die vom Assembler erzeugten S–Records bestehen aus einer Folge von ASCII–Zeichen, beginnend mit einem „S“, und haben prinzipiell folgenden Aufbau:

S*xyyaaaaaaddddddddd...ddddcs*

Hierbei bedeuten

x: Typkennung. Verwendet werden

0: Startrecord

1: Datenrecord

2: Datenrecord

3: Datenrecord

9: Endrecord

yy: Byteangabe. Angegeben wird die Anzahl der im Record noch folgenden Byte (nach ASCII–Hex –> binär Wandlung) einschließlich der Checksumme.

aaaaaa: Adreßangabe. **RTOS–UH** verwendet nur relativierte Adressen, d. h. *aaaaaa* ist relativ zur Ladeadresse des S–Record–Files gerechnet. *aaaaaa* umfaßt 2*(*x*+1) Zeichen, d. h. S1–Records können 2 Byte Offset, S2–Records 3 Byte Offset und S3–Records 4 Byte Offset ausdrücken.

dddd...: Datenbereich. Die Daten werden in hexadezimaler Form als ASCII–Text angegeben. Bei **RTOS–UH** können hier auch Zeichen auftreten, die der hexadezimale Darstellung nicht entsprechen. Es handelt sich dann um Laderdirektiven o. ä.

cs: Checksumme. Die Checksumme wird durch ein Byte derart gegeben, das die Addition aller Bytes des S–Records, beginnend bei der Längenangabe und die Checksumme einschliessend, ohne Berücksichtigung der Überläufe $FF ergibt.

Es enthalten unter **RTOS–UH**

S0–Records: im Adreßfeld die Länge des Datenbereiches, der zwischen S0– und S9–Record von Daten–Records beschrieben wird. Der Datenbereich kann interne, zusätzliche Informationen enthalten.

Daten–Records: im Adreßfeld die relative Startadresse des von diesem Record beschriebenen Datenbereiches; im Datenbereich die dazugehörigen Daten sowie ggf. Lader–Direktiven.

S9–Record: enthält nur interne Informationen.

7.8 Assembler–Fehlermeldungen

Fehlermeldungen werden zeilenweise eingebettet und durch einen Stern unter dem inkriminierten Zeichen markiert, an der Stelle, an der der Assembler die Abweichung erkennen konnte. Zusätzlich wird am linken Rand dieser Zeile `<ERROR>` eingefügt, um die fehlerhaften Zeilen in einem Listing schneller auffinden zu können.

Im Fehlerfalle wird – wann immer möglich – ein NOP–Code eingesetzt um ggf. nach dem Laden korrigieren zu können. Dies ist allerdings bei bestimmten Fehlern, die erst im Pass 2 erkannt werden können, nicht möglich.

BYTE–FRACTION.	Ein `FORMAT` wurde so definiert, daß die Gesamtzahl generierter Bits nicht durch 8 teilbar ist.
DEF–ERROR.	Falsch definiert. Z. B. Vorwärtsbezug bei `EQU` oder zu spät als Register, d. h. nachdem das Symbol bereits benutzt wurde.
DC–OVFL.	Es sind zu viele Ausdruecke innerhalb einer DC-Direktive. Auf mehrere DCs aufteilen.
DOUBLE–DEF.	Das Symbol wurde mehrmals definiert.
FORW.REF.	Vorwärtsbezug hier nicht erlaubt.
FP–FORM–ERROR.	In einem 68k-FPU-Befehl wurde ein unzulässiges (Daten-)Format angegeben.
ID_TOO_LONG.	Der Bezeichner ist länger als erlaubt.
IF/FIN.	Strukturfehler bei Benutzung von `.IF` oder `.ELSE` bzw. `.FIN`.
LENGTH.	Operation und Länge harmonieren nicht.

LIMIT.	Grenzwert des Ausdrucks überschritten, oder z. B. durch Null dividiert.
LR_LOCKED	Der Transferassembler konnte diese Anweisung nicht übersetzen, weil das Linkregister benötigt wird, aber durch ein `LOCK` vorher blockiert wurde.
MODE.	Adressierungsart ist hier nicht erlaubt.
NO_FORMAT_OPEN.	In Spalte 1 steht das Zeichen „/“, aber es ist kein `FORMAT` mehr geöffnet.
NOT–IMPL.	Nicht implementierter Befehl oder Konstrukt.
P1/2–MATCH.	Label im Pass 2 entdeckt, das im Pass 1 noch nicht vorkam. Übersetzung wird abgebrochen.
OPD/FORMAT–DOUBLE.	Bei „`OPD`“ oder „`FORMAT`“ wird ein bereits vergebener Mnemo verwendet.
R/A–ERROR.	Ausdruck *RE* statt *AE* oder umgekehrt. Der falsche Typ wird jedoch eingesetzt, Programm i. a. unbrauchbar.
r*xy*_LOCKED	Der Transferassembler konnte diese Anweisung nicht übersetzen, weil das Register r*xy* (möglich: r25 ... r31) benötigt wird, aber durch ein `LOCK` vorher blockiert wurde.
SYNTAX.	Keine Produktionsregel gefunden (3*NOP).
UNDEFINED.	Das Symbol wurde nicht definiert.

Daneben gibt es noch einige nicht an die aktuelle Zeile gebundene Fehlermeldungen, die zu einem Abbruch des Übersetzungslaufes führen:

Can't_open_include_file.	Der zu inkludende File konnte nicht gefunden oder nicht geöffnet werden (z.B. wegen exklusiver Benutzung durch anderen Prozeß).
End_of_input_file.	Bevor das reguläre Ende (`END`) des Quellfiles erreicht wurde, endete dieser. Kann durch bedingte Assemblierung entstanden sein, wenn das `END` im inaktiven Teil des Textes steht.
Illegal_Branch_Address.	Beim Transferassemblieren wurde eine Abhängigkeit der Zieladresse von Befehlslängen erkannt, die zu einem höchstwahrscheinlich fehlerhaften Ergebnis führen würde.
Incl.fileend_inside_field.	Ein File, der mit `.INCLUDE` eingebunden wurde, endet innerhalb eines aktiven Feldes der übersetzten Zeile. Es können nur komplette Zeilen inkluded werden.
Input–fail.	Abbruch des Laufes, weil Input–File nicht lesbar/vorhanden ist, oder die End–Of–File Bedingung vor dem `END` eingetreten ist.
Internal_Error.	Eine fehlerhafte Datensituation innerhalb des Übersetzers wurde durch die internen Selbstprüfungen im Assembler erkannt. Falls der Fehler beständig ist: bitte Beispielprogramm aufheben und Fehler anzeigen!
Not_PowerPC_translated.	Anweisung konnte nicht transferassembliert werden, z.B. Befehl des Supervisorprogrammiermodelles.
Peculiar coding!	Es wurde ein merkwürdiger Programmierstil erkannt, der vom Transferassembler zurückgewiesen wird. (Auch im native PowerPC-mode möglich)

Table–overflow.	Der Speicherplatz innerhalb des Assemblers reicht nicht aus. Mit `SZ` beim Aufruf des Assemblers Listenplatz vergrössern (bis `SZ` = 10000 möglich!).

7.9 Einbettung von Assemblerprogrammen

RTOS–UH eignet sich zusammen mit der Vielzahl von Systemtraps auch sehr gut für die maschinennahe Codierung. Dabei sind allerdings einige Konventionen zu beachten, da mit fehlerhaften Programmen in Maschinensprache durchaus ein „Absturz“ des gesamten Systemes verursacht werden kann. Dies kann sogar schon beim Laden eines falsch kodierten Modules erfolgen. Der Lader benötigt nämlich für Module einen sogenannten *Modulkopf* und für Tasks einen sog. *Taskkopf* im Vorspann des eigentlichen Programmes. Dem *Taskkopf* **muß** dabei ein Deklarationsblock folgen, den man unbedingt mit Hilfe des Formates `TSKDCB` generieren sollte, weil er sich zwischen 68k und PowerPC unterscheidet! Besteht ein Programmblock aus mehreren Modulen/Tasks, so müssen diese vom Programmierer miteinander verzeigert werden. Der *Nullzeiger* zeigt an, daß es keinen Vordermann oder Hintermann zu diesem Modul/Task–Kopf gibt.

Modulkopf:	`DC.L`	`0`	oder Addresse nächster M/T–Kopfs
	`DC.L`	`0`	oder Addresse vorheriger M/T–Kopfs
	`DC`	*modtype*	Typeindicator: siehe unten
	`DC.B`	`'......'`	*namelink* , 6 bytes (s.u.)
	`...`	Von hier ab freie Kodierung des Modules.	
Taskkopf:	`DC.L`	`0`	oder Addresse nächster M/T–Kopf
	`DC.L`	`0`	oder Addresse voriger M/T–Kopf
	`DC`	*tasktype*	(ist unten erläutert)
	`DC.B`	`'......'`	*namelink* , 6 bytes (s.u.)
Task-DCB:	`TSKDCB`	*prio,wsplen,start*	Datei „`GENERAL.FOR`“ included
	`...`	Von hier ab freie Kodierung der Task.	

modtype: Es sind 3 Typen von Modulen im System definiert, jedoch ist für normale Anwendungen nur das Standardmodul sinnvoll:

$0010	=	Normales Modul.
$0050	=	PEARL-Shell („`SMDL`“), nur für Compiler sinnvoll.
$0090	=	Für `PROM`-Befehl („`PMDL`“)

namelink: Entweder unmittelbar der Task– bzw. Modulname mit endigen Blanks zusammen genau 6 Buchstaben Länge, oder in den ersten 4 Byte die relativierte (siehe Beispile) Adresse auf einen beliebig langen Namen, der mit `$FF` enden muß. Im 2. Fall müssen die Bytes 5 und 6 auf Null gesetzt sein. Bei Systemausgaben (z. B. `S`–Kommando) werden Namen nur bis zum 24. Zeichen ausgegeben.

tasktype: Es sind 4 Kombinationen sinnvoll:

$0001 = Normale Task, ohne „RESIDENT“-Attribut.
$0081 = „Residente“ Task, die ihren „TWSP“ behält.
$0041 = Autostarttask, läuft nach Abort sofort.
$00C1 = Kombination: „Residente“ Autostarttask.

prio: Für Anwendertask sind nur 16 bit-Werte größer als Null zugelassen. Bei I/O–Dämonen kann der Wert Null zur Definition einer variablen Priorität verwendet werden.

wsplen: Jedes System benötigt hiervon eine große Anzahl Bytes für eigene Zwecke, hauptsächlich zur Ablage des Kontextes. Diese Mindestzahl darf auf gar keinen Fall unterschritten werden. Die absolute Mindestzahl kann man aus der Datei „COMEQU“ mit dem Symbol „PMBUF“ erhalten. Wird der Hyperprozessor benutzt, – z.B. weil eine PEARL-E/A oder der Aufruf eines PEARL-Unterprogrammes gebraucht wird – so muß zusätzlich der Platz „PMBSZ“ addiert werden. Auch die FPU verlangt weiteren Platz (der Hyperprozessor ist dann in jedem Fall mit dabei). Folgende Richtzahlen enthalten jeweils eine kleine Reserve, die für die nächste Zeit reichen sollte:

$00000100 = PowerPC ohne FPU.
$00000140 = PowerPC ohne FPU mit Hyperproc.
$00000290 = PowerPC mit FPU +++.
$00000080 = 68xxx, ohne FPU.
$000000C0 = 68xxx, ohne FPU mit Hyperproc.
$00000230 = 68020 ... 68060, mit FPU +++.

! → Beim Start einer Task setzt der Prozeßumschalter „PU“ das Register `A4 (r12)` auf den Anfang des Taskworkspace. Das Register `A5 (r13)` sollte man um den obigen gültigen Mindestwert höher als `A4 (r12)` einstellen, wenn man es wie in der PEARL-Welt als Anfangszeiger auf den lokalen workspace verwenden will. Das Register `A7` wird **nicht** gesetzt! Dies muß der Assemblerprogrammierer selbst erledigen, zum Beispiel wie folgt:

```
WSPLEN  EQU  $400
        ...
        LEA  WSPLEN.T,A7
```

! → In sehr alten Systemen vor ca. 1986 wurde ein kürzerer Taskkopf als heute benutzt – uralte Quellfiles unbedingt prüfen!

7.9.1 Beispiele für Modul–/Taskköpfe

7.9.1.1 Einzelner Taskkopf

```
*-------------------------------------------------*
      .INCLUDE   .../GENERAL.FOR                 *
*                                                 *
       DC.L 0,0          keine weiteren Koepfe    *
       DC   $0001        normale Task             *
       DC.L name-$       Zeiger auf den Namen     *
       DC   0            Nullwort                 *
       TSKDCB  100,120,start  Prio=100, Worksp=120*
*-------------------------------------------------*
start  MOVE D0,D1        erste Anweisung der Task *
       ...               weitere Aktionen         *
*
name   DC.B 'Testtask',$FF                        *
*-------------------------------------------------*
       END
```

7.9.1.2 Verzeigerung mehrerer Köpfe

In diesem Beispiel ist zu sehen, wie Task– oder Modulköpfe untereinander zu verzeigern sind. Ohne die Verzeigerung ist nach dem Laden nur der erste Kopf im System vorhanden.

```
*-------------------------------------------------*
*      Erstes Modul:                              *
*                                                 *
MOD1   DC.L MOD2          Vorwaertszeiger          *
       DC.L 0             kein Vorgaenger          *
       DC   $0010         Typ : Modul              *
       DC.B 'Modul1'      Name des Moduls          *
*      ...                                        *
       ...                freie Kodierung          *
*-------------------------------------------------*
*      zweites Modul:                             *
*                                                 *
MOD2   DC.L 0             kein weiteres Modul      *
       DC.L MOD1          vorheriges Modul         *
       DC   $0010         Typ : Modul              *
       DC.L mod2na-$      Zeiger auf Modulnamen    *
       DC   0             Langnamen-Indikator      *
*      ...                                        *
       ...                freie Kodierung          *
*      ...                                        *
*      Relativ adressierter Langname:             *
mod2na DC.B 'Mod2_long_name',$FF                  *
*      ...                                        *
       ...                freie Kodierung          *
*      ...                                        *
*-------------------------------------------------*
*      Herstellen des Bezuges auf das letzte      *
*      Modul:                                     *
*                                                 *
       END MOD2           in diesem Fall MOD2      *
```

Analog wird mit mehr als 2 Modulen oder Tasks verfahren.

7.9.2 Task-Deklarationsblock

Für die Erzeugung des Taskdeklarationsblockes sollte man unbedingt – wie oben beschrieben – das Format `TSKDCB` verwenden. Damit ist sichergestellt, daß die Software sowohl vom Assembler als auch vom Transferassembler richtig an das jeweilige Zielsystem angepasst wird. Die folgenden Beschreibungen sind daher nur für den Notfall – etwa weil die Datei `GENERAL.FOR` nicht verfügbar ist – hier angegeben. Die Werte geben den Aufbau Stand Februar 1997 wieder.

TASK-DCB für 68xxx:

`DC`	*priority*	Taskpriorität 1...255
`DC.L`	*wsplen*	Mindestwert beachten !!
`DC.L`	`0,0`	Zeiger für systemeigene Zwecke
`DC`	*priority*	wie oben, später variable Laufprio
`DC.L`	*Startadresse*	Adresse der 1. Anweisung der Task
`DC.L`	`0,0,0,0,0,0,0,0`	
`DC.L`	`0,0,0,0,0,0,0,0`	insgesamt 64 Bytes Null
`...`	`...`	Von hier an freie Kodierung.

Task-DCB für PowerPC:

`DC`	*priority*	Taskpriorität 1...255
`DC`	`0`	Systemintern
`DC.L`	*wsplen*	Mindestwert beachten !!
`DC.L`	`0,0`	Zeiger für systemeigene Zwecke
`DC`	*priority*	wie oben, später variable Laufprio
`DC`	`0`	Systemintern
`DC.L`	*Startadresse*	Adresse der 1. Anweisung der Task
`DC.L`	`0,0,0,0,0,0,0,0`	
`DC.L`	`0,0,0,0,0,0,0`	insgesamt 60 Bytes Null
`...`	`...`	Von hier an freie Kodierung.

Kapitel 8: Innenstrukturen des Systemes

8.1 Die Systemtraps

8.1.1 Hinweise zur Benutzung der Traps

Die Systemtraps sind die eigentlichen Funktionsträger innerhalb des **RTOS–UH**-Systemes. Sie sind seit mehr als einem Jahrzehnt weitgehend unverändert geblieben und haben dabei mehrere tausend Jahre makellose Betriebserfahrung vorzuweisen. Normalerweise werden Traps nur innerhalb von compilergenerierten Konstrukten – und damit in gesicherter Umgebung – aufgerufen. Dennoch stehen sie auch dem Assemblerprogrammierer zur Verfügung. Hier allerdings muß mit großer Sorgfalt gearbeitet werden: Aus Effizienzgründen prüfen Traps nicht erneut, ob sie von ihrem Aufrufer korrekt parametriert wurden. Neben dem Zerschellen der Aufrufertask sind bei Fehlparametrierungen durchaus auch andere unbeteiligte Tasks gefährdet. Im Extremfall ist auch ein Totalabsturz des Systemes nicht auszuschließen.

Änderungen an Traps werden zur Unterstützung von Robustheitsnachweisen unserer Anwender sehr sorgfältig dokumentiert. Dabei ist Abwärtskompatibilität oberstes Gebot.

Die Beschreibung der Traps gilt in gleicher Weise für die bisherigen 68k-Assemblerprogramme und für den **T–Code**. Der Transferassembler für den PowerPC erzeugt den jeweils passend parametrierten Supervisor-Call automatisch. Nur bei Traps mit eingebautem Skip – so zum Beispiel beim `TOQ` – sind Besonderheiten des T–Codes zu beachten. Bedenken Sie bitte bei der Kodierung für den PowerPC, daß alle Traps neben den jeweils angegebenen 68k-Registern auch die Register r25 ... r31 sowie das Linkregister verändern können.

Verwenden Sie unbedingt die Datei `COMEQU` um symbolische Adressen oder Offsets einzubinden. Damit werden Unterschiede bei den verschiedenen Prozessorfamilien automatisch ausgeglichen.

Das Inkluden der Datei `COMTRAP` erspart Ihnen die manuelle Definition der Trap- Opcodes.

Vorsicht:

Traps dürfen nicht von der Supervisorebene aus – etwa in Interruptroutinen – aufgerufen werden! Einzige Ausnahme ist der `DPC`–Trap, der hier eine Sonderstellung hat. Im Gegensatz dazu darf das `PIRTRI`–Link nur auf Supervisorebene aufgerufen werden.

Auch wenn viele Traps anscheinend auch auf Supervisorebene funktionieren, so ist dies in jedem Fall eine Fehlprogrammierung, weil damit das Konzept von **RTOS–UH** unterlaufen wird und die Echtzeitqualitäten massiv gefährdet werden.

8.1.2 Tabelle der Traps

Umklammerte Traps sind nicht im Nukleus, sondern in irgendeiner anderen Scheibe angesiedelt, sofern vorhanden.

$4E40	ACTQ	Activate quick. Task–ID–pointer is in A1
$4E41	TERMI	Exit = Terminate internal = self–termination
$4E42	CON	Continue task given by name in $66(A4)
$4E43	DPC	Start a dispatching cycle
$4E44	-- --	Ehemals PREVQ, $A054 benutzen!
$4E45	SCAN	System-scanner (for mounting of **RTOS**–**UH**+loader)
$4E46	REQU	Request semaphore. Adr. of sema is in A1
$4E47	RELEA	Release semaphore. Adr. of sema is in A1
$4E48	FETCE	Fetch a communication–element, D1.L=size, A1=ptr
$4E49	RELCE	Release a communication–element. Pointer is in A1
$4E4A	XIO	Xfer a communication–element to in/output–handler
$4E4B	PENTR	Procedure entering (Workspace alloc. etc.)
$4E4C	RETN	Return from procedure (complement to PENTR)
$4E4D	TOQ	Take of queue (Inside i/o–handler–tasks)
$4E4E	(TOV)	Hyperprocessor „on“ = to virtual code switching
$4E4F	OFF	Dispatching and interrupts „off“ + supervisormode
$A000	TERME	Terminate (external) task by name in $66(A4)
$A002	ERROR	Send error–message to corresponding userterminal
$A004	WSFS	Workspace forward search (A1 is loaded)
$A006	ITBO	Identify task by name in $66(A4), (A1 is loaded)
$A008	WSFA	Workspace fixed address request.
$A00A	IOWA	I/O–wait by communication–element in A1
$A00C	WSBS	Workspace backward search (A1 is loaded)
$A00E	GAPST	Generate and prepare a subtask (son–process)
$A010	TERV	Terminate (self) and vanish (son–process exit)
$A012	DVDSC	Device–(facility)–tester (LDN expected in D1)
$A014	ACT	Activate task by name in $66(A4)
$A016	TIAC	Time–scheduled activation of task by name $66(A4)
$A018	TICON	Time–scheduled continuation — “ —
$A01A	ACTEV	Interrupt–scheduled activ. of task by name – “ –
$A01C	CONEV	Interrupt–scheduled cont. of task by name – “ –
$A01E	QSA	Quote–scanner with answer (Lex.text by adr. A2)
$A020	RUBBL	Rubber for blanks (Text–pointer is in A2)
$A022	PREV	Prevent task by name in $66(A4)
$A024	TIACQ	Time–scheduled activ. of task (quick) by TID=A1
$A026	TRIGEV	Trigger = simulation of an interrupt
$A028	SUSP	Self suspending of executing task

$A02A	RWSP	Release workspace by pointer in A1
$A02C	TIRE	Time–scheduled RESUME of a task
$A02E	(PIT)	Process–data input (implement. dependent)
$A030	(POT)	Process–data output — " —
$A032	ENAB	Enable selected process–interrupts
$A034	DISAB	Disable selected process–interrupts
$A036	LITRA	Line–tracer in real environment
$A038	LITRAV	Line–tracer virtual (inside hyperproc)
$A03A	CSA	Character scan alternatively (text ptr in A2)
$A03C	IMBS	Identify module by string (String ptr in A2)
$A03E	RCLK	Read system–clock. Result is in D1.L
$A040	ITS1T	Index–tester for 1–dim arrays
$A042	ITS2T	Index–tester for 2–dim arrays
$A044	ITS3T	Index–tester for 3–dim arrays
$A046	MD2B60	Multiply D2.L by 60 (long + fast!)
$A048	ITBS	Identify task by string (string–ptr in A2)
$A04A	RSTT	Reset T-Link and new TWS
$A04C	INTD1	Integer (long) into D1 by text–pointer in A2
$A04E	TICONQ	Time–scheduled cont. of task (quick) by TID=A1
$A050	CONQ	Continue task quick by TID=A1
$A052	DELTST	(Right) Delimiter–test of text (ptr is A2)
$A054	PREVQ	Prevent task quick by TID=A1
$A056	EVACTQ	Interrupt–scheduled task–activ. quick by TID=A1
$A058	TERMEQ	Terminate task quick by TID=A1
$A05A	EVCONQ	Interrupt–scheduled task–cont. quick by TID=A1
$A05C	CACHCL	Cache clear or NOP if no cache
$A05E	STBCLK	Set Battery Hardware-Clock
$A060	ITS1TL	Index–tester for 1–dim arrays with long index
$A062	ITS2TL	Index–tester for 2–dim arrays with long index
$A064	ITS3TL	Index–tester for 3–dim arrays with long index
$A066	(DATASC)	Date to ASCII conversion
$A068	(CLKASC)	Clock to ASCII conversion
$A06A	---	Implementation dependent
$A06C	DCDERR	Decode error–message
$A06E	WFEX	Wait for exit
$A070	MSGSND	Message send
$A072	RESRB	Reserve Bolt
$A074	FREEB	Free Bolt
$A076	ENTRB	Enter Bolt
$A078	LEAVB	Leave Bolt
$A07A	TRY	Try to request semaphore
$A07C	---	Reserviert für Erweiterungen

$A07E	---	Reserviert für Erweiterungen
.....	---	Reserviert für Erweiterungen
$A0A0	---	Reserviert für Erweiterungen

Line–A–Traps oberhalb von $A090 können in besonderen OEM-Implementierungen in begrenzter Zahl ebenfalls belegt werden. Die Transferadressen aller Line–A–Traps beginnen auf EXCORG+$400 (für $A000), EXCORG+$404 (für $A002) ... usw. in 4–Byte–Schritten. EXCORG erhält man aus der Datei COMEQU (Aktuell 0 beim 68k und $4000 beim PowerPC). Also: Rechtes Trap–Byte mal 2 plus EXCORG (aus COMEQU) plus $400. Der Anschluß kann dann etwa über Scheibe 14 hergestellt werden. Dabei ist zu beachten, daß beim 68k-System von $800 abwärts bis $600 der System–Stack den Links entgegenwächst und Traplinks oberhalb des Trapcodes $A0FE bei hochaufgeladenen Systemen zerstört werden könnten.

$A0A2	???	Für OEM-Sonderanwendungen
...	...	—- “ —-
$A0FE	???	Letzter erlaubter Trap

Desweiteren stehen noch zwei Einsprungadressen im Nukleus zur Verfügung: CD7TAS und PIRTRI. Sie werden hier wie Traps beschrieben, obwohl sie anders angeschlossen werden.

CD7TAS	Convert D7 to ASCII–String
PIRTRI	Process-Interrupt trigger

ACT = $A014 — Activate Task by name

Eingaberegister: `D1.W` Priorität der Aktivierung.
`OPNAME.T` 6 ASCII–Bytes des Tasknamens oder 4 Byte Adresse des Namensstring, der mit `$FF` endet.

Veränderte Register: `D7,D1,A2`

Die angegebene Task wird in der Speicherverwaltung gesucht. Während der Suche bleibt der Trap preemptionfähig. Die Usernummer der Tasks wird nicht berücksichtigt. Falls eine so bezeichnete Task nicht gefunden wird, so erfolgt Fehlermeldung, und eine Operation unterbleibt.

Falls `D1.W EQ 0` ist, wird `D1` aus der Taskdefaultprio geladen. Ist `D1` negativ, so erfolgt eine Fehlermeldung und die Aktivierung unterbleibt. Danach wird geprüft, ob die Task bereits im Dispatcherring steht. Ist dies nicht der Fall, so wird sie gemäß der Priorität aus `D1` eingelinkt, und der Trap endet mit Dispatcherstart.

War die Task dagegen bereits im Dispatcherring, so wird die Blockierbedingung „`waiting for activation`“ untersucht und gelöscht.

1. Blockierbedingung war gesetzt.
   Es wird geprüft, ob die Task prioritätsgerecht eingelinkt ist; falls nicht, so wird sie entsprechend „umgelinkt“. Der Trap endet mit einem Dispatcherstart.

2. Blockierbedingung war nicht gesetzt.
   Die Aktivierung wird mit ihrer Priorität in den Puffer der Task geschrieben, falls dort noch Platz ist (max. 3). Wenn kein Platz im Puffer ist, so erfolgt Fehlermeldung, und die Operation unterbleibt. Der Trap endet ohne Dispatcherstart.

Fehlermeldungen:	`... wrong prio`	(`D1.W` negativ)
	`... overflow (activate)`	(Aktiv.Puffer Überlauf)
	`... not loaded (activate)`	(Task wurde nicht gef.)

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME`:

```
OPNAME EQU $66  beim 68K
OPNAME EQU $B4  beim PowerPC
```

Activate Task by interrupt–schedule **ACTEV = $A01A**

Eingaberegister:	`D1.W`	Priorität
	`OPNAME.T`	Textadresse oder Text
	`OPFATI.T`	Prozeß–Interrupt–Ereignis
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D1,D7,A1,A2`	

Eine Task, deren Name oder deren Adresse des Namens in `OPNAME.T = OPNAME(A4)` steht, wird zur Aktivierung eingeplant. In `D1` wird die Priorität der Aktivierung übergeben. Ist `D1` gelöscht, wird die Default–Priorität eingetragen. Die aktuelle Priorität der laufenden Task bleibt unbeeinflußt. In `OPFATI.T = OPFATI(A4)`
wird die Prozeß–Interrupt–Maske eingetragen, auf die die Task eingeplant werden soll. Bestehende Aktivierungs-Einplanungen werden gelöscht. Die Fehlermeldungen entsprechen denen beim Trap `ACT` beschriebenen.

Beispiel:

```
ACTEV  OPD    $A01A         Trap-Definition
       ...
       LEA    TSKNAM,A0     Adresse des Tasknamens
       MOVE.L A0,OPNAME.T   Eintrag der Adresse
       CLR    OPNAME+4.T    kein Text
      _MOVE.L =$80000000,OPFATI.T  Interrupt-Maske
       ACTEV                auf Prozessir. einplanen
       ...
TSKNAM DC.B  'Alarm'
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME` und `OPFATI`:

`OPNAME EQU $66` und `OPFATI EQU $6C` beim 68K
`OPNAME EQU $B4` und `OPFATI EQU $BC` beim PowerPC

ACTQ = $4E40 — Activate quick

Eingabe:	D1.W	Priorität
	A1.L	Adresse der Task (TID)
Ausgaberegister:	-	
Veränderte Register:	D1,D7,A1	

Eine Task, deren Adresse in A1 steht, wird aktiviert. In D1 wird die Priorität der Aktivierung übergeben. Ist D1 gleich Null, wird die Task mit ihrer Standardpriorität gestartet. Negative Prioritäten sind den Systemtasks vorbehalten und damit nicht erlaubt, sie führen zu einer Fehlermeldung. Läuft die Task schon, wird die Aktivierung gepuffert. Beim Überlaufen des Aktivierungspuffers erhält man ebenfalls eine Fehlermeldung.

Beispiel:

```
ACTQ OPD  $4E40    Trap-Definition
     ...           TID in A1
    _MOVE =$20,D1  Prio=$20
     ACTQ          Activate Task
     ...
```

Cache clear **CACHCL = $A05C**

Eingaberegister: `--`
Ausgaberegister: `--`
Veränderte Register: `D7`

Im Nukleus wird hier zunächst nur eine Leeroperation angeschlossen. Der Trap muß darum in den Implementierungsscheiben bei allen Prozessoren, die einen Cache besitzen, neu definiert werden. Seine Aufgabe besteht darin, alle Prozessor-Caches ungültig zu machen. Beim 68040 z. B. bedeutet dies, daß alle dirty-lines (Copyback mode) des Datencaches in den Speicher gebracht werden müssen.

Der Trap wird an vielen Stellen der Systemsoftware benutzt, zum Beispiel in der Shell beim „`SM`“- und “`SD`“-Befehl. Auch der Lader und manche I/O-Treiber setzen ihn ein. Der Systemprogrammierer kann nur mit ihm sicherstellen, daß etwa ein mit `MOVE` in den Datenspeicher geschriebener Maschinenbefehl vom Prozessor dort als Instruktion gefunden wird oder daß beim memory-mapped I/O die Daten nicht nur im internen Cache, sondern auch in der Außenwelt ankommen.

CD7TAS = JSR xxxx — Convert D7 to ASCII–String

Eingaberegister: `D7.W` Hex.–Zahl
Ausgaberegister: `D6.L` ASCII–Text der Hex.–Zahl
Veränderte Register: `D7`

Hierbei handelt es sich nicht um einen Trap! Vielmehr kann diese Routine über eine feste Adresse, deren Wert man mit der Datei `COMEQU` erhält, angesprungen werden:

```
JSR CD7TAS
```

angesprungen werden. Die Hex.–Zahl in `D7.W` wird Zeichen um Zeichen in einen ASCII–String verwandelt, der in `D6.L` steht. Diese Routine kann genutzt werden, um sich die Umwandlung in ASCII–Strings z. B. bei der Ausgabe einer Adresse zu sparen. Die niederwertige Hälfte von `D7`, also `D7.W` ist nach der Routine unverändert.
Beispiel:

```
.INCLUDE    .../COMEQU.NOL (ohne Liste)
 ...
_MOVE =$12A4,D7      Hex.-Zahl
 JSR  CD7TAS         konvertieren
 ...                 D6=$31324134
```

Für den Notfall (`COMEQU` nicht zur Hand) hier die wahrscheinlichen Adressen für `CD7TAS`:

```
CD7TAS EQU  $8A4    68k-Familie
CD7TAS EQU  $5210   PowerPC-Familie
```

Clock to ASCII Conversion **CLKASC = $A068**

Eingaberegister:	`D1.L`	Zeit in msec
	`A2.L`	Zieladresse des Ausgabestrings
Veränderte Register:	`D1.L,D5,D6,D7,SR`	
	`A2.L`	zeigt auf erste freie Byte nach Ausgabestring

Die in `D1.L` übergebene Zeitangabe in msec wird in einen Ausgabestring (8 Zeichen, Aufbau: hh:mm:ss) verwandelt und auf die in `A2` übergebene Adresse geschrieben. `A2` wird um die Anzahl der Zeichen erhöht.

Beispiel: (nicht T–Code kompatibel!)

```
CLKASC OPD    $A068      Trapdefinition
time   EQU    $88A       time  (Systemzelle 68k)
timeb  EQU    $88E       timeb (Systemzelle 68k)
       ...
       MOVEM.L time,D1/D2   time+timeb lesen
       ADD.L  D2,D1         Systemzeit errechnen
       LEA    BUFFER,A2     Zieladresse laden
       CLKASC               Zeit auf (A2)+
       ...
```

Man beachte bitte den `MOVEM`–Befehl im Beispiel! Würde man nämlich die Zellen time und timeb durch 2 „Moves“ lesen, so sind unsinnige Ergebnisse möglich, falls der Clockinterrupt die beiden Leseoperationen trennt. In Multiprozessorsystemen und bei den meisten RISC-Prozessoren funktioniert das jedoch nicht korrekt und so müssen derartige Sequenzen bei der Umstellung auf legalen T–Code verändert werden:

Beispiel: (legaler T–Code)

```
CLKASC OPD    $A068      Trapdefinition
RCLK   OPD    $A03E      Trapdefinition
       ...
       RCLK              Uhrzeit nach D1
       LEA    BUFFER,A2  Zieladresse laden
       CLKASC            Zeit auf (A2)+
       ...
```

CON = $4E42 — Continue Task by name

Eingaberegister:	`OPNAME.T`	Textadresse oder Text
	`A4.L`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D7,A1,A2`	

Eine Task, deren Name oder deren Adresse des Namens in `OPNAME.T = OPNAME(A4)` steht, wird fortgesetzt. Ist die Task nicht geladen oder nicht suspendiert, wird eine entsprechende Fehlermeldung ausgegeben und der Aufrufer suspendiert. Während der Suche im Speicher bleibt der Trap preemptionfähig.

Beispiel:

```
CON  OPD    $4E42       Trap-Definition
     ...
     LEA    TSKNAM,A0   Adresse des Tasknamens
     MOVE.L A0,OPNAME.T Eintrag der Adresse
    _CLR    OPNAME+4.T  kein Text
     CON                fortsetzen
     ...
```

Die Distanzwerte von `OPNAME` für den Notfall:

```
OPNAME EQU  $66   68k-Familie
OPNAME EQU  $B4   PowerPC-Familie
```

Continue Task by interrupt–schedule **CONEV = $A01C**

Eingaberegister:	`OPNAME.T`	Textadresse oder Text
	`OPFATI.T`	Prozeß–Interrupt–Maske
	`A4.L`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D7,A1,A2`	

Eine Task, deren Name oder deren Adresse des Namens in `OPNAME.T = $66(A4)` steht, wird zur Fortsetzung eingeplant. Bestehende Fortsetzungs-Einplanungen werden gelöscht. In `OPFATI.T = $6C(A4)` muß die Prozeß–Interrupt–Maske eingetragen sein. Während der Suche ist der Trap preemptionfähig.

Beispiel:

```
CONEV OPD    $A01C       Trap-Definition
      ...
      LEA    TSKNAM,A0   Adresse des Tasknamens
      MOVE.L A0,OPNAME.T Eintrag der Adresse
      CLR    OPNAME+4.T  kein Text
      MOVE.L =$00040000,OPFATI.T Interrupt-Maske
      CONEV              bei Interrupt fortsetzen
      ...
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME` und `OPFATI`:

`OPNAME EQU $66` und `OPFATI EQU $6C` beim 68K
`OPNAME EQU $B4` und `OPFATI EQU $BC` beim PowerPC

CONQ = $A050 — Continue quick

Eingaberegister: `A1.L` Adresse der Task (TID)
Ausgaberegister: `-`
Veränderte Register: `D7`

Eine Task, deren Adresse in `A1` steht, wird fortgesetzt. Ist die Task nicht suspendiert, wird eine entsprechende Fehlermeldung ausgegeben und der Aufrufer suspendiert.

Beispiel:

```
CONQ   OPD   $A050   Trap-Definition
       ...           TID in A1
       CONQ          fortsetzen
       ...
```

! → Der Trap prüft nicht, ob über `A1` überhaupt ein sinnvoller Task-ID übergeben wurde. Dies muß der Aufrufer unbedingt sicherstellen!

Character–Scan alternate **CSA = $A03A**

Eingabe-Register:	`A2.L`	Adresse des zu unters. Textes
Ausgabe-Register:	`A2.L`	Inkrementiert um 1 falls gefunden
	`SR`	Status der Funktion
Veränderte Register:	`D7,SR`	
	`PC`	überspringt das Wort nach dem Trap

Die beiden hinter dem Trap im Speicher folgenden Bytes werden nacheinander mit dem Zeichen auf `(A2)` verglichen. Stimmt eines der beiden mit dem Eingabetext überein, so wird `A2` um eins erhöht und das Statusregister auf „`EQ`“ gesetzt. Stimmt keines der beiden Bytes mit `(A2)` überein, so bleibt `A2` unverändert, und im Statusregister wird die Kondition „`NE`“ gesetzt.

In jedem Fall wird das auf den Trap folgende Rechnerwort (2 Bytes beim 68k, 4 Bytes beim PowerPC) bei der Rückkehr übersprungen.

Der Trap eignet sich für eine einfache Textanalyse und wird innerhalb des Bedieninterpreters eingesetzt. Er ist darum auch zur Realisierung neuer Bedienbefehle optimal geeignet.

Beispiel:

```
CSA  OPD $A03A  Trap-Definition
     ...        A2 zeigt auf Eingabetext
     CSA        Aufruf
     DC.B 'Aa'  Pruefe, ob kleines/grosses A folgt.
     BEQ  Weg   Wenn ja, springe mit erhoehtem A2
```

DATASC = $A066 — Date to ASCII Conversion

Eingaberegister:	D0.W	Datum
	A2.L	Zieladresse des Ausgabestrings
Veränderte Register:	D0,D7	zerstört
	A2.L	zeigt auf nächstes Byte nach Ausgabestring

Die in D0 übergebene Datumsangabe (Anzahl der Tage seit 31.12.1983) wird in einen Ausgabestring (10 Zeichen, Aufbau: *tt-mm-jjjj*) verwandelt und auf die in A2 übergebene Adresse geschrieben. A2 wird um die Anzahl der Zeichen erhöht. Ist D0.W = $0000 (nicht gesetztes Datum), so wird die Zeichenfolge „----------“ ausgegeben.

Beispiel:

```
DATASC OPD  $A066     Trapdefinition
      .INCLUDE   .../COMEQU.NOL  (ohne Liste)
       ...
      _MOVE DATE,D0   Datum aus Systemzelle laden
       LEA  BUFFER,A2 Zieladresse laden
       DATASC         Datum auf (A2)+
       ...
```

Falls die Datei COMEQU nicht zur Hand ist, hier die wahrscheinlichen Adressen von DATE:

```
DATE  EQU  $80A   bei 68k-Familie
DATE  EQU  $5058  bei PowerPC-Familie
```

Decode Error–Text **DCDERR = $A06C**

Eingaberegister:	`A2.L`	Zieladresse für Ausgabetext
	`D1.W`	Error-code-Wort
	`D5.W`	Verfügbarer Platz auf Zieladresse
Ausgaberegister:	`A2.L`	Inkrementierte Zieladresse
	`D5.W`	noch verfügbarer Platz auf Zieladresse
Veränderte Register:	`A2,D1,D5,D6,D7`	

Das 16-bit Wort in `D1` wird in wortweise zusammengesetzten Text umgewandelt und nach (A2)+ geschrieben. Dabei wird mit Hilfe von `D5` eine Überwachung vorgenommen, die ein Überschreiben des Zielpuffers verhindern kann: Die Textgenerierung endet sofort, wenn der Zählerstand in `D5` erschöpft ist. Sowohl `A2` als auch `D5` werden vom Trap sinnentsprechend verändert zurückgegeben.

Das Wort in `D1` bestehe aus den vier Nibbles (Hexzahlen) *abcd*. Das höchstwertige Nibble, hier *a*, ist für den Dekodiervorgang ohne Bedeutung, da es nur Informationen für den Error-Dämon bzw. den Exceptionhandler enthält.

Beispiel:

```
DCDERR OPD   $A06C
       LEA   output,A2    Zieladresse des Textes
       MOVEQ =25,D5       Max. Platz im Puffer
      _MOVE  =$0285,D1    Kuenstl. Error-code
       DCDERR
       .....
* Auf der Zieladresse Text 'wrong address (trap)'
* ablegen. A2 und D5 sind passend veraendert.
```

a: Für diesen Trap ohne Bedeutung					
b: Auswahl aus folgendem Vorrat:					
0	Blank	1	not	2	wrong
3	zero–division	4	CHK	5	blocks
6	breakpoint	7	directory	8	disc
9	memory	A	module	B	missing
C	underflow	D	alignment[1]	E	– –
F	– –				
c: Auswahl aus folgendem Vorrat:					
0	Blank	1	bus–error	2	device–ldn
3	prio	4	loaded	5	suspended
6	active	7	command	8	address
9	op–code	A	priviledged	B	overflow
C	in system	D	I/O	E	operand
F	– –				
d: Auswahl aus folgendem Vorrat:					
0	Blank	1	(activate)	2	(terminate)
3	(continue)	4	(xio–call)	5	(trap)
6	(floppy/harddisc)	7	loader–input	8	rec–checksum
9	label	A	(mode)	B	timing
C	(array)index	D	FPU–68881[2]	E	parameterlist
F	– –				

[1] Nur beim PowerPC
[2] Nur beim 68k

Delimiter–Test **DELTST = $A052**

Eingaberegister:	A2.L	Adresse des zu unters. Textes
Ausgaberegister:	A2.L	Inkrementiert bis Delimiter
	SR	Bei Delimiter auf „EQ“
Veränderte Register:	D7,SR	

Es wird geprüft, ob der zu untersuchende Text als nächstes Zeichen einen Delimiter enthält. Als Delimiter gelten Semikolon($3B), Bindestrich($2D) und <Return>($0D). Blanks($20) und Kommata($2C) werden überlesen. A2 wird solange erhöht, bis ein Zeichen gefunden wird, das kein Blank oder Komma ist. Ist dieses Zeichen ein Delimiter, so wird das Statusregister auf „EQ“ gesetzt, andernfalls auf „NE“. A2 zeigt auf dieses Zeichen.

Dieser Trap eignet sich zur Analyse eines Textes, z. B. bei der Realisierung neuer Bedienbefehle.

Beispiel:

```
DELTST  OPD  $A052
        ...          A2 zeigt auf den Eingabetext
        DELTST       Aufruf
        BNE NOLIM    Springt, wenn kein Delimiter
        ...
```

DISAB = $A034 — Disable Prozeßinterrupt

Eingaberegister: D0.L Interruptmaske
Ausgaberegister: -
Veränderte Register: D0,D7

Das Bitmuster in D0 wird so mit der Enable–Maske verknüpft, so daß alle Prozeßinterrupts, deren Bits in D0 auf '1' gesetzt sind, gesperrt werden.

Beispiel:

```
DISAB   OPD   $A034   Trap-Definition
        ...
       _MOVE.L =$40000000,D0  Interruptmaske
        DISAB         Sperrung des Interrupts
        ...
```

Dispatcher Call **DPC = $4E43**

Eingaberegister: -
Ausgaberegister: -
Veränderte Register: -

Wenn der Trap im Supervisormode des Prozessors aufgerufen wird, so wird dieser beendet. Der Prozessor wird in den User-mode gebracht, und es wird ein Dispatcherstart forciert. Damit ist dieser Trap das Gegenstück zum `OFF`-Trap, siehe Seite 499).

Der Trap kann allerdings auch jederzeit aus dem Usermode heraus aufgerufen werden, auch dann erfolgt ein Dispatchersuchlauf. So läßt sich ggf. erzwingen, daß auf irgendwelche Manipulationen an Taskzuständen sofort reagiert wird.

Wichtig: Interruptprozesse starten einen Dispatcherlauf nicht mit diesem Trap, sondern durch Setzen der Dispatcher-Call-Flag — das ist entweder eine Speicherzelle (68k) oder ein Bit im Prozessorzustandsregister (PowerPC)! Das Setzen erfolgt im T–Code mit dem Format `DPCALL` (aus der Datei `SUPERVIS.FOR`). Diese Flag wird grundsätzlich bei jedem Übergang vom Supervisor– in den Usermode beachtet. (Eine direkte Rückkehr vom Supervisor- in den Usermode, etwa mit `RTE`, ohne weitere Maßnahmen ist verboten!)

Beispiel (T–Code):

```
   DPC    OPD  $4E43    Trap-Definition
   OFF    OPD  $4E4F       --- " ---
          ...
          OFF           To supervisor
          ...           ... superv. code ...
          DPC           Start Dispatcher
          ...

In Interruptroutinen aber:

         .INCLUDE   .../SUPERVIS.FOR
         .INCLUDE   .../COMEQU
          ...
          DPCALL        alert dispatcher (if task-state changed)
          ...
          JMP  DISEX    always: IR-Exit by dispatcher!
```

DVDSC = $A012 — Device–Description–Link

Eingaberegister: D1.B LDN
Ausgaberegister: D1.L Diff. Adr. der Device–Parameter
Veränderte Register: D7

Der Trap liefert die Differenz der Adresse der Device–Parameter zu A1, wie sie beim DD– und SD–Kommando beschrieben werden. In D1.B muß die LDN der Datenstation übergeben werden. Es wird nicht geprüft, ob die entsprechende LDN überhaupt im System vorhanden ist. Bei Veränderungen der Device–Parameter muß also sichergestellt sein, daß die Adresse einer gültigen LDN verwendet wird, sonst kann es zu Systemabstürzen kommen, die sich u. U. erst später zeigen.

Beispiel:

```
DVDSC   OPD  $A012    Trap-Definition
        ...
       _MOVE.B =2,D1  LDN = 2 (Port 2)
        DVDSC         Adresse von Device-Para
        MOVE.B =1,1(A1,D1.L)  Zweites Byte von
        ...           A2: auf ESC-Sequenzen setzen
        ...
```

Enable Prozeßinterrupt **ENAB = $A032**

Eingaberegister: D0.L Interruptmaske
Ausgaberegister: -
Veränderte Register: D7

Das Bitmuster in D0 wird mit der Enable–Maske „geodert“. Damit werden alle Prozeßinterrupts, deren Bits in D0 auf '1' gesetzt sind, freigegeben.

Beispiel:

```
ENAB  OPD   $A032   Trap-Definition
      ...
     _MOVE.L =$80000000,D0 Interruptmaske
      ENAB          Freigabe des Intrrupts
      ...
```

ENTRB = $A076 — Enter Boltvariable

Eingaberegister: `A1` Adresse der Bolt-Variablen
Ausgaberegister: `-`
Veränderte Register: `D7`

Die mit `A1` angegebene Boltvariable wird daraufhin untersucht, ob ein weiterer „Enter“ möglich ist.

Ist dies der Fall, so wird der Entercount um eins erhöht und der Trap ohne weitere Aktion verlassen.

Die aufrufende Task wird in folgenden Fällen blockiert:

- Wenn die Boltvariable „reserved“ ist.
- Wenn der Entercount erschöpft ist.
- Wenn ein oder mehrere „Reserver“ bereits ihr Interesse angemeldet haben und warten.

Wenn die aufrufende Task blockiert wird, so wird der Boltvariablen der Zustand „mindestens ein Enterer wartet“ aufgeodert. Neuer Taskzustand ist dann „`SEMA`“.

Maximal kann eine Boltvariable in **RTOS–UH** 8191 mal „entered“ sein.

Beispiel:

```
ENTRB OPD   $A076
      ...
      LEA   Boltx,A1   Adresse der Boltvariablen
      ENTRB            Task wird je nach
      ...              Vorzustand evtl. blockiert
Boltx DC    0          Bolts sind 16-bit Obj.
```

Write Error–Message **ERROR = $A002**

Eingaberegister: Parameter über `PC`, ggf. `A1` sowie `$66(A4)`
Ausgaberegister: `-`
Veränderte Register: `D7,PC` überspringt Wort nach dem Trap

Es wird eine wortweise zusammengesetzte Meldung erzeugt und im Normalfall über den Error–Dämon (Task `#ERRDM`) zum Standard–Error-device/file des Nutzers geleitet. Der Text wird durch den Inhalt der Zelle `OPNAME.T = OPNAME(A4)` sowie durch das Wort hinter dem Trap bestimmt. Wenn dieses aus den Hexziffern „*abcd*“ besteht, so bestimmen *b*, *c* und *d* den eigentlichen Text, während *a* eine Zusatzinformation festlegt. In `OPNAME.T` steht entweder ein 6 Byte langer Text oder eine 4 Byte Adresse gefolgt von einem Nullwort (Adreßindikator). Textende durch `$FF`. Die Textausgabe wird bei einem Leerzeichen abgebrochen. Der Text muß konstant sein, da er nicht gepuffert wird!

Wenn für den aufrufenden Prozeß ein eigener Exception-Handler angeschlossen ist, so wird dieser aktiviert. Je nach im Exception-Frame vereinbartem Mode unterbleibt ggf. die Ausgabe des Textes durch den Error-Dämon. Die Dekodierung des Textes ist mit Hilfe des `DCDERR`-Traps auf Seite 467 möglich. Auch der Error-Dämon benutzt intern den `DCDERR`-Trap.

Beispiel:

```
ERROR  OPD   $A002
       ERROR        Trap-Aufruf
       DC    $1234  a=1,b=2,c=3,d=4
* Meldung waere >>task:filename wrong prio (xio-call)
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME`:

```
OPNAME EQU $66  beim 68K
OPNAME EQU $B4  beim PowerPC
```

a: enthält folgende funktionelle Bits:					
2	unterdrücke den Text in `OPNAME`				
8	suspendiere die aufrufende Task				
b: Auswahl aus folgendem Vorrat:					
0	Blank	1	not	2	wrong
3	zero–division	4	CHK	5	blocks
6	breakpoint	7	directory	8	disc
9	memory	A	module	B	missing
C	underflow	D	alignment[1]	E	– –
F	– –				
c: Auswahl aus folgendem Vorrat:					
0	Blank	1	bus–error	2	device–ldn
3	prio	4	loaded	5	suspended
6	active	7	command	8	address
9	op–code	A	priviledged	B	overflow
C	in system	D	I/O	E	operand
F	– –				
d: Auswahl aus folgendem Vorrat:					
0	Blank	1	(activate)	2	(terminate)
3	(continue)	4	(xio–call)	5	(trap)
6	(floppy/harddisc)	7	loader–input	8	rec–checksum
9	label	A	(mode)	B	timing
C	(array)index	D	FPU–68881[2]	E	parameterlist
F	– –				

[1] Nur beim PowerPC
[2] Nur beim 68k

Interrupt–schedule activation quick **EVACTQ = $A056**

Eingaberegister:	`D1.W`	Priorität
	`A1.L`	Adresse der Task (TID)
	`OPFATI.T`	Prozeß–Interrupt–Maske
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D1,D6,D7,A1`	

Eine Task, deren Adresse in `A1` steht, wird zur Aktivierung eingeplant. Bestehende Aktivierungs-Einplanungen werden gelöscht. In `D1` wird die Priorität der Aktivierung übergeben. Ist `D1` gelöscht, wird die Default–Priorität eingesetzt. Die Priorität der aktuell laufenden Task wird nicht geändert. In `OPFATI.T = OPFATI(A4)` muß die Prozeß–Interrupt–Maske eingetragen sein.

Beispiel:

```
EVACTQ  OPD   $A056   Trap-Definition
        ...           TID in A1
       _MOVE.L =$80004000,OPFATI.T Interrupt-Maske
        EVACTQ        auf Interrupt einplanen
        ...
```

Für den Notfall (Datei `COMEQU` nicht zur Hand) hier die wahrscheinlichen Werte von `OPFATI`:

```
OPFATI  EQU  $6C  in der 68k-Familie
OPFATI  EQU  $BC  in der PowerPC-Familie
```

EVCONQ = $A05A — Event continue quick

Eingaberegister:	`A1.L`	Adresse der Task (TID)
	`OPFATI.T`	Prozeß–Interrupt–Maske
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D6,D7,A1`	

Eine Task, deren Adresse in `A1` steht, wird zur Fortsetzung eingeplant. Bestehende Fortsetzungs-Einplanungen werden gelöscht. In `OPFATI.T = $6C(A4)` muß die Prozeß–Interrupt–Maske eingetragen sein.

Beispiel:

```
EVCONQ  OPD   $A05A   Trap-Definition
        ...           TID in A1
       _MOVE.L =$80001000,OPFATI.T Interrupt-Maske
        EVCONQ        auf Interrupt einplanen
        ...
```

Für den Notfall (Datei `COMEQU` nicht zur Hand) hier die wahrscheinlichen Werte von `OPFATI`:

```
OPFATI  EQU  $6C  in der 68k-Familie
OPFATI  EQU  $BC  in der PowerPC-Familie
```

Fetch Communication–Element **FETCE = $4E48**

Eingaberegister:	`D1.L`	Größe des I/O–Buffers
Ausgaberegister:	`A1.L`	Adresse des CE's
Veränderte Register:	`D1,D6,D7`	

Der Inhalt von `D1.L` wird als Größe des effektiv nutzbaren zu schaffenden Puffers im CE angesehen. Will man den Puffer (`IOBUF`, siehe Beschreibung CE in 8.3.1 auf Seite 559) nicht benutzen, so ist durchaus ein Aufruf mit `D1.L=0` sinnvoll. Im Register `A1` wird die Adresse des CEs zurückgegeben. Die Ausführung dieser Instruktion kann die exekutierende Task blockieren:

1. Die Task hat bereits ihr Kontingent an CE–Speicherraum verbraucht. Sie wird mit „`CWS?`“ blockiert und erst wieder lauffähig, wenn andere in ihrem Besitz befindliche CEs zu freiem Speicher rückverwandelt sind.
2. **RTOS–UH** hat nicht mehr genügend Speicher zur Verfügung. Die Task wird in der Kondition „`PWS?`“ blockiert und erst wieder lauffähig, wenn irgendwo genügend Speicher freigeworden ist.

Die Blockierung im ersten Fall läßt sich vermeiden, wenn das Least–significant Bit (ungerade Zahl) im Eingaberegister `D1.L` gesetzt ist. Dieses Bit wird nicht bei der Größenberechnung berücksichtigt, sondern dient als Indikator für die Zulassung einer „kontingentüberschreitenden“ Anforderung.

Das CE wird vom `FETCE` mit folgenden Einträgen vorparametriert:

`PRIO`	Eigenpriorität der Task
`BUADR`	Adresse von `IOBUF(A1)`
`STATIO`	`$00`
`FNAME}`	1 Blank+`$FF`

Die restlichen Parameter (`LDNIO, RECLEN, MODE ...`) müssen von der Task besetzt werden, bevor das CE über „`XIO`“ benutzt werden kann. Natürlich dürfen dabei auch die obigen vorbesetzten Parameter verändert werden.

FREEB = $A074 — Free Boltvariable

Eingabe-Register: `A1.L` Addresse der Boltvariablen
Veränderte Register: `D5,D6,D7`

Dieser Trap ist Teil des „`FREE`“-Konstruktes aus PEARL. Er kann allerdings auch ohne den entsprechenden Umgebungscode direkt benutzt werden. Die mit `A1` adressierte Boltvariable wird bedingungslos auf den Zustand „`FREE`“ gebracht. Evtl. vorhandene auf diese Variable wartende „Enterer“ oder „Reserver“ werden entblockiert, was ggf. wie jede Entblockierung einen Dispatcherstart bewirkt.

Wenn man als Assemblerprogrammierer das extrem schnelle Konstrukt des PEARL-Compilers beim „`FREE`“-Statement nachbilden will, so kann wie folgt verfahren werden:

```
FREEB  OPD   $A074
       ...
       LEA   Boltx,A1     Adresse der 16-bit Bolt
       LSL   (A1)         Test and clear sign-bit
       BEQ.B Weiter       Branch if no waiting Task
       FREEB              Total clear etc.
Weiter ...
Boltx  DC    0            Bolts sind 16 Bit, init 0
```

Bitte verwenden Sie nur entweder den Trap pur oder exakt obiges schnelleres Konstrukt. Der Transferassembler für den PowerPC kann nämlich nur genau diesen `LSL`-Befehl in eine gegen Taskwechsel gesicherte Ersatzkonstruktion mit Hilfe des `lwarx`-Befehles übersetzen.

Der Trap, bzw. das komplette PEARL-Konstrukt kann auch benutzt werden, um eine Boltvariable unbedingt freizugeben. Eine Boltvariable im „entered-state“ dürfte eigentlich nie dem `FREE`-Trap angeboten werden – wenn man sich über die Folgen im Klaren ist, kann das dennoch eine wichtige Programmierhilfe zur Re-Initialisierung sein: Die Boltvariable steht hinterher auf Null, natürlich darf sich dabei keine lebendige Task mehr im kritischen Pfad befinden.

Generate and prepare SubTask **GAPST = $A00E**

Eingabe-Register:	D1.L	Total size of required task–header
	D6.W	Priority of son–process
	OPNAME.T	6 Bytes of son's name (kann später noch geändert werden)
	A4.L	muß auf Taskworkspace zeigen
Ausgabe-Register:	A1.L	Pointer to generated task–header
	CCR	„EQ“ if possible, „NE“ if no space av.
Veränderte Register:	D1,D7	

Es wird ein Taskkopf der in D1 angegebenen Größe (muß mindestens der Größe des Task-DCB entsprechen, die aus der Zelle PTHLEN gelesen werden kann, wahrscheinlicher Inhalt: $62) nach Suche von oben nach unten erzeugt. Konnte der benötigte Platz nicht gefunden werden, so retourniert der Trap mit „NE“ und es wird keine Aktion ausgeführt. Anderenfalls antwortet er mit „EQ“ und läd A1 (Zeiger auf den Taskkopf). Der Task–Declaration Block „Task-DCB“ wird wie folgt vorbesetzt:

TYPE	:	User–no. des Aufrufers + TYPE Task
PRIO	:	Eingangswert D6, nicht mehr änderbar!
NAME	:	6 Bytes aus OPNAME.T des Aufrufers, änderbar
WSPLEN	:	Defaultiert zu $78 (minimaler Platz), änderbar
...	:	No schedule, No buff. activation, no TWS
BLOCK	:	Waiting for activation (blocked, but linked)

Der Anwender muß jetzt unbedingt den Start–PC, SPC(A1), auf die zu exekutierende Code–Sequenz bringen! Er kann WSPLEN(A1) und NAME(A1) noch verändern (etwa bei Langnamen den langrelativen Zeiger einsetzen). Irgendwelche Parameter kann man dem Sohn nur über die Zellen hinter dessen Task-DCB – also nicht vor $64(A1) – einschreiben, wenn D1.L groß genug war. Der Sohn kann die Parameter dort später mit Hilfe seiner eigenen TID abholen und auswerten.

Nachdem alle Parameter versorgt worden sind, kann man das Blockbyte löschen und einen Dispatcherstart mit dem Trap `DPC` (`$4E43`) wagen. Der Sohn setzt sich nun in Gang und raubt — je nach PRIO — eventuell dem erzeugenden Prozeß den Prozessor. Irgenwann wird aber auch der erzeugende Prozeß hinter dem `CLR BLOCK(A1)` oder dem `DPC`–Trap fortgesetzt, er muß also entsprechend weitergeführt werden.

Der Trap wird im Bedieninterpreter benutzt, z. B. `COPY`, `LOAD`, `P`, `AS` um diesen zeitlich sowie speicherplatzmäßig zu entlasten.

Er ist hervorragend für kompliziertere Shell–Extensions geeignet, erfordert allerdings auch Sorgfalt bei der Anwendung. Wenn der Sohnprozeß nach getaner Arbeit verschwinden soll, so muß man ihn nur auf den Trap `TERV` (`$A010`) statt `TERMI` (`$4E41`) laufen lassen. Wenn man auf das Ende des Sohnprozesses warten will, so empfiehlt sich dafür der Trap `WFEX`, beschrieben auf Seite 539.
Verwenden Sie unbedingt die Datei `COMEQU` oder `COMEQU.NOL` (ohne Liste) um die richtigen Displacements zu erhalten. Zwischen dem 68k und dem PowerPC gibt es einige Unterschiede

Identify Module by String **IMBS = $A03C**

Eingaberegister:	A2.L	Adresse des Modulnamens
Ausgaberegister:	A1.L	Adresse des Moduls
	SR	„EQ“ wenn gefunden, sonst „NE“
Veränderte Register:	D7,A2	

Der RAM–Bereich des Rechners wird nach einem Modul durchsucht, dessen Name mit dem von A2 adressierten übereinstimmt. Wird das Modul gefunden, antwortet der Trap mit „EQ“, und in A1 steht die Adresse des Moduls. Wird das Modul nicht gefunden, lautet die Antwort „NE“. Der Name muß mit einem ASCII–Zeichen kleiner $2F oder Semikolon ($3B) enden. Während der Suche ist der Trap preemptionfähig.

Beispiel:

```
IMBS  OPD $A03C    Trap-Definition
      ...
      LEA TEXT,A2  Adr. von Modulnamen-> A2
      IMBS         Suchen
      BEQ FOUND    Springe, wenn gefunden
      ...
TEXT  DC.B 'Mist'  Modulname
      DC.B $20     Ende des Namens
```

INTD1 = $A04C — Integer to D1

Eingaberegister:	`A2.L`	Adresse des Textes
Ausgaberegister:	`D1.L`	32 Bit Integer
	`SR`	„`NE`“ wenn keine Ziffer, sonst „`EQ`“
	`A2`	um Anzahl Ziffern inkrementiert
Veränderte Register:	`D7`	

Eine Zahl in ASCII–Darstellung wird in eine 32 Bit Integer–Zahl gewandelt. In `A2` muß die Anfangsadresse der ASCII–Zahl stehen. Das Ergebnis wird in `D1` zurückgegeben. Die Umwandlung wird abgebrochen, wenn `A2` auf ein ASCII–Zeichen zeigt, welches nicht zwischen 0 (`$30`) und 9 (`$39`) zeigt. Liegt gleich das erste Zeichen außerhalb, so antwortet der Trap mit „`NE`“. Ist das erste Zeichen eine Ziffer, ist die Antwort „`EQ`“. Es wird nicht geprüft, ob ein Overflow auftritt!

Beispiel:

```
INTD1  OPD $A04C  Trap-Definition
       ...        A2 zeigt auf ASCII-String
       INTD1      Wandeln
       BNE MUELL  springe, wenn keine Zahl
       ...
```

I/O–Wait–Function **IOWA = $A00A**

Eingaberegister: `A1.L` muß auf ein CE zeigen
Ausgaberegister: `-`
Veränderte Register: `D7`

Es wird geprüft, ob das mit `A1` bezeichnete Element noch in einer Warteschlange steht oder noch in laufender Bearbeitung der Betreuungstask ist. In diesem Fall wird die den „`IOWA`“ exekutierende Task blockiert im Status „`I/O?`“. Ist das Element bereits vollständig bearbeitet, so wirkt der Befehl wie ein „No Operation“.

Mit der Blockierung der Task wird im CE–Modewort nachträglich das „Wartebit“ gesetzt, so daß die aufrufende Task sofort mit der Beendigung des I/O–Vorganges wieder lauffähig wird.

Die Verwendung dieses Traps empfiehlt sich insbesondere, wenn Inputoperationen früh vor der Benutzung der Daten in Auftrag gegeben wurden und die Daten nun gebraucht werden. Auch nach einem Output ohne Wartebit muß diese Funktion aufgerufen werden, wenn das CE neu parametriert werden soll.

Warnung:

Der Trap kann in der jetzigen Form nicht benutzt werden, wenn das CE mit gesetztem Return-Bit zum I/O-Dämonen geschickt wurde. Die Blockierung durch diesen Trap wird nämlich nicht aufgehoben, wenn das CE in die eigene Warteschlange zurückkehrt, und die Task bleibt ewig blockiert.

ITBO = $A006 — Identify Task by Opname

Eingaberegister:	`OPNAME.T`	Textadresse oder Text
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`A1.L`	Adresse der Task
	`SR`	„`EQ`“ wenn gefunden, sonst „`NE`“
Veränderte Register:	`D7,A2`	

Der RAM–Bereich des Rechners wird nach einer Task durchsucht. Die Task wird durch den Inhalt von `OPNAME.T = OPNAME(A4)` beschrieben. In `OPNAME.T` steht entweder ein 6 Byte langer Text oder eine 4 Byte Adresse eines Textes, der mit einem ASCII–Zeichen kleiner `$2F` oder Semikolon (`$3B`) enden muß, und ein Wort `$0000`. Wird die Task gefunden, antwortet der Trap mit „`EQ`“, und in `A1` steht die Adresse der Task. Wird die Task nicht gefunden, lautet die Antwort „`NE`“.

Beispiel:

```
ITBO   OPD  $A006  Trap-Definition
       ...
       MOVE.L =$41464645,OPNAME.T 'AFFE' nach OPNAME
       MOVE  =$2020,OPNAME+4.T  Mit Blanks auf 6 Byte
       ITBO        Suchen
       BEQ FOUND   Springe, wenn gefunden
       ...
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME`:

```
OPNAME EQU $66  beim 68K
OPNAME EQU $B4  beim PowerPC
```

Identify Task by String **ITBS = $A048**

Eingaberegister:	A2.L	Adresse des Tasknamens
Ausgaberegister:	A1.L	Adresse der Task
	SR	„EQ“ wenn gefunden, sonst „NE“

Veränderte Register: D7,A2

Der RAM–Bereich des Rechners wird nach einer Task durchsucht, deren Namen mit dem von A2 adressierten übereinstimmt. Wird die Task gefunden, antwortet der Trap mit „EQ“ und in A1 steht die Adresse der Task (TID). Wird die Task nicht gefunden, lautet die Antwort „NE“. Der Name muß mit einem ASCII–Zeichen kleiner $2F oder Semikolon ($3B) enden.

Während der Suche ist der Trap preemptionfähig.

Beispiel:

```
ITBS   OPD  $A048   Trap-Definition
       ...
       LEA TEXT,A2  Adr. des Tasknamen-> A2
       ITBS         Suchen
       BEQ FOUND    Springe, wenn gefunden
       ...
TEXT   DC.B 'Mist'  Taskname
       DC.B $20    Ende des Namens
```

ITS1T = $A040 — Index–test one–dimension

Eingaberegister:	`A1.L`	Adresse Feldbeschreibungsblock
	`D0.W`	linearer Index
Ausgaberegister:	`-`	
Veränderte Register:	`A1,D5,D6,D7`	

Es wird geprüft, ob der lineare Index eines eindimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, hier also auf die Feldgrenze der ersten Dimension. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Er ist nicht für selbstgeschriebene Assemblerprogramme gedacht, da man dort meist bessere Prüfmöglichkeiten zur Verfügung hat.

Beispiel:

```
ITS1T   OPD  $A040  Trap-Definition
        ...         Index in D0.W
        ...         A1.L zeigt auf 1st Dimension
        ITS1T       teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

Long Index–test one–dimension **ITS1TL = $A060**

Eingaberegister: `A1.L` Adresse Feldbeschreibungsblock
`D0.L` linearer Index

Ausgaberegister: `-`

Veränderte Register: `A1,D5,D6,D7`

Es wird geprüft, ob der lineare Index eines eindimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, hier also auf die Feldgrenze der ersten Dimension. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Er ist eigentlich nicht zur Anwendung in Assemblerprogrammen gedacht.

Beispiel:

```
ITS1TL  OPD  $A060  Trap-Definition
        ...         Index in D0.L
        ...         A1.L zeigt auf 1st Dimension
        ITS1TL      teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

ITS2T = $A042 — Index–test two–dimension

Eingaberegister: `A1.L` Adresse Feldbeschreibungsblock
`D0.W` linearer Index

Ausgaberegister: `-`

Veränderte Register: `A1,D5,D6,D7`

Es wird geprüft, ob der lineare Index eines zweidimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, in dem die Feldgrenzen in umgekehrter Reihenfolge liegen. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Er ist eigentlich nicht zur Anwendung in Assemblerprogrammen gedacht.

Beispiel:

```
ITS2T   OPD  $A042  Trap-Definition
        ...         Index in D0.W
        ...         A1.L zeigt auf Feldbeschreibungsb.
        ITS2T       teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

Long Index–test two–dimension **ITS2TL = $A062**

Eingaberegister: `A1.L` Adresse Feldbeschreibungsblock
`D0.L` linearer Index

Ausgaberegister: `-`

Veränderte Register: `A1,D5,D6,D7`

Es wird geprüft, ob der lineare Index eines zweidimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, in dem die Feldgrenzen in umgekehrter Reihenfolge liegen. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Der Trap ist eigentlich nicht zur Anwendung durch den Assemblerprogrammierer gedacht.

Beispiel:

```
ITS2TL  OPD  $A062  Trap-Definition
        ...         Index in D0.L
        ...         A1.L zeigt auf Feldbeschreibungsb.
        ITS2TL      teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

ITS3T = $A044 — Index–test three–dimension

Eingaberegister: `A1.L` Adresse Feldbeschreibungsblock
`D0.W` linearer Index

Ausgaberegister: `-`

Veränderte Register: `A1,D5,D6,D7`

Es wird geprüft, ob der lineare Index eines dreidimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, in dem die Feldgrenzen in umgekehrter Reihenfolge liegen. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Er ist eigentlich nicht zur Anwendung durch den Assemblerprogrammierer gedacht.

Beispiel:

```
ITS3T   OPD  $A044  Trap-Definition
        ...         Index in D0.W
        ...         A1.L zeigt auf Feldbeschreibungsb.
        ITS3T       teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

Long Index–test three–dimension **ITS3TL = $A064**

Eingaberegister:	`A1.L`	Adresse Feldbeschreibungsblock
	`D0.L`	linearer Index
Ausgaberegister:	`-`	
Veränderte Register:	`A1,D5,D6,D7`	

Es wird geprüft, ob der lineare Index eines dreidimensionalen Feldes innerhalb der Feldgrenzen liegt. `A1` muß auf den Feldbeschreibungsblock (siehe auch PEARL-Assembler-UP, 8.4.1) zeigen, in dem die Feldgrenzen in umgekehrter Reihenfolge liegen. Wenn der Index außerhalb der Feldgrenzen liegt, wird die Fehlermeldung „`wrong index`“ ausgegeben. War die Zeilenüberwachung eingeschaltet, wird auch die Zeilennummer mit ausgegeben. Der Pearl–Compiler generiert diesen Trap bei der `+T`–Option. Er ist wie alle Indextest-Traps eigentlich nicht zur Anwendung durch den Assemblerprogrammierer gedacht.

Beispiel:

```
ITS3TL  OPD  $A064  Trap-Definition
        ...        Index in D0.L
        ...        A1.L zeigt auf Feldbeschreibungsb.
        ITS3TL     teste Index
        ...
```

Hinweis:

Dieser Trap wird vom PEARL90-System nicht mehr benutzt. Er ist nur noch zur Abwärtskompatibilität im Systemkern enthalten!

LEAVB = $A078 — Leave Boltvariable

Eingaberegister: `A1.L` Adresse der Boltvariablen
Ausgaberegister: `-`
Veränderte Register: `D5,D6,D7`

Der Trap ist das Gegenstück zum `ENTRB`-Trap, der auf Seite 474 beschrieben ist. Es sind vier Fälle denkbar:

- Der Entercount ist im normalen Bereich, und es wartet kein „Reserver“: Einzige Aktion ist die Dekrementierung der Boltvariablen.
- Der Entercount beim Aufruf ist 1, und es warten 1 oder mehrere „Reserver“: Die Boltvariable geht in den Zustand „reserved“, und der höchstpriore auf sie wartende Prozeß wird lauffähig gemacht. Ein Dispatcherlauf folgt.
- Der Entercount beim Aufruf steht auf dem Maximum (`$3FFF`), und ein oder mehrere „Enterer“ warten: Alle wartenden Tasks werden freigegeben (Dispatcherlauf) und können ihre Anforderung wiederholen.
- Der Entercount beim Aufruf war fälschlicherweise Null: Eine Reduktion unterbleibt, aber evtl. auf die Boltvariable wartende Tasks werden freigegeben.

Beispiel:

```
LEAVB  OPD  $A078
       ...
       LEA  Boltx,A1
       LEAVB
       ...
```

Line Tracer **LITRA = $A036**

Eingaberegister:	PC	Zeilennummer steht nach PC
	BRKADR	Break–Adresse
Ausgaberegister:	-	
Veränderte Register:	A1,D1,D6,D7	

Wenn eine Task diesen Trap exekutiert und die Zeilennummer in BRKADR = $3E(TID) mit der Zeilennummer nach diesem Trap übereinstimmt, wird sie suspendiert, und es erfolgt die Meldung „breakpoint suspended“ unter Angabe der Zeilennummer. Die Zeilennummer nach dem Trap wird in jedem Fall übersprungen. Desweiteren wird die hinter dem Trap stehende Zeilennummer in LINENO.T = $A2(A4) eingetragen und ersetzt die bisher evtl. dort stehende. Der PEARL–Compiler generiert diesen Trap, wenn die +M–Option eingeschaltet ist und er realen Code erzeugt. Der Eintrag der Zeilennummer nach BRKADR kann mit Hilfe des TRACE–Kommandos erfolgen.

Beispiel:

```
LITRA   OPD  $A036   Trap-Definition
        ...
        LITRA        Line Trace
        DC  $0001    Zeile Nummer 1
        ...
```

LITRAV = $A038 — Line Tracer virtuell

Eingaberegister: `D1.W` Zeilennummer

Ausgaberegister: `-`

Veränderte Register: `A1,D1,D6,D7`

Wenn eine Task diesen Trap exekutiert und die Zeilennummer in `BRKADR = $3E(TID)` mit der Zeilennummer in `D1` übereinstimmt, wird sie suspendiert, und es erfolgt die Meldung „`breakpoint suspended`“ unter Angabe der Zeilennummer. Die Zeilennummer in `D1` wird in `LINENO.T = $A2(A4)` eingetragen. Der Hyperprozessor (Laufzeitsystem von PEARL) benutzt diesen Trap als Teil eines virtuellen Befehls, den der Pearl–Compiler bei eingeschalteter `+M`–Option erzeugt. Der Eintrag der Zeilennummer nach `BRKADR` kann mit Hilfe des `TRACE`–Kommandos erfolgen.

Beispiel:

```
LITRAV  OPD   $A038  Trap-Definition
        ...
       _MOVEQ =1,D1  Zeilennummer nach D1
        LITRAV       Line Trace virtuell
        ...
```

Multiply D2 by 60 **MD2B60 = $A046**

Eingaberegister: `D2.L` wird mit 60 multipliziert
Ausgaberegister: `D2.L`
Veränderte Register: `D2,D7`

Der Inhalt von `D2` wird mit `60` multipliziert. Das Ergebnis steht ebenfalls in `D2`. Dieser Trap kann bei der Berechnung der Uhrzeit in Stunden, Minuten, Sekunden verwendet werden.

Beispiel:

```
MD2B60  OPD  $A046     Trap-Definition
        ...            D2 Stunden
        MD2B60         aus Stunden->Minuten
        MOVE.L D2,MIN  Speichern
        MD2B60         aus Min.->Sekunden
        MOVE.L D2,SEC  Speichern
        ...
```

MSGSND = $A070 — Message send

Eingaberegister:	A1.L	Pointer Communication–Element
	D1.L	Task-Identifier der Zieltask
Ausgaberegister:	-	
Veränderte Register:	D1,D5,D6,D7	

Mit diesem Trap kann einer in D1.L über ihren Task-Identifier bezeichneten Task das in A1.L bezeichnete Communication–Element geschickt werden. Ist die Zieltask inaktiv oder blockiert („waiting for activation“), so wird sie aktiviert bzw. diese eine Blockierbedingung wird aufgehoben. Ist deren Defaultpriorität Null, so erfolgt eine dynamische Priorisierung (siehe XIO, Seite 550). Ist im Mode-Byte des CE das Wartebit gesetzt, so wird die aufrufende Task durch den Trap blockiert im Zustand I/O?. Diese Blockierung hebt der Empfänger der Nachricht nach deren Auswertung mit Hilfe des RELCE-Traps erst später wieder auf.

Der Trap funktioniert völlig analog zum XIO-Trap, kann allerdings das CE an beliebige Tasks verschicken. Genau wie beim XIO wird auch hier eine prioritätsgerechte (an Hand der Zelle PRIO im CE) Einkettung vorgenommen: Dringende Nachrichten kommen ganz nach vorne in die Schlange.

Das weitere Schicksal des CE nach dessen Abarbeitung durch den Message-Empfänger wird durch das Byte STATIO im CE bestimmt. Wenn das Bit STABRE (Bitno. 1) gesetzt ist, wird das CE mit dem RELCE des Empfängers in freien Speicher verwandelt. Ist dagegen das Bit STABRT(Bitno. 2) gesetzt, so kehrt das CE nach Abarbeitung in die eigene CE-Schlange des Aufrufers zurück und kann von dort bei Bedarf mit dem TOQ-Trap geholt werden.

Beispiel:

```
MSGSND  OPD     $A070
        ...     CE nach A1
        ...     Target TID nach D1.L
        MSGSND
```

Switch Dispatcher off **OFF = $4E4F**

Eingaberegister: -
Ausgaberegister: -
Veränderte Register: SR

Der Prozessor wird in den privilegierten Mode mit gesperrtem Interruptsystem (beim 68k: Interruptebene 7) gebracht. Damit kommen keine Interrupts und Dispatchereingriffe mehr zum Zuge.

Achtung!

Dieser Befehl ist mit größter Sorgfalt anzuwenden und darf nur für sehr kurze Zeit (max. ca. 20 ... 50 Maschinenbefehle) zur Inhibierung der Interrupts führen. Sinn dieser Anweisung ist, bei bestimmten Problemen in E/A–Treibern Sequenzen von wenigen Befehlen unteilbar zu machen.

Der privilegierte Zustand wird in legalem T–Code durch Aufruf des DPC-Traps beendet.

In reinen 68k-Programmen kann auch der Befehl ANDI =$D8FF,SR verwendet werden, wenn ohnehin sofort irgendein anderer Trap von **RTOS–UH** folgt. Zwischenzeitliche Veränderungen von Taskzuständen können sich dadurch allerdings verzögert auswirken.

PENTR = $4E4B — Procedure entry

Eingaberegister:	`D1.L`	Nutzbare Workspace–Größe
	`A5.L`	Wird im alten Workspace gerettet
Ausgaberegister:	`A1.L`	Adresse des Workspaces
	`A5.L`	Zeiger auf erstes Nutzbyte
	`SR`	„`NE`“ kein Erfolg, „`EQ`“ `A1` geladen
Veränderte Register:	`D1,D7,A1,A5`	

Der von **RTOS–UH** verwaltete Speicherbereich wird von oben nach unten auf die erste freie Sektion durchsucht, in die eine Sektion der in `D1` angegebenen Größe samt ihrem Verwaltungskopf hineinpaßt. `D1` enthält also die effektiv nutzbare `WSP`–Größe. Die Suche beginnt dabei versuchsweise zunächst an der Stelle, an der beim letzten Mal erfolgreich Speicher zugeteilt werden konnte. Wenn das nicht gelingt, wird nach unten weitergesucht, und erst danach werden die oberhalb liegenden Freisektionen inspiziert. Während der Suche ist der Trap preemptionfähig.

Wird keine passende freie Sektion gefunden, so antwortet der Trap mit „`NE`“ anderenfalls mit „`EQ`“. Das Register `A5` wird in den Verwaltungskopf gerettet und anschließend ebenso wie `A1` neu geladen. Register `A1` zeigt auf die erzeugte Sektion, `A5` auf die Stelle in der Sektion, ab der der Anwender `D1` Datenbytes ablegen darf.

Die Umkehroperation hierzu ist der Trap „`RETN`“ (`$4E4C`). Man beachte, daß die so erzeugte Speichersektion als „`PWS`“, d. h. „Procedure–Work–Space“ verbucht wird und mit der Terminierung der einstmals erzeugenden Task automatisch wieder zu freiem Speicher wird. Dafür sorgt das sog. „T–link“, eine Kette, die ihren Ursprung im „Task–WorkSpace“ hat und alle von der Task angeforderten CEs (Communication–Elements) und „`PWS`“ miteinander zu einem Ring verbindet. Will man die Sektion von der Task ablösen, wie es zum Beispiel der Editor mit neuen Blöcken macht, so muß die Sektion mit einer besonderen Prozedur aus dem „T–link“ herausgenommen werden. In solchen Fällen empfiehlt sich allerdings nicht dieser Trap, sondern der Trap `WSBS` (`$A00C`).

Beispiel:

```
PENTR  OPD    $4E4B
RETN   OPD    $4E4C
      .INCLUDE  .../COMEQU.NOL  wegen PRTNAD *
       ...
       BSR    SUBR      Unterprogramm
       ...
*-----------------------------------------*
SUBR   MOVE.L =500,D1   500 Bytes Daten
       PENTR
       BNE    MIST      B: Kein Platz mehr
       MOVE.L (A7)+,PRTNAD(A1) Return-adr ableg.
       ...
       ...      Nun koennen die Bytes 0(A5) bis
       ...      incl 499(A5) benutzt werden
       ...
       RETN             Wsp zurueck+Jump
```

Hier die wahrscheinlichen `EQUs` für den Fall, daß die Datei `COMEQU` nicht zur Hand ist:

```
PRTNAD  EQU  $1A    Für die 68k-Familie
PRTNAD  EQU  $1C    Für die PowerPC-Familie
```

PIRTRI = kein TRAP — Prozeßinterrupt Triggern

Eingaberegister: `D1.L` Bit(s) des Prozeßinterrupts
Ausgaberegister: `-`
Veränderte Register: `D1`

Hierbei handelt es sich nicht um einen Trap, sondern um eine Linkzelle für einen Transfer in den Nukleus! Der Anschluß ist das hintere Ende einer Supervisorfunktion und darum nicht für den normalen Nutzer, sondern ausschließlich zur Systemerweiterung durch den Implementierer vorgesehen.

Über `PIRTRI` kann eine interrupterzeugende Hardware einen Prozeßinterrupt auslösen. Im `D1` wird die 32-bit Eventmaske übergeben, die festlegt, welche(r) der 32 Prozeßinterrupt „gefeuert“ werden soll(en). Diese Eventmaske entspricht genau derjenigen aus dem Systemteil von PEARL-Programmen in der Klammer des Schlüsselwortes „`EV(...)`“. Nur wenn der entsprechende Interrupt enabled ist, erfolgt eine Aktion.

Aus Gründen der Effizienz und wegen unterschiedlicher Kodierung der Rückfallmechanismen kommt eine Formulierung der Aufrufkonvention im T–Code nicht in Frage.
Prozessorspezifisch müssen genau die unten angegebenen Register gerettet werden!!

Beispiel 68k-Familie: Der Interrupt wurde auf `$100` „eingeklinkt“.

```
PIRTRI EQU  $80E   Adresse des Soft-IR  (besser: COMEQU)
IID    EQU  $7FE   Interrupt Identifier (besser: COMEQU)

       DC  irmal-irpt        mal-funktion
irpt   MOVE IID,-(A7)        save IID
       MOVE =$100,IID        new IID
       MOVEM.L D1/D6/D7/A1,-(A7) reg. -> Systemstack
       ...            Interrupt zuruecknehmen
       MOVE.L =$80000000,D1  event code setzen
       MOVEA.L PIRTRI,A1     Sprungadresse laden
       JMP  (A1)             Prozess-IR feuern
```

Prozeßinterrupt Triggern **PIRTRI (Forts.)**

Mit dem `JMP (A1)` wird in den Systemkern gesprungen, der überprüft, ob der entsprechende Prozeßinterrupt „enabled“ ist. Wenn dies der Fall ist und eine oder mehrere Tasks auf den oder die IR's eingeplant sind, werden sie aktiviert oder fortgesetzt. Die Interruptroutine wird vom Systemkern beendet, und es wird ein Dispatcherlauf forciert, um die Änderungen der Taskzustände wirksam werden zu lassen.

Bei der PowerPC-Hardware liegen die Verhältnisse sehr viel komplizierter. Der Prozessor selbst verfügt nur über einen sehr rudimentären Unterbrechnungsmechanismus, der durch äußere Spezialhardware unterstützt werden muß.

Wir studieren hier exemplarisch die Hardware der Motorola VME-Karte MVME1600, die weitgehend der ursprünglichen PowerPC Reference-Plattform entspricht: Alle Interruptsignale werden über den sog. `PIC`-Baustein (PC-Baustein von Intel!) im ISA-Bridge Controller (IBC) geleitet. Passend zu diesem Baustein enthält die Implementierungsscheibe eine Art „Interruptgateway“, das die Überleitung vom Hardware-IR-Slot des PowerPC zu 16 `PIC`-Jumpslots managt. (Wahrscheinliche Jumpslotadressen: `$4100+4*`Irno) Alle Interruptantwortroutinen müssen in dieses „Interruptgateway“ zurückkehren, da am Baustein noch komplizierte Endeoperationen ausgeführt werden müssen. Im Beispiel soll ein Hardwareinterrupt den Event mit der Kodierung `$02000000` auslösen. Die Ansprungadresse von `GENEV` muß dazu vorher auf den zum Interrupt gehörenden `PIC`-Jumpslot gebracht worden sein.

```
         .INCLUDE     .../FORM1600  Spezialformate MVME1600
          ...
GENEV    PIRIBC    $02000000        erledigt alles
```

Für andere Prozessorhardware sollten Sie bei uns nachfragen.

PIT = $A02E — Peripherie–Input

Eingaberegister: `D6.L`
Ausgaberegister: `D1.L`
Veränderte Register: `alle außer D6.L, A4.L bis A7.L`
(impl. abhängig)

Es wird eine binäre, ungepufferte Eingabe in implementierungsspezifischer Form durchgeführt. Die Adresse des peripheren Gerätes sowie evtl. eine Beschreibung der Zugriffsart sind in `D6.L` enthalten, `D1.L` enthält das eingelesene Datum. `D1.L` wird implementationsabhängig nicht in voller Länge eingelesen.

Der Trap ist nicht Bestandteil des Nukleus von **RTOS–UH**. Genauere Informationen entnehme man daher bitte dem Implementierungshandbuch des jeweiligen Systemes.

Peripherie–Output **POT = $A030**

Eingaberegister: `D6.L, D1.L`
Ausgaberegister: `-`
Veränderte Register: alle außer `D6.L, D1.L, A4.L` bis `A7.L` (implementationsabhängig)

Es wird eine binäre, ungepufferte Ausgabe in implementierungsspezifischer Form durchgeführt. Die Adresse des peripheren Gerätes sowie evtl. eine Beschreibung der Zugriffsart sind in `D6.L` enthalten, `D1.L` enthält das auszugebende Datum. `D1.L` wird implementationsabhängig nicht in voller Länge ausgegeben.

Der Trap ist nicht Bestandteil des Nukleus. Eine genauere Beschreibung ist darum dem jeweiligen Implementierungshandbuch zu entnehmen.

PREV = $A022 — Prevent Task by name

Eingaberegister:	`OPNAME.T`	Textadresse oder Text
	`A4.L`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D7,A1,A2`	

Eine Task, deren Name oder deren Adresse des Namens in `OPNAME.T = OPNAME(A4)` steht, wird ausgeplant. Auch im Aktivierungspuffer aufgelaufene Aktivierungen werden gelöscht. Ist die Task nicht im System vorhanden, wird mit einer entsprechenden Fehlermeldung reagiert und der Aufrufer suspendiert.

Während der Suche nach der Task ist der Trap preemptionfähig.

Beispiel:

```
PREV   OPD  $A022   Trap-Definition
       ...
       MOVE.L ='HALL',OPNAME.T  'HALL' nach OPNAME
      _MOVE  ='O ',OPNAME+4.T   'O '   nach OPNAME
       PREV                    Task 'HALLO' ausplanen
       ...
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPNAME`:

```
OPNAME EQU $66  beim 68K
OPNAME EQU $B4  beim PowerPC
```

Prevent quick **PREVQ = $A054**

Eingaberegister: `A1.L` Adresse der Task (`TID`)

Ausgaberegister: `-`

Veränderte Register: `D7,A1`

Eine Task, deren Adresse in `A1.L` steht, wird ausgeplant. Auch im Aktivierungspuffer aufgelaufene Aktivierungen werden gelöscht.

Der Trap prüft nicht, ob ihm in `A1` eine legale Task-ID übergeben wird.

Beispiel:

```
PREVQ   OPD  $A054  Trap-Definition
        ...         TID in A1
        PREVQ       Task ausplanen
        ...
```

QSA = $A01E — Quote–scan with answer

Eingaberegister:	A2.L	Adresse des zu untersuchenden Textes
Ausgaberegister:	A2.L	Bei „EQ“ um Stringlänge inkrementiert
	SR	„EQ“ falls gleich
Veränderte Register:	D7	
	PC	Überspringt folgendes Wort

Durch das PC–relative Wort hinter dem Trap wird die Adresse eines Strings übergeben. Der ASCII–String muß mit dem Zeichen $FE beendet sein. QSA prüft nun, ob der String mit der über (A2) erreichbaren Sequenz übereinstimmt. Kleinbuchstaben des (A2)–String werden versuchsweise in Großbuchstaben verwandelt, falls der Vergleich eines Zeichens nicht gelingt. Wenn bis zum $FE alle Zeichen des String mit der (A2)–Sequenz übereinstimmen, wird A2 auf das nächste Zeichen der Sequenz vorgerückt und CCR auf „EQ“ gesetzt. Stimmt auch nur ein Zeichen (trotz Kleingroßkonvertierung) nicht überein, so bleibt A2 unverändert, und CCR wird auf „NE“ gesetzt.

In jedem Fall werden die beiden auf den Trap folgenden Bytes bei der Rückkehr übersprungen, beim PowerPC zwei weitere Füllbytes.

Der Trap eignet sich für eine einfache Textanalyse und wird innerhalb der Shell eingesetzt. Er ist darum zur Realisierung neuer Bedienbefehle optimal geeignet.

Beispiel (T–Code):

```
QSA  OPD   $A01E   Trap-Definition
     ...           A2 zeigt auf Eingabetext
     QSA           Aufruf
     DC  TEXT-$    Stringadresse relativiert.
     BEQ Irgendwo  Springt bei erhoehtem A2 nach Irgendwo.
     ...
TEXT DC.B 'STIMMTS',$FE
```

Der BEQ wird ausgeführt, wenn A2 vor dem Trap z. B. auf einen Text der Form *Stimmtsblabla* oder *sTIMMtSxx* etc. zeigt. Nach dem QSA zeigt A2 dann z. B. auf *blabla* bzw. *xx*.

Read–Clock **RCLK = $A03E**

Eingaberegister: -
Ausgaberegister: D1.L Uhrzeit in Millisekunden
Veränderte Register: D7

Die aktuelle Uhrzeit wird gelesen und in D1 zurückgegeben. D1 enthält die Uhrzeit in Millisekunden. Prinzipiell kann man bei 68k-Systemen die Uhrzeit auch durch einen MOVEM-Befehl zum gleichzeitigen Lesen der Zellen TIME und TIMEB mit anschließender Addition der beiden Register ermitteln (siehe Seite 461 CLKASC-Trap). Der Trap sieht hier eine zum T–Code kompatible, sichere und multiprozessorkompatible Alternative vor.

Beispiel:

```
RCLK   OPD  $A03E      Trap-Definition
       ...
       RCLK            Uhrzeit lesen
       MOVE.L D1, ...  Uhrzeit speichern
       ...
```

RELCE = $4E49 — Release CE

Eingaberegister: `A1.L` Zeiger auf CE
Ausgaberegister: `-`
Veränderte Register: `D5,D6,D7,A1`

Das mit `A1` angegebene Communication–Element wird vom Besitzer freigegeben oder vom I/O-Dämonen zurückgegeben. Zum Verständnis dieses Traps ist es erforderlich, daß man sich mit dem besonderen I/O-Konzept von **RTOS–UH** und der CE–Philosophie vertraut macht. Dazu wird auf die Beschreibung des CEs auf Seite 559 hingewiesen.

Warnung!

Es ist unbedingt sicherzustellen, daß `A1` auf ein existierendes CE (Communication–Element) zeigt, da sonst u. U. das ganze System zum Absturz gebracht werden kann.

An Hand des Verwaltungszeigers `FORS` des in `A1.L` bezeichneten CEs wird zunächst geprüft, ob das CE zur Zeit in einen E/A–Vorgang verwickelt ist. Ist das nicht der Fall, d. h. ist `FORS=0`, so wird es sofort in freien Speicher umgewandelt, und der Trap kehrt zurück.

Wenn dagegen das CE in einer Warteschlange steht und noch nicht in Bearbeitung einer Betreuungstask (I/O–Dämon) ist, so wird lediglich das Release–Bit (`STABRE =` Bitno.1 in `STATIO`) gesetzt und der Trap verlassen.

Release CE **RELCE (Forts.)**

Im besonderen Fall (`FORS=$00000001`, (siehe auch Seite 534 `TOQ`)), bei dem das CE bereits in Bearbeitung durch eine Betreuungstask (I/O–Dämon) ist, wird nach folgender Testreihenfolge verfahren:

1. Ist die exekutierende Task Besitzer des CEs?
   In diesem Fall wird nur das Release–Bit in `STATIO` gesetzt, und der Trap kehrt zurück.
2. Ist im CE die Zelle `TIDO` gelöscht, d. h. gibt es keinen Besitzer mehr?
   Falls das so ist, wird das CE zu freiem Speicher, und der Trap kehrt zurück.
3. Ist dieser Testpunkt erreicht, wird fest davon ausgegangen, daß der `RELCE` von einem I/O–Dämon exekutiert wurde. Vor weiterer Untersuchung des CEs wird ggf. der auf die Fertigstellung der I/O–Operation wartende Besitzer des CEs lauffähig gemacht.
4. Ist das „Release–Bit“ (`STABRE =` Bitno.1) in `STATIO` gesetzt?
   Jetzt wird davon ausgegangen, daß der aufrufende I/O–Dämon alle Operationen mit dem CE abgeschlossen hat und dieses nunmehr zu vernichten ist. Das CE wird zu freiem Speicher, und der Trap kehrt zurück.
5. Ist das „Return–Bit“ (`STABRT =` Bitno.2) in `STATIO` gesetzt?
   In diesem Fall wird davon ausgegangen, daß der aufrufende I/O–Dämon (I/O–Task) das CE nach Abarbeitung in die CE–Schlange des Besitzers zurückgeben will. Das CE wird in die CE–Schlange des Besitzers eingekettet. Damit trotz der Verkettung erkennbar bleibt, daß dieses CE nicht mehr in eine I/O–Operation verwickelt ist, wird das „OwnQueue–Bit“ (`STABOQ =` Bitno.3) in `STATIO` gesetzt und der Trap verlassen.

RELEA = $4E47 — Release semaphore

Eingabe-Register: `A1.L` Adresse der Semavariable (2 Byte)
Ausgabe-Register: `-`
Veränderte Register: `D5,D6,D7`

Die angegebene Semaphorvariable (per Adresse in `A1`) wird um `1` erhöht. Je nach neuem Wert sind drei Fälle zu unterscheiden:

1. Der neue Wert ist größer als Null. Es erfolgt unmittelbar ohne weitere Aktion die Rückkehr aus dem Trap.
2. Der neue Wert ist Null, und bei einer mit der höchsten Priorität beginnenden Suche wird unter allen Tasks im Blockierzustand „`SEMA`“ eine gefunden, die genau auf diese Semavariable wartet. Diese Task wird entblockiert. Die Suche wird nicht fortgesetzt, die Semavariable erhält den Wert `-1`, damit mögliche weitere wartende Tasks später nicht „vergessen“ werden.
3. Der neue Wert ist Null, und es wurde keine Task entdeckt, die auf genau diese Semavariable wartet. Der Wert der Semavariablen wird auf `+1` gesetzt, und der Trap wird ohne weitere Aktion verlassen.

Man beachte, daß der „Außenwelt“ gegenüber der zeitweilig angenommene Wert `-1` gleichwertig zur Null („Requested“) ist und nur interne Bedeutung hat („Requested and task may be waiting“).

Beispiel:

```
RELEA  OPD   $4E47
       ...
       LEA   >SEMA4,A1   z.B. globale externe Semavariable
       RELEA
       ...
```

Request Semaphore **REQU = $4E46**

Eingabe-Register:	`A1.L`	Adresse der Semavariablen (2 Byte)
Ausgabe-Register:	`-`	
Veränderte Register:	`D7`	
im Task–WSP:	`OPFATI.T`	im TWS

Die angegebene Semavariable (per Adresse in `A1`) wird um `1` erniedrigt.

Ist der neue Wert größer oder gleich Null, so erfolgt die Rückkehr ohne weitere Aktion.

Ist der neue Wert jedoch negativ, so wird er auf `-1` korrigiert, und die anfordernde Task wird blockiert. Im Taskworkspace der anfordernden Task (`OPFATI(A4)`) wird eine Notiz hinterlassen, die später ggf. vom Trap `RELEA` aufgefunden werden kann. Innerhalb des Traps ist kein Dispatchereingriff möglich, so daß die notwendige Unteilbarkeit des Request gesichert ist. Obwohl in diesem Fall der Taskworkspace benutzt wird, braucht `A4` nicht besetzt zu sein. Der Trap holt sich die entsprechende Adresse des `TWS` auf andere Weise.

Man beachte, daß der „Außenwelt“ gegenüber der zeitweilig angenommene Wert `-1` gleichwertig zur Null („Requested“) ist und nur interne Bedeutung hat („Requested and task waiting“).

Beispiel:

```
REQU  OPD   $4E46
      ...
      LEA  >SEMA4,A1  Globale Semavariable
      REQU            Je nach vorherigem
                      Zustand wird die
                      Task evtl. blockiert
      ...
```

RESRB = $A072 — Reserve Boltvariable

Eingaberegister: `A1.L` Adresse der Boltvariablen
Ausgaberegister: `--`
Veränderte Register: `D7`

Der Trap ist Teil des Reserve-Konstruktes, das der PEARL-Compiler generiert. Er sollte nur mit dem entsprechenden Vorspann benutzt werden, da sonst trotz korrekter Funktion der Geschwindigkeitsvorteil der Boltvariablen verloren geht.

Die Anwendung **muß** im T–Code wie folgt aussehen:

```
RESRB  OPD    $A072     Trap-definition
       ...
       LEA    Boltx,A1  Adr of Bolt
       TAS    (A1)      Test and set
       BEQ.B  weiter    branch if it was free
       RESRB
weiter ....
```

Der Trap wird also nicht ausgeführt, wenn der Vorzustand der Boltvariablen „Free“ (=`$0000`) war. Durch den `TAS` wird dann als neuer Zustand „Reserved, nobody waiting“ (=`$8000`) in der Boltvariablen abgelegt. Der Transferassembler übersetzt genau diesen `TAS` in eine längere geschützte Sequenz auf Basis der Befehle `lwarx` und `stwarx`.

Wenn der Trap zur Ausführung kommt, so wiederholt er zunächst testweise den `TAS` (es könnte ja sein, daß die Boltvariable inzwischen freigeworden ist). Falls der `TAS` diesmal „`EQ`“ abliefert, wird der Trap ohne weitere Aktion verlassen. Ansonsten wird die Task blockiert und der Boltvariablen der Zustand „Reserver waiting“ aufgeordert, so daß als neue mögliche Zustände „x-times entered + reserver waiting“ oder „Reserved + reserver waiting“ entstehen können.

Man beachte, daß auch nach vergeblicher `RESRB`-Operation keine Enter-Operationen mehr möglich sind, bis die Boltvariable wieder frei ist.

Return from procedure **RETN = $4E4C**

Eingaberegister:	`A5.L`	Zeiger auf zuletzt eingerichteten Workspace
Ausgaberegister:	`A5.L`	Wird aus dem freigebenen `WSP` geladen
	`PC`	Wird aus dem freigegebenen `WSP` geladen
Veränderte Register:	`A1,D7`	

Der vorher mit `PENTR` eingerichtete `PWSP` (Procedure–WorkSPace) wird mit diesem Trap wieder freigegeben. Vorher wird allerdings das vom `PENTR` gerette Register `A5` zurückgeladen.

Wichtig!

Der Trap kehrt nicht zum Aufrufer zurück, sondern holt sich die Rückkehradresse ebenfalls aus dem Verwaltungskopf der mit dem Eingaberegister `A5` bezeichneten Speichersektion. Auf `PRTNAD`, erreichbar über `-8(A5)`, muß also unbedingt eine verwertbare Fortsetzungsadresse stehen. Der Trap `PENTR` schreibt dort nichts hin (s. Beispiel nächste Seite)!.

Der Trap wird in den Formaten `PROCEX`, `X8090` und `QX8090` benutzt. Der alte PEARL80-Compiler generiert bei jedem `RETURN-` Statement einen `RETN`.

RETN (Forts.) **Return from procedure**

Beispiel:

```
PENTR  OPD   $4E48
RETN   OPD   $4E4C
       ...
       BSR   SUBR          Unterprogramm
       ...
*-----------------------------------------------*
SUBR  _MOVE.L =500,D1      500 Bytes Daten
       PENTR
       BNE    MIST         B: Kein Platz mehr
       MOVE.L (A7)+,-8(A5) Return-adr ableg.
       ....
       ....   Nun koennen die Bytes 0(A5) bis
       ....   incl 499(A5) benutzt werden
       ....
       RETN                Wsp zurueck+Jump
       ...
MIST   ...                 Fehlermeldung ..
```

Vorsicht:

Die früher empfohlene Konstruktion mit `PRTNAD(A1)` statt `-8(A5)` erfordert das Inkluden der `COMEQU`-Datei, weil das Displacement `PRTNAD` in der PowerPC-Familie einen anderen Wert als in der 68k-Familie hat!

Restart Task, TWS new **RSTT = $A04A**

Eingaberegister:	`D1.L`	New Size of Task-Workspace
Ausgaberegister:	`A4.L`	Zeiger auf neuen Task-Workspace
Veränderte Register:	`A1,D5,D6,D7`	

Der Trap löscht zunächst einen evtl. Anschluß eines Exception-Handlers, d. h. `SIGLNK(TID)` wird auf Null gesetzt. Danach arbeitet er die komplette T-Link-Kette des Aufrufers ab und löscht jedes Element genauso, wie es der `TERMI`-Trap tut, d. h. Communication–Elemente werden wie beim `RELCE`–Trap abgelöst etc. Während der Abarbeitung der Kette ist der Trap preemptionfähig.

Nach dem Abräumen des T-Links wird schließlich der bisherige Task-Workspace zurückgegeben und durch einen neuen mit der in `D1.L` angegebenen Länge ersetzt. Ist dafür allerdings nicht genügend Platz verfügbar, so wird die aufrufende Task blockiert im Zustand „`PWS?`“, und der Versuch wird zu gegebener Zeit wiederholt. Vorläufig behält die Task dabei noch den bisherigen Task-Workspace.

Nach der Rückkehr aus dem Trap ist `A4` neu geladen. Alle im alten Task-Workspace bisher gespeicherten Daten sind verloren! Der Trap ermöglicht der aufrufenden Task, ihre gesamte Speichersituation neu zu organisieren. Sekundäre Shellprozesse benutzen ihn, z. B. wenn sich ein Compiler nach Abschluß seiner eigentlichen Arbeit um die kaum Workspace benötigenden Shellfolgeanweisungen kümmern muß.

RSTT (Forts.) — Reset T-link and new TWS

Man kann nur eine beschränkte Anzahl von Daten aus dem bisherigen „Leben“ der Task in das neue hinüberretten: Es sind die Register, die der Trap nicht verändert sowie notfalls einige Daten aus dem permanten Task-Header.

Wichtiger Hinweis!

Die in `D1.L` übergebene Länge muß den Verwaltungskopf des Task-WSP mit enthalten, außerdem muß ggf. der Platz zum Retten der FPU-Register vorgesehen werden. Der Trap setzt `A5` nicht neu auf und korrigiert auch die Zelle `WSPLEN` im permanenten Task–Head nicht.

Beispiel:

```
RSTT  OPD  $A04A          Trap-definition
      ...
     _MOVE.L  =....,D1    new TWSP-size
      RSTT                Ein neues Leben ...
```

Rubber Blanks **RUBBL = $A020**

Eingaberegister:	`A2.L`	Adresse des zu untersuchenden Textes
Ausgaberegister:	`A2.L`	Um Anzahl Blanks inkrementiert
Veränderte Register:	`D7`	

In einem Text können Leerzeichen überlesen werden. `A2` zeigt auf den Text und wird solange inkrementiert, wie Leerzeichen hintereinander im Text stehen. Dieser Trap kann bei der Realisierung von neuen Bedienbefehlen sehr nützlich sein. Er wird an vielen Stellen in der Shell eingesetzt.

Beispiel:

```
RUBBL  OPD  $A020  Trap-Definition
       ...         A2 zeigt auf Text
       RUBBL       Ueberliest Blanks
       ...         Weitere Textanalyse
```

RWSP = $A02A — Release Workspace

Eingaberegister: A1.L Adresse der zu löschenden Sektion
Ausgaberegister: -
Veränderte Register: D7

Die angegebene Speichersektion wird in freien Speicherraum zurückverwandelt. Anschließend darf A1 nicht mehr als Zeiger verwendet werden. Falls irgendwo Tasks auf die Zuteilung von „TWS“ (Task–WorkSpace) oder „PWS“ (Prozedur–WorkSpace) gewartet haben, erfolgt nunmehr möglicherweise ihre Freigabe — falls der Platzbedarf nun befriedigt werden kann.

Warnung!

Mit diesem Trap dürfen nur solche Sektionen gelöscht werden, die weder im „S“– (Dispatcher, I/O–Queue) noch im „T“–link (Task–link) eingekettet sind, schwere später auftretende Abstürze können sonst resultieren. So müssen z. B. mit WSFA, WSBS und WSFS eingerichtete „PWS“–Sektionen ausgelinkt werden, bevor dieser Trap eingesetzt werden darf. Das Auslinken ist z. B. beim WSBS auf Seite 543 beschrieben. Mit PENTR eingerichtete Sektionen werden dagegen nicht mit RWSP rückgegeben, sondern durch RETN (ohne vorheriges Auslinken!) gelöscht. Ebenso werden mit FETCE erzeugte CEs (Communication–Elemente) mit RELCE (ebenfalls ohne vorheriges Auslinken) getilgt.

Der Trap RWSP wird intern in den Traps RELCE und RETN benutzt, nachdem diese das Auslinken, sowie ihre Nebenoperationen erledigt haben. Auch innerhalb des UNLOAD–Befehles sowie im Trap TERV wird er benutzt. Der Editor verwendet RWSP beim „ERASE“.

Beispiel:

```
RWSP   OPD    $A02A
WSBS   OPD    $A00C
       ...
      _MOVE.L ...,D1  Wsp-Size
       WSBS           Wsp anfordern
       Auslinken!     siehe WSBS-Beschr.
       ...            Nun als 'Dauerblock' benutzbar
                      evtl auch von anderen Tasks
       RWSP           Sektion zurueckgeben.
```

Scanner Trap **SCAN = $4E45**

Eingaberegister:	`D7.L`	Nummer der Scheibe
	`A1,A6`	Nur bei Fortsetzungssuche
Ausgaberegister:	`A1.L`	ggf. Scheibenadresse
	`A6.L`	Intern für nächste Suche
	`D7.L`	Intern für nächste Suche
	`SR`	„`EQ`“ wenn gefunden, sonst „`NE`“
Veränderte Register:	`D5,D6,D7,A6`	

Es wird der Scan–Bereich, der beim Nukleus eingetragen ist oder der mit Hilfe einer 0-er Scheibe angeschlossen wurde (siehe Seite 637), nach der Scheibe, deren Nummer in `D7` übergeben wird, durchsucht.

Wenn die Scheibe gefunden wird, antwortet der Trap mit „`EQ`“, und man erhält in `A1` die Adresse nach der Signalmarke. Nun kann man `A1` auswerten –sollte aber die Registerinhalte `D7.L`, `A1.L` und `A6.L` dabei nicht verändern. Nachdem man `A1` ausgewertet hat, kann man den Trap erneut aufrufen, dank der in `D7`, `A1` und `A6` geretteten Daten setzt er nun die Suche fort und antwortet ggf. wieder mit „`EQ`“ etc.

Wird die Scheibe nicht, bzw. wird keine weitere derartige mehr gefunden, so antwortet der Trap mit „`NE`“.

Beispiel:

```
SCAN   OPD  $4E45   Trap-Definition
       ...
      _MOVEQ =1,D7  Scheibe 1
LABEL  SCAN          suchen
       BNE.S READY   B: nichts zu finden
       ...           Action, aber A1, A6, D7
       ...           bleiben unveraendert
       BRA.S LABEL   Fortsetzung der Suche
READY  ...           hier geht's weiter...
```

STBCLK = $A05E — Set Battery Clock

Eingaberegister:	`D1`	Date oder Marker für Zeit
	`D2.L`	wenn `D1 = 0`, Zeit in msec
Ausgaberegister:	`-`	
Veränderte Register:	`D7`	

Dieser Trap wird beim `CLOCKSET`– und `DATESET`–Kommando der Shell exekutiert. Im Nukleus des Systemes liegt auf dem Trap zunächst nur eine Leeroperation. Vom Implementierer oder vom Nutzer kann allerdings ein passender eigener Trap angeschlossen werden, um eine eventuell vorhandene Hardware–Uhr zu stellen.

Beim `CLOCKSET` ist das Register `D1` mit Null und `D2.L` mit der Uhrzeit in Millisekunden gesetzt.

Beim `DATESET` enthält `D1` das Datum „BCD-kodiert“:

30.8.1994 `-> D1 = $30081994`

Suspend actual running Task **SUSP = $A028**

Eingaberegister: -

Ausgaberegister: -

Veränderte Register: D7,A1

Die gerade laufende Task wird suspendiert. Bis auf D7 und A1 bleiben alle Register und Speichersektionen der Task erhalten, so daß die Task zu einem späteren Zeitpunkt fortgesetzt werden kann. Sie kehrt also erst nach fremder Hilfe wieder aus dem Trap zurück.

Die Traps CON und CONQ sind geeignet, um mit Hilfe anderer Tasks diese Suspendierung wieder aufzuheben.

Beispiel:

```
SUSP   OPD  $A028  Trap-Definition
       ...         Task laeuft
       SUSP        Task suspendiert sich
       ...
```

TERME = $A000 — Terminate external

Eingaberegister: `OPNAME.T` Textadresse oder Text
`A4.L` muß auf Taskworkspace zeigen
Ausgaberegister: `-`
Veränderte Register: `D1,D5,D6,D7,A1,A2`

Die Task, deren Name oder deren Zeiger auf dem Namen in `OPNAME.T = OPNAME(A4)` steht, wird in der Speicherverwaltung gesucht. Wird die Task nicht gefunden, so wird die Meldung „`... not loaded (terminate)`“ abgesetzt, und es erfolgt keine Operation.

Ist die Task inaktiv (nicht im Dispatcherring), so kehrt der Trap ohne weitere Aktion zurück.

Ist die zu terminierende Task identisch mit der den Trap ausführenden Task, so verwandelt der Trap sich in den `TERMI`–Trap.

In allen anderen Fällen wird die adressierte Task lauffähig gemacht, nachdem vorher der Fortsetzungs-PC und die Laufpriorität wie folgt manipuliert wurden:

- Der neue PC zeigt auf einen `TERMI`–Trap innerhalb des Nukleus, wobei vorher noch der Code `$00000001` nach `MSGLNK` (Zieltask-TID) gebracht wird. Damit wird evtl. wartenden Tasks (siehe `WFEX` auf Seite 539) das irreguläre Ende mitgeteilt.
- Die adressierte Task erhält eine neue Laufpriorität, die identisch ist zur Priorität der diesen Trap ausführenden Task. Allerdings wird die Task im Dispatcherring vor die den Trap ausführende eingekettet.

Beispiel:

```
TERME OPD  $A000   Trap-Definition
      ...
      MOVE.L ='REGL',OPNAME.T  'REGL'
      MOVE   ='ER',OPNAME+4.T  'ER'  (6 Bytes)
      TERME                    'REGLER' terminieren
      ...
```

Terminate quick **TERMEQ = $A058**

Eingaberegister: `A1.L` Adresse der Task (`TID`)

Ausgaberegister: `-`

Veränderte Register: `D1,D5,D6,D7,A1`

Dieser Trap ist der hintere Teil von `TERME`, wobei jedoch die angesprochene zu beendende Task nicht gesucht wird, sondern schon in `A1` übergeben wird.

Wie beim `TERME` so wird auch hier der PC und die Priorität der angesprochenen Task so manipuliert, daß sie auf einen `TERMI` läuft.

Warnung!

Es ist unbedingt sicherzustellen, daß `A1` im Moment des Aufrufes wirklich der Task–ID einer existierenden Task ist. Der Trap selbst prüft dies aus Zeitgründen nicht.

Auf evtl. bestehende Einplanungen der adressierten Task nimmt der Trap keinen Einfluß. Falls bei der noch gepufferte Aktivierungen vorliegen, so werden diese wie beim `TERMI` üblich nachgeholt.

Beispiel:

```
TERMEQ  EQU  $A058  Trap-Definition
        ...        TID nach A1.L
        TERMEQ     Terminate Task
        ...
```

TERMI = $4E41 — Terminate internal

Eingaberegister: -

Ausgaberegister: -

Veränderte Register: alle

Eine Task, die diesen Befehl exekutiert, wird sofort beendet. Die Registerinhalte werden nicht gerettet und gehen somit in jedem Fall verloren. Bei einer nichtresidenten Task werden alle Speichersektionen (PWS, TWS, CE) an das Betriebssystem zurückgegeben. Residente Tasks behalten ihren Taskworkspace. Die Ausgabe–CE's verbleiben in der Schlange, während Eingabe–CE's, die nicht schon von der E/A–Betreuungstask bearbeitet werden, aus der Schlange entfernt werden.

Eingeplante Tasks verbleiben auch nach dieser Anweisung im Ring des Prozeßumschalters (Dispatcher). War bereits eine neue Aktivierung gepuffert, so wird die Task der gepufferten Aktivierung entsprechend mit ggf. anderer Priorität erneut lauffähig gemacht.

Wenn es andere Tasks gibt, die mit WFEX (siehe Seite 539) auf das Ende dieser Task warten, so werden diese entblockiert. Dabei wird ihnen der Inhalt von MSGLNK auf deren eigene MSGLNK–Zelle kopiert.

Während des sukzessiven Abbaues der (evtl. sehr zahlreichen) Speichersektionen ist eine Prozeßumschaltung möglich (Preemption).

Terminate and vanish **$TERV = $A010**

Eingaberegister: -
Ausgaberegister: -
Veränderte Register: alle

Es werden die gleichen Aktionen ausgeführt wie beim TERMI ($4E41). Zusätzlich verschwindet die Task aus dem System, falls sie nicht eingeplant ist oder noch Aktivierungen gepuffert sind. Eine verschwundene Task kann natürlich nicht mehr aktiviert werden.

Dieser Trap kann dazu genutzt werden, um einen Sohnprozeß, der mit GAPST erzeugt wurde, nach Erledigung seiner Aufgabe aus dem System zu entfernen.

Beispiel:

```
TERV  OPD  $A010  Trap-Definition
      ...         Code des Sohnprozesses
      ...
      TERV        Ende des Sohnprozesses
```

Hinweis:

Der Trap sollte bei Sohnprozessen der Shell nicht benutzt werden. Es würden dann nämlich keine Nachfolgekommandos mehr ausgeführt, und es würde an den übergeordneten Prozeß keine Antwort („gelungen“ oder „mißlungen“) zurückgegeben. (Zur Codierung shellkonformer Sohnprozesse in Assemblersprache benötigen Sie entsprechende Extrainformationen.)

TIAC = $A016 — Time–schedule activation

Eingaberegister:	`D1.W`	Priorität
	`OPNAME.T`	Textadresse oder Text
	`OPFATI.T`	Startzeit
	`OPINTV.T`	Intervall
	`OPLTI.T`	Endzeit
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D1,D6,D7,A1,A2`	

Eine Task, deren Name oder deren Namensadresse in `OPNAME.T = OPNAME(A4)` steht, wird zur Aktivierung eingeplant. In `D1` wird die Priorität der Aktivierung übergeben. Ist `D1` gelöscht, wird die Default–Priorität eingetragen. Läuft die Task bereits, so bleibt die aktuelle Priorität unbeeinflußt.

In `OPFATI(A4)` (First Activation Time) wird entweder (gesetztes Vorzeichenbit) eine relative Verzögerungszeit in Millisekunden oder (ohne gesetztes Vorzeichenbit) der absolute Zeitpunkt für die erste Aktivierung vorgegeben. Liegt ein absolut angegebener erster Aktivierungszeitpunkt vor der aktuellen Systemuhrzeit, so wird die Task zur vorgegebenen Uhrzeit, jedoch erst am folgenden Tag, zur Ausführung eingeplant.

Mit `OPINTV(A4)` (Intervall) wird der Abstand zyklischer Aktivierungen vorgegeben. Ist der Wert negativ oder Null, findet keine zyklische Einplanung statt.

Mit `OPLTI(A4)` (last time) wird der letzte Aktivierungszeitpunkt zyklischer Einplanungen vorgegeben. Bei gesetztem Vorzeichenbit ist er relativ zum Istzeitpunkt, sonst absolut. Als Endlosindikator dient der größtmögliche positive Wert im obersten Byte.

Bestehende andere Einplanungen zur Aktivierung – auch solche auf externe Interrupts – werden gelöscht. Die Fehlermeldungen entsprechen den beim Trap `ACT` beschriebenen.

Time–schedule activation **TIAC (Forts.)**

Der Trap überträgt die aus den übergebenen Parametern errechneten Ergebnisse in den Taskdescriptionblock (Zellen `TIA`, `TINV` und `TIL`). Ein Überschreiben der Eingabeparameter von `TIAC` nach dessen Aufruf beeinflußt folglich bestehende Einplanungen in keiner Weise.

Beispiel:

```
TIAC  OPD $A016                    Trap-Definition
      ...                          TID in A1
      MOVE.L ='Test',OPNAME(A4)    Name
      MOVE.L ='xy',OPNAME+4(A4)     ''
      MOVE.L =$80002710,OPFATI(A4) after 10 sec
      MOVE.L =1000,OPINTV(A4)      all 1 sec
     _MOVE.B =$7F,OPLTI(A4)        Endlos-Indikator
      TIAC                         nach 10 Sek alle 1 Sek
      ...
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für obige Displacements:

`OPNAME EQU $66` und `OPFATI EQU $6C` beim 68K
`OPINTV EQU $70` und `OPLTI  EQU $74` beim 68K

`OPNAME EQU $B4` und `OPFATI EQU $BC` beim PowerPC
`OPINTV EQU $C0` und `OPLTI  EQU $C4` beim PowerPC

TIACQ = $A024 — Time–schedule activation quick

Eingaberegister:	`D1.W`	Priorität
	`A1.L`	Adresse der Task (`TID`)
	`OPFATI.T`	Startzeit
	`OPINTV.T`	Intervall
	`OPLTI.T`	Endzeit
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D1,D6,D7,A1,A2`	

Eine Task, deren Task-ID (Adresse des Task-DCB) in `A1` steht, wird zur Aktivierung eingeplant. Alle anderen Parameter entsprechen vollständig denen des Traps `TIAC`. `TIAC` läuft nämlich nach der Task-Suche als `TIACQ` weiter.

Bitte lesen Sie auf Seite 528 bei `TIAC` nach.

Beispiel:

```
TIACQ  OPD  $A024                   Trap-Definition

       LEA    TDCBXY,A1             TID -> A1
       MOVE.L =$80002710,OPFATI(A4) after 10 sec
       MOVE.L =1000,OPINTV(A4)      all 1 sec
      _MOVE.B =$7F,OPLTI(A4)        ohne Ende
       TIACQ                        nach 10 Sec aktivieren
       ...
```

Wie bei allen Quick-versionen eines Trap, so muß auch hier sichergestellt werden, daß im Register `A1` wirklich ein gültiger Task-ID (`TID`) steht!

Continue by time–schedule **TICON = $A018**

Eingaberegister: `OPNAME.T` Textadresse oder Text
`OPFATI.T` Zeitdauer oder Zeitpunkt
`A4.L` muß auf Taskworkspace zeigen
Ausgaberegister: `-`
Veränderte Register: `D6,D7,A1,A2`

Eine Task, deren Name oder deren Adresse des Namens in `OPNAME.T = OPNAME(A4)` steht, wird zur Fortsetzung eingeplant. Bestehende Fortsetzungseinplanungen werden gelöscht. Einplanungen zur Aktivierung bleiben aber unbeeinflußt.

Wenn die Task nach einer bestimmten Zeitdauer fortgesetzt werden soll, muß in `OPFATI.T = OPFATI(A4)` die Zeitdauer in Millisekunden mit gesetztem obersten Bit eingetragen sein.

Soll die Fortsetzung zu einem bestimmten Zeitpunkt erfolgen, darf das oberste Bit nicht gesetzt sein. Liegt unter Berücksichtigung der Systemuhrzeit ein vergangener Aktivierungszeitpunkt vor, erfolgt die Taskeinplanung zur angegebenen Uhrzeit am folgenden Tag.

Beispiel:

```
TICON  OPD  $A018          Trap-Definition
       ...
       LEA  TSKNAM,A0      Adresse des Tasknamens
       MOVE.L A0,OPNAME.T  Eintrag der Adresse
       CLR  OPNAME+4.T     kein Text
      _MOVE.L =$800003E8,OPFATI.T 1 sec Zeitdauer
       TICON               nach 1 Sec fortsetzen
       ...
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für obige Displacements:

`OPNAME EQU $66` und `OPFATI EQU $6C` beim 68K
`OPNAME EQU $B4` und `OPFATI EQU $BC` beim PowerPC

TICONQ = A04E — Time continue quick

Eingaberegister:	`A1.L`	Adresse der Task (`TID`)
	`OPFATI.T`	Zeitdauer oder Zeitpunkt
	`A4.L`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D6,D7,A1,A2`	

Eine Task, deren Adresse in `A1` steht, wird zur zeitgesteuerten Fortsetzung eingeplant. Bisher bestehende Einplanungen zur Fortsetzung werden gelöscht. Das gilt auch für ereignisgesteuerte Fortsetzungseinplanungen, jedoch bleiben evtl. Einplanungen zur Aktivierung unangetastet.

Wenn die Task nach einer bestimmten Zeitdauer fortgesetzt werden soll, muß in `OPFATI.T = OPFATI(A4)` die Zeitdauer in Millisekunden mit gesetztem obersten Bit eingetragen sein. Soll die Fortsetzung zu einem bestimmten Zeitpunkt erfolgen, darf das oberste Bit nicht gesetzt sein, und Taskeinplanungen, die unter Berücksichtigung der aktuellen Systemuhrzeit in der Vergengenheit liegen, werden zur entsprechenden Uhrzeit des nächsten Tages vorgenommen.

Beispiel:

```
TICONQ  OPD  $A04E     Trap-Definition
        ...            ...
        LEA    TSDCBXY,A1  TID -> A1
       _MOVE.L =$800003E8,OPFATI.T 1 sec Zeitdauer
        TICONQ         nach 1 Sec fortsetzen
        ...
```

Der Trap ist ein Zwischeneinstieg in das hintere Ende des Traps `TICON`.

Time resume **TIRE = $A02C**

Eingaberegister:	`OPFATI.T`	Zeitdauer oder Zeitpunkt
	`A4`	muß auf Taskworkspace zeigen
Ausgaberegister:	`-`	
Veränderte Register:	`D6,D7,A1`	

Die Task, die diesen Trap absetzt, wird suspendiert und nach der angegebenen Zeit oder zu einem bestimmten Zeitpunkt fortgesetzt. Der Trap besteht quasi aus einer untrennbaren Einheit von Suspendierung und Selbsteinplanung.

Irgendwelche bestehenden Einplanungen zur Fortsetzung werden gelöscht. Jedoch bleiben Einplanungen zur Aktivierung unangetastet.

Wenn die Task nach einer bestimmten Zeitdauer fortgesetzt werden soll, muß in `OPFATI.T = OPNAME(A4)` die Zeitdauer in Millisekunden mit gesetztem obersten Bit stehen. Andernfalls wird die eingetragene Zeit als Zeitpunkt interpretiert, und die Task zur Fortsetzung eingeplant. Liegt der angegebene Fortsetzungszeitpunkt vor der aktuellen Systemuhrzeit, erfolgt eine Einplanung zum angegebenen Zeitpunkt des folgenden Tages.

Beispiel:

```
TIRE   OPD  $A02C   Trap Definition
       ...
      _MOVE.L =$80000008,OPFATI.T 8 msec Zeitdauer
       TIRE         nach 8 msec fortsetzen
       ...
      _MOVE.L =1000*60*60*23,OPFATI.T  23 Uhr in msec
       TIRE   Fortsetzung um 23:00:00.000 Uhr
```

Falls die Datei `COMEQU` nicht zur Hand ist, hier die wahrscheinlichen Werte für `OPFATI`:

`OPFATI EQU $6C` beim 68K
`OPFATI EQU $BC` beim PowerPC

TOQ = $4E4D — Take of queue

Eingaberegister: `--`
Ausgaberegister: `A1.L,PC`
Veränderte Register: `A1,D7`

Mit diesem Trap wird versuchsweise ein Communication–Element aus der CE–Schlange der aufrufenden Task geholt. Ist die Warteschlange leer, so wird die Exekution mit dem nächsten folgenden Befehl fortgesetzt. Ist die Warteschlange nicht leer, so wird `A1` mit dem Zeiger auf das vorne — am Kopf der Schlange — stehende Communication–Element geladen. Vor der Rückkehr wird jetzt der `PC` um `2` (68k) bzw. um `4` (PowerPC) erhöht, so daß der nächstfolgende Einwortbefehl übersprungen wird. Das Element ist danach ausgekettet, und sein Vorwärtszeiger `FORS` steht auf `$00000001`, um die in „`RELCE`“ und „`IOWA`“ enthaltenen Abfragen korrekt vorzubereiten. Außerdem wird das „OwnQueue-Bit“ (`STABOQ =` Bitno.3) in `STATIO` zurückgesetzt, falls es gesetzt war.

In älteren Systemversionen mußte noch das Register `D1` als Eingangsparameter versorgt werden. Dies ist nun nicht mehr erforderlich. Auf den alten Trap fußende Programme brauchen aber nicht geändert zu werden.

Der Trap ist das Gegenstück zum `XIO` und `MSGSND` (siehe dazu `XIO` auf Seite 549 und `MSGSND` auf Seite 498). `XIO` bringt ein Communication–Element in die CE–Warteschlange einer I/O–Task (I/O–Dämon). `MSGSND` funktioniert gleichartig, jedoch mit jeder beliebigen Task als Ziel. Die typische Anwendung von `TOQ` ist innerhalb der I/O–Dämonen. Aber auch jede andere Task, die Botschaften oder rückkehrende eigene CEs erwartet, kann diesen Trap sinnvoll nutzen.

Take of queue **TOQ = (Forts.)**

Vorsicht!

Die frühere typische Benutzung mit einem weiteren Trap unmittelbar hinter `TOQ` ist kein legaler T–Code, weil damit implizit die Codelänge des Folgetraps eingearbeitet war. Wenn alte Assemblerprogramme mit solchen Konstrukten auf den PowerPC übertragen werden sollen, so müssen sie unbedingt auf die im Beispiel angegebene Form umgestellt werden. Der Folgetrap wird dabei durch einen `BRA.B` ersetzt.

Beispiel (T–Code):

```
TOQ    EQU   $4E40
TERMI  EQU   $4E41
       ...
TAKE   TOQ           CE aus Schlange holen
       BRA.B TERMEX  2/4 byte langer Sprung wenn Schlange leer
       ...           Verarbeitung des CE
       BRA   TAKE    Hole naechstes
       ...
TERMEX TERMI         Terminate
```

Hinweis: Sollte nach dem vergeblichen `TOQ` während des Sprunges nach `TERMEX` doch noch ein CE auflaufen, so wird eine Taskaktivierung in den Puffer geschrieben und der `TERMI` bewirkt sofort einen Neustart der Task.

TOV = $4E4E — To virtual

Eingaberegister: -
Ausgaberegister: -
Veränderte Register: D2--D7,A0--A3,A6

Im Nukleus ist hier zunächst ein „wrong opcode“ angeschlossen. Erst durch die Hyperprozessor-Scheibe wird der Trap benutzbar. Er dient zum Umschalten des Prozessors in den virtuellen Hyperprozessor-Mode.

Bitte beachten!

Ein bedeutend schnelleres Einschalten des Hyperprozessors ist durch die folgende T–Code Befehlsfolge möglich.

```
HYPLNK   EQU $8CA
...
MOVEA.L  HYPLNK,A6
XJSR     (A6)
```

Man beachte, daß in der PEARL-Laufzeitwelt A6 sowieso schon mit dem Inhalt der Zelle HYPLNK permanent geladen ist.

Der Trap wird nur noch aus Gründen der Kompatibiltät zu alten S-Records angeboten.

Die wahrscheinlichen Adressen von HYPLNK:
HYPLNK EQU $8CA in der 68k-Familie
HYPLNK EQU $51A4 in der PowerPC-Familie

Trigger Event **TRIGEV = $A026**

Eingaberegister: `D1.L` Interrupt–Maske
Ausgaberegister: `-`
Veränderte Register: `D7`

Alle Interrupts, die durch ein gesetztes Bit in `D1.L` beschrieben werden und die „enabled“ sind, werden gefeuert. Damit werden alle Tasks, die auf die entsprechenden Interrupts zur Aktivierung oder Fortsetzung eingeplant sind, freigegeben. Im System wird nicht unterschieden, ob ein Interrupt von der Hardware oder dem `TRIGEV`–Trap ausgelöst wurde. Der Trap benutzt innerhalb des Nukleus exakt den gleichen Mechanismus, wie er bei `PIRTRI` (siehe Seite 502) zum Einsatz kommt. Somit ist es also möglich, Hardware–Interrupts zu simulieren und ihre Wirkung zu testen.

Beispiel (T–Code):

```
TRIGEV  OPD  $A026   Trap-Definition
        ...
       _MOVEQ =1,D1  EV 1
        TRIGEV       trigger EV 1
        ...
```

TRY = $A07A — Try Semaphore

Eingaberegister:	A1	Adresse der Semaphore
Ausgaberegister:	D0.L	logisches Resultat
	SR	„EQ“ oder „MI“
Veränderte Register:	D0, D7	

Der Trap arbeitet ähnlich wie REQU. Er führt jedoch in keinem Fall zu einer Blockierung des Aufrufers.

Wenn die Semaphore „frei“ ist („requestable“), dann wird sie um 1 erniedrigt und der Trap anwortet mit „MI“. In D0.L steht das Datum $FFFF8000. Dies entspricht der Registerwertigkeit eines PEARL-Bit(1) Objektes ’1’B1.

War die Semaphore dagegen „belegt“ („rot“), so bleibt sie unverändert und der Trap antwortet mit „EQ“. In D0.L steht das Datum $00000000. Dies entspricht der Registerwertigkeit eines PEARL-Bit(1) Objektes ’0’B1.

TRY wird vom PEARL-Compiler beim gleichnamigen PEARL90-Konstrukt generiert.

Beispiel (T–Code):

```
TRY     OPD  $A07A    Trap-Definition
        ...
        LEA  Sema1,A1 Adresse der Sema
        TRY           Versuche request
        BEQ  Nein     Bei branch: Sema unver"andert
        ...           Nun im kritischen Pfad
```

Wait for exit **WFEX = $A06E**

Eingaberegister:	`A1.L`	Task–ID to wait for
Ausgaberegister:	`SR`	Performance indication
	`...(TID)`	`MSGLNK` wird verändert
Veränderte Register:	`D6,D7`	

Zunächst wird untersucht, ob in `A1` die TID einer Task aus dem Dispatcherring steht. Während dieser Suche ist der Trap preemptionfähig.

Ist `A1` kein gültiger Task–ID aus dem Dispatcherring, so wird der Trap mit dem Status „`NE`“ verlassen. Vorher wird als Report-code das Datum `$FFFFFFFF` auf `MSGLNK(TID)` im Taskdescriptionblock (Task-DCB im permanenten Task–Head) des Aufrufers abgelegt. Weitere Aktionen unterbleiben.

Wenn (was der Normalfall sein sollte) `A1` im Dispatcherring gefunden wurde, so wird die Blockierbedingung „`waiting for activation`“ bei der durch `A1` adressierten Task aufgehoben, falls sie gesetzt war. Die den Trap aufrufende Task wird blockiert im Zustand „`SEMA`“. Der mit `A1` adressierten Task wird das mitgeteilt, damit bei deren Ende (exit) der `TERMI`– bzw. `TERME`–Trap die Entblockierung ausführt.

Mit der freiwilligen oder gewaltsamen Beendigung der durch `A1` beschriebenen Task wird der Aufrufer dieses Traps wieder lauffähig und erhält sowohl über `SR` als auch über seine eigene Zelle `MSGLNK(TID)` einen Report. Das Statusregister `SR` ist im Zustand „`EQ`“ um anzuzeigen, daß tatsächlich ein Wartezustand eingenommen wurde – im Gegensatz zum Fall bei dem `A1` auf eine inaktive Task zeigt oder gar ungültig ist. Bezüglich `MSGLNK(TID)` sind zwei Fälle zu unterscheiden:

WFEX (Forts.) **Wait for exit**

- Die mit `A1` bezeichnete Task beendet sich selbst. In diesem Fall überträgt sie den Inhalt ihrer eigenen Zelle `MSGLNK(A1)` in `MSGLNK(TID)` des Aufrufers. Wenn nichts besonderes passiert ist, wird das der Code `$00000000` sein. Es ist aber möglich daß der mit `A1` adressierte Prozeß hier vorher eine Botschaft für denjenigen, der auf ihn wartet, deponiert hat. Dabei sind die Muster `$FFFFFFFF`, `$00000000` ... `$00000010` bereits mit fester oder reservierter Bedeutung belegt. Ein irreguläres Ende kann durch Ablage des Codes `$00000001` dem Wartenden mitgeteilt werden.
- Die mit `A1` bezeichnete Task wird von jemandem mit `TERME` beendet, z. B. durch das „`UNLOAD`“-Kommando der Shell. In diesem Fall wird das vorzeitige irreguläre Ende automatisch durch den Eintrag des Datums `$00000001` in `MSGLNK(TID)` angezeigt, eine evtl. vorher dort deponierte Botschaft wird überschrieben.

Der Trap eignet sich besonders, um logische Warteketten im Zusammenhang mit Sohnprozessen aufzubauen. Siehe dazu auch die Beschreibung des GAPST–Traps auf Seite 481. Scheitert ein Sohnprozeß bzw. wird er abgebrochen, so wird bei richtiger Benutzung die ganze Wartekette rückwärts abgebaut. Die Shell benutzt intern diesen Trap im Zusammenhang mit dem WAIT–Befehl.

Beispiel:

```
TID   EQU  $802
GAPST OPD  $A00E
WFEX  OPD  $A06E
      ...          Set up parameters for GAPST
      GAPST        Create son process -> A1
      ...          Final alignment of son process
      WFEX         Wait for process in A1
      BNE   ...    branch if no waiting
      MOVEA.L TID,A1     own Task-ID
      TST.L   MSGLNK(A1) inspect report
      BEQ     allfine
      ...
```

Falls die Datei COMEQU nicht zur Hand ist, hier die wahrscheinlichen Displacementwerte für MSGLNK:

```
MSGLNK  EQU  $48   68k-Familie
MSGLNK  EQU  $48   PowerPC-Familie
```

WSBS = $A00C — Workspace backward search

Eingaberegister:	`D1.L`	Größe des angeforderten Workspace
Ausgaberegister:	`A1.L`	Adresse der Speichersektion
	`CCR`	„`NE`“ kein Erfolg, „`EQ`“ `A1` geladen
Veränderte Register:	`D1,D7`	

Der von **RTOS–UH** verwaltete Speicherbereich wird von oben nach unten nach dem ersten freien Stück abgesucht, in welches die Anforderung hineinpaßt. Zunächst wird an der Stelle begonnen, an der beim letzten Mal erfolgreich Speicher zugewiesen wurde. Erst wenn es darunter keinen freien Platz gibt, werden auch die oberen freien Sektionen inspiziert. Der untere nicht benötigte Rest der gefundenen Freisektion wird in eine neue kleinere Freisektion verwandelt. `A1` wird so geladen, daß es auf den (nach oben bündig liegenden) zugeteilten Bereich zeigt. Das Statusregister `CCR` wird zur Rückantwort benutzt. Wenn nämlich kein Platz gefunden wurde, so antwortet der Trap mit „`NE`“, anderenfalls (wenn `A1` geladen werden konnte) mit „`EQ`“.

Man beachte, daß die so erzeugte Speichersektion als „`PWS`“, d. h. „Procedure–WorkSpace“ verbucht wird und mit der Terminierung der einstmals erzeugenden Task automatisch wieder zu freiem Speicher wird. Dafür sorgt das sog. „T–link“, eine Kette, die ihren Ursprung im „Task–WorkSpace“ hat und alle von der Task angeforderten CEs (Communication–Element) und „`PWS`“ miteinander zu einem Ring verbindet. Will man die Sektion von der Task ablösen, wie es zum Beispiel der Editor mit neuen Blöcken macht, so muß die Sektion mit einer besonderen Prozedur aus dem „T–link“ herausgenommen werden, die unten erläutert ist.

Der mit `D1.L` angeforderte Speicher kann wegen des T–links erst ab dem Displacement `WLOLD` benutzt werden!

Workspace backward search **WSBS (Forts.)**

Beispiel (T–Code):

```
WSBS  OPD    $A00C
     _MOVE.L =$1000,D1           Sektion 4 kB brutto
      WSBS
      BNE    MIST                B: kein Platz mehr
*---- Nur wenn 'Dauerblock' gewuenscht wird: auslinken!
      OFF                        unteilbare Sequenz
      MOVEA.L BACKT(A1),A2       A2 nur als Beisp.
      MOVE.L FORT(A1),FORT(A2)   linker Nachb.
      MOVEA.L FORT(A1),A2
      MOVE.L BACKT(A1),BACKT(A2) rechter Nachb.
      MOVE   =$0010,TYPE(A1)     als 'MODULE' ausg.
      MOVE.L ='Mod1',NAME(A1)    Modulname
     _MOVE   ='23',NAME+4(A1)    Modulname 6 Bytes
      DPC                        back to user-mode
```

WSBS (Forts.) **Workspace backward search**

Die symbolischen Displacements müssen im T–Code aus der Datei `COMEQU` inkluded werden. Sie unterscheiden sich zwischen der 68k- und der PowerPC-Familie. Gleich sind jedoch `NAME=$0A` und `TYPE=8`. Eine so erzeugte Speicherdauersektion kann nur mit `RWSP` oder den Bedienbefehl `UNLOAD` *name* wieder eliminiert werden. Bei ausgelinkten Sektionen können ab Displacement `$10` eigene Daten abgelegt werden.

Der Trap `WSBS` ist ein Bruder der Traps `WSFA`, `WSFS` und `PENTR`, die ähnliche Grundeigenschaften aufweisen. Wie diese, so ermöglicht auch `WSBS` jederzeit während der Suche eine Taskumschaltung (Preemption).

Eine verantwortungsvolle und sorgfältige Anwendung versteht sich von selbst!

Hier für den Notfall (`COMEQU` nicht zur Hand) die wahrscheinlichen Displacements:

```
FORT   EQU  $0A     68k-Familie
BACKT  EQU  $0E     68k-Familie
WLOLD  EQU  $16     68k-Familie

FORT   EQU  $0C     PowerPC-Familie
BACKT  EQU  $10     PowerPC-Familie
WLOLD  EQU  $18     PowerPC-Familie
```

Workspace fixed address request **WSFA = $A008**

Eingaberegister:	`D1.L`	Gewünschte Adresse, teilbar durch 4!
	`A1.L`	Endadresse+4, teilbar durch 4!
Ausgabe-Register:	`A1.L`	Bei Erfolg `A1 = D1`
	`CCR`	„`NE`“ kein Erfolg, „`EQ`“ `A1` geladen

Veränderte Register: `D1,D5,D6,D7`

Der Trap prüft, ob zwischen der Untergrenze in `D1.L` und der Obergrenze+4 in `A1.L` freier Bereich liegt. Ist dies nicht der Fall, so antwortet der Trap mit „`NE`“, anderenfalls mit „`EQ`“. Eventuelle Reste oberhalb und unterhalb werden — falls groß genug — zu freien Sektionen. Der Ausgangswert von `A1` ist identisch mit dem Eingangswert von `D1`, falls `CCR = „EQ“`.

Man beachte, daß die so erzeugte Speichersektion als „`PWS`“, d. h. „Procedure–WorkSpace“ verbucht wird und mit der Terminierung der einstmals erzeugenden Task automatisch wieder zu freiem Speicher wird. Dafür sorgt das sog. „T–link“, eine Kette, die ihren Ursprung im „Task–WorkSpace“ hat und alle von der Task angeforderten CEs (Communication–Element) und „`PWS`“ miteinander zu einem Ring verbindet. Will man die Sektion von der Task ablösen, wie es zum Beispiel der Editor mit neuen Blöcken macht, so muß die Sektion mit einer besonderen Prozedur aus dem „T–link“ herausgenommen werden, die schon auf Seite 543 erläutert wurde.

Beispiel:

```
WSFA  OPD    $A008
     _MOVE.L =$8000,D1   Sektionadr=$8000
      LEA   $9000.L,A1   Bereich:$8000-$8FFF
      WSFA
      BNE   MIST         B:Bereich nicht frei
*---- Nur wenn 'Dauerblock' gewuenscht wird: auslinken!
      OFF                unteilbare Sequenz
      MOVEA.L BACKT(A1),A2     A2 zuf. Beisp.
      MOVE.L  FORT(A1),FORT(A2)  linker Nachb.
      MOVEA.L FORT(A1),A2
      MOVE.L  BACKT(A1),BACKT(A2) rechter Nachb.
      MOVE   =$0010,TYPE(A1) als 'MODULE' ausg.
     _MOVE ...           Modulname 6 Bytes
      DPC                wieder user-mode
```

WSFA (Forts.) — Workspace fixed address request

Die symbolischen Displacements wurden bereits auf Seite 543 beschrieben. Im Übrigen gelten alle anderen Anmerkungen zum Trap `WSBS` entsprechend. Auch die mit `WSFA` erzeugte und danach ausgelinkte Speichersektion kann nur mit `RWSP` oder dem Bedienbefehl „`UNLOAD` *name*“ wieder eliminiert werden.

Der Trap `WSFA` ist ein Bruder der Traps `WSBS`, `WSFS` und `PENTR`, die ähnliche Grundeigenschaften aufweisen.

Natürlich stört dieser Trap die automatische Speicherverwaltung und ist darum nur für Spezial- und Testzwecke sinnvoll.

Workspace forward search **WSFS = $A004**

Eingaberegister:	`D1.L`	Größe des angeforderten Workspace
Ausgaberegister:	`A1.L`	Adresse der Speichersektion
	`CCR`	„`NE`“ kein Erfolg, „`EQ`“ `A1` geladen
Veränderte Register:	`D1,D7`	

Der von **RTOS–UH** verwaltete Speicherbereich wird von unten nach oben nach dem ersten freien Stück abgesucht, in welches die Anforderung hineinpaßt. Dann wird `A1` als Zeiger auf diesen Abschnitt geladen. Der ggf. vorhandene (obere) Rest wird zur freien Sektion verwandelt. Das Statusregister wird zur Rückantwort benutzt. Wenn nämlich kein Platz gefunden wurde, so antwortet der Trap mit „`NE`“, anderenfalls (wenn `A1` geladen werden konnte) mit „`EQ`“.

Man beachte, daß die so erzeugte Speichersektion als „`PWS`“, d. h. „Procedure–WorkSpace“ verbucht wird und mit der Terminierung der einstmals erzeugenden Task automatisch wieder zu freiem Speicher wird. Dafür sorgt das sog. „T–link“, eine Kette, die ihren Ursprung im „Task–WorkSpace“ hat und alle von der Task angeforderten CEs (Communication–Element) und „`PWS`“ miteinander zu einem Ring verbindet. Will man die Sektion von der Task ablösen, wie es zum Beispiel der Editor mit neuen Blöcken macht, so muß die Sektion mit einer besonderen Prozedur aus dem „T–link“ herausgenommen werden, die schon auf Seite 543 beschrieben wurde.

Beispiel:

```
WSFS OPD     $A004
    _MOVE.L  =$1000,D1   Sektion 4 kB brutto
     WSFS
     BNE     MIST        B: kein Platz mehr
*--- Nur wenn 'Dauerblock' gew"unscht wird: auslinken!
     OFF                 unteilbare Sequenz
     MOVEA.L BACKT(A1),A2     A2 zuf. Beisp.
     MOVE.L  FORT(A1),FORT(A2)  linker Nachb.
     MOVEA.L FORT(A1),A2
     MOVE.L  BACKT(A1),BACKT(A2) rechter Nachb.
     MOVE   =$0010,TYPE(A1) als 'MODULE' ausg.
    _MOVE....            Modulname 6 Bytes
     DPC                 wieder user-mode
```

WSFS (Forts.) — Workspace forward search

Die symbolischen Displacements wurden bereits auf Seite 543 beschrieben. Im Übrigen gelten alle anderen Anmerkungen zum Trap `WSBS` entsprechend. Auch die mit `WSFS` erzeugte und danach ausgelinkte Speichersektion kann nur mit `RWSP` oder dem Bedienbefehl „`UNLOAD` *name*“ wieder eliminiert werden.

Der Trap `WSFS` ist ein Bruder der Traps `WSFA`, `WSBS` und `PENTR`, die ähnliche Grundeigenschaften aufweisen. Genau wie jene erlaubt er während der Suche jederzeit eine Taskumschaltung (Preemption).

Eine verantwortungsvolle, sorgfältige Anwendung versteht sich von selbst!

Transfer CE for Input/Output **XIO = $4E4A**

Eingaberegister: `A1.L` muß auf ein CE zeigen
Ausgaberegister: `-`
Veränderte Register: `D1,D5,D6,D7`

Das durch `A1` festgelegte Communication–Element wird mit Hilfe der dort eingetragenen Priorität an den ihm zustehenden Platz in die Warteschlange eines I/O–Dämons (I/O–Task) eingekettet. Welcher I/O–Dämon zuständig ist, ermittelt der Trap an Hand des Parameters `LDNIO` im Communication–Element.

Wenn eine ungültige „`LDN`“ (logical dation number) das Auffinden eines I/O–Dämonen unmöglich macht, so löst der Trap ein Fehlersignal (Exception) mit der Kennung „`... wrong device-ldn (xio-call)`“ aus, setzt den Parameter `RECLEN` auf Null und führt die Operation „`RELCE`“ so aus wie es ein I/O–Dämon normalerweise tun würde. Das Fehlersignal kann mit Hilfe der CE–Parametrierung nur dann unterdrückt werden, wenn das Bit `STABRE` (im Byte `STATIO` des CE) nicht gesetzt ist.

Ist die über die `LDN` adressierte Zieltask inaktiv oder blockiert („`waiting for activation`“), so wird sie aktiviert bzw. diese eine Blockierbedingung wird aufgehoben. Wenn im Mode–Byte des CEs das Wartebit gesetzt ist, wird die aufrufende Task durch den Trap im Zustand `I/O?` blockiert. Diese Blockierung hebt der Empfänger der Nachricht nach deren Auswertung mit Hilfe des `RELCE`–Traps erst später wieder auf.

Der Trap funktioniert völlig analog zum `MSGSND`–Trap, kann allerdings das CE nur an I/O–Dämonen verschicken. Genau wie beim `MSGSND` wird auch hier eine prioritätsgerechte (an Hand der Zelle `PRIO` im CE) Einkettung vorgenommen: Dringende Nachrichten kommen ganz nach vorne in die Schlange. Die Warteschlange kann natürlich niemals überlaufen.

Das weitere Schicksal des CE nach dessen Abarbeitung durch den I/O–Dämon wird durch das Byte `STATIO` im CE bestimmt. Wenn das Bit `STABRE`(Bitno. 1) gesetzt ist, wird das CE mit dem `RELCE` des Empfängers in freien Speicher verwandelt. Ist dagegen das Bit `STABRT`(Bitno. 2) gesetzt, so kehrt das CE nach Abarbeitung in die eigene CE–Schlange des Aufrufers zurück und kann von dort bei Bedarf mit dem `TOQ`–Trap geholt werden.

XIO (Forts.) **Transfer CE for Input/Output**

Hinweis!

Im Normalfall haben die I/O–Dämonen eine Defaultpriorität von Null. Der `XIO`–Trap (wie übrigens auch der `MSGSND`–Trap) interpretiert eine Null dort nämlich als „dynamische Priorität“: Die Priorität des I/O–Dämonen wird optimal an die Gegebenheiten angepaßt, so daß durch den E/A–Vorgang möglichst keine Tasks behindert werden, deren Priorität oberhalb der im CE eingetragenen liegt!

8.2 Das Filesystem

8.2.1 Der Verwaltungskopf

Es wird mit logischen Blöcken gearbeitet, deren jeweilige physikalische Position auf dem Speichermedium durch die „Untergliederungsdaten“ errechenbar ist. Alle Blöcke sind von gleicher Größe. So werden etwa beim „B“-Format für DD-Disketten 5 Sektoren zu einer Blockgröße von 5 kB zusammengefaßt. Der erste Block trägt die Nummer 0. Die Blöcke werden je nach Inhalt in zwei Typen unterschieden: den Verwaltungsblock und den Datenblock. Mit dem Block 0 beginnt der Hauptverwaltungsteil des Mediums. Ab einer bestimmten Blocknummer (bei Disketten ist das in der Regel Block 1) beginnt der Bereich der Datenblöcke, in die für Unterdirectories wieder einzelne kürzere Verwaltungsblöcke eingestreut sein können.

Block 0 beginnt bei Disketten auf Track 00, Sektor 1 und Seite 0. Platten können mehrere, auch systemfremde, Partitionen haben. Dort liegt der Beginn des Blockes 0 jeweils genau an der physikalischen Stelle der Platte, an der die entsprechende Partition beginnt.

Der Hauptverwaltungsblock beginnt stets im Block 0. Seine Struktur ist in der Tabelle 8.1 genau wiedergegeben. Bei Platten ist der Inhalt über mehrere fortlaufende Blöcke verteilt.

$00	HDDRV	11 00	4 Bytes Kennungskopf des UH-Fman zur Identifikation.
$02		00 00	Gehört noch zum Kopf
$04	SODSID	00 0x	Single Or Double-SIDed x=0 single x=1 double etc.
$06	FBLEN	xx xx	Anzahl phys. Bytes pro Block, immer n·256
$08	NOBLPT	xx xx	2-er Logarithmus aus Blockzahl/Track/Seite. Beispiel: 0 -> 1 Bl/Track, 1 -> 2 Bl/Track etc.
$0A	NOSEPB	xx xx	Anzahl Sektoren pro Block, Sektorinkrement.
$0C	HDBLNO	xx xx	HeaDer BLock Nummer −1
$0E	ABMLEN	xx xx	Anzahl der insgesamt vorhandenen Blöcke.
$10	ABMIDX	aa aa	„ABM“-Start-Index.
$12	DIRSTR	dd dd	Directory-Start-Index zum Auffinden des Dir.
$14	DIRLEN	xx xx	Directory-Länge = max. mögl. Anzahl Files
$16	DIRLEE	xx xx	Länge des Einzeleintrages/File im Directory. Mindestens Namenslänge+8 (normal: 16)
$18	NAMLEN	xx xx	Namenslänge in Bytes im Directory, typ. 8
$1A	DIRNUM	nn nn	Number of this directory
$1C	RES1	00 00	Reserviert
$1E	RES2	00 00	Reserviert
$20	RES3	00 00	Reserviert
$22	LABEL	.....	„Label“-Text der Diskette (16 Bytes)
$32	EXTMRK	.....	4 bytes Extension-valid mark
$36	EXTTB	.....	Table for 15 Extensions

...

$aaaa Assigned-block-map „ABM“. Zu jedem Block gehören 4 Bytes. Die ersten beiden bezeichnen den Besitzer-File, die letzten beiden erzeugen eine Kette. Ein unbenutzter (freier Block) ist durch 4 Nullbytes zu erkennen. Die „ABM“ wird so initialisiert, daß die Blöcke, die dieser Verwaltungsteil belegt, automatisch zu Anfang für immer belegt sind. Gleiches gilt für Blöcke, die beim Formatieren als fehlerhaft erkannt wurden. Insgesamt enthält die „ABM“ (4·Anzahl Blöcke) Bytes.

$dddd Directory. Wird wie folgt initialisiert: Erster Eintrag Name **FREE** (mit Blanks auf Namenslänge gefüllt) und alle weiteren (total: DIRLEN·DIRLEE) Bytes jeweils Null. Das Directory kann auch vor der „ABM“ liegen.

Tabelle 8.1: Filesystem, Verwaltungskopf

8.2.2 Die Datenblöcke

Die Datenblöcke benötigen zu Anfang nur die Hardware-Initialisierung. Die innere Struktur ermöglicht ein Lesen der Information auch bei Verlust des Verwaltungskopfes – sofern die Zuordnung der Blocknummern zu den physikalischen Sektoren noch bekannt ist. Jeder Datenblock sieht wie folgt aus:

`$00`	FORBLK	`xx xx`	Folgeblock (0 falls letzter Block des Files)
`$02`	BACBLK	`xx xx`	Vorgängerblock (0 falls erster Block)
`$04`	MINIDX	`aa aa`	Index des ersten Nutzdatenbytes.
`$06`	ACTIDX	`bb bb`	Aktueller Lese-/Schreibindex
`$08`	WRTIDX	`cc cc`	Writer-Index. Schleppzeiger von ACTIDX.
`$0A`	WMXIDX	`dd dd`	Indexgrenze, letztmögl. Byte des Blocks+1
`$aaaa`			Erstes Nutzdatum
`$bbbb`			(Zufälliger) aktueller Zeiger.
`$cccc`			(Zufällige) Schreiber-Schlepposition.
`$dddd-1`			Letztes mögliches Nutzdatum

Tabelle 8.2: Filesystem, Datenblock

8.2.3 Eigene Driver für das RTOS–UH-Filesystem

Um eigene Driver schreiben zu können, die weitgehend unabhängig von den internen Revisionen der Filehandler sind, ist es unbedingt notwendig, den Driver mit dem gemeinsamen Head der entsprechenden Version des Filehandlers zu assemblieren, der in Ihrem System verwendet wird. Dazu muß die Versionsnummer des Heads mit der der Filemanger übereinstimmen. Im Augenblick ist die Version 3 Revision 8 der Filehandler gültig, es ist also ein Head der Version 3.x zu verwenden.

Ein Filehandler Driver ist eine Task, die in ihrem Taskworkspace eine Reihe von Parametern aufsetzt, die Controller Hardware initialisiert und dann in den eigentlichen Filehandler springt. Dieser ruft dann eine Reihe von Unterprogrammen auf, mit denen z. B. ein Block von der Diskette/Platte gelesen wird, ein Block geschrieben wird oder auch das Medium formatiert wird. Alle diese Unterprogramme kehren immer wieder über einen `RTS` zum Filehandler zurück. Fehler werden immer dem Filehandler gemeldet, ein Driver sollte nie eine eigene Ausgabe von Fehlermeldungen veranlassen.

Die Unterprogramme, die der Driver dem Filehandler zur Verfügung stellen muß, sind:

-RDISC — um einen Block vom Medium zu lesen. Die Parameter dazu werden in zwei Datenblöcken übergeben, auf die `A0` und `A6` zeigen.

-WDISC — um einen Block auf das Medium zu schreiben. Die Parameter werden wie bein „RDISC“-Unterprogramm übergeben.

Eine Anzahl von weiteren Unterprogrammen sind nicht unbedingt notwendig. Wenn sie nicht implementiert werden, können sie durch ein einfaches `RTS` ersetzt werden.

–FORM — um eine Anzahl von Tracks auf einem Medium physikalisch zu formatieren. Die Parameter dazu werden in festen Zellen im Taskworkspace übergeben. Wenn dieses Unterprogramm nicht vorhanden ist, wird bei einem FORM Befehl an den Filehandler nur das Medium auf defekte Blöcke untersucht und das Root Directory neu eingerichtet. Die physikalische Formatierung kann dann z. B. mit einem anderen Programm erfolgen. Dies ist z. B. bei Platten an SCSI-Controllern notwendig, da der Filehandler nicht genügend Informationen zum Einrichten der Description Pages im Controller übergeben kann.

–DESEL — Der Filehandler ruft dieses Unterprogramm auf, wenn auf keinem der von dieser I/O-Task betreuten Laufwerken noch ein File offen ist. Typischerweise kann man damit die Select Lampen aller Laufwerke ausschalten.

–DISPOP — wird während des Formatierens automatisch für jeden Block aufgerufen, nachdem zuvor „RDISC“ aufgerufen wurde. Die Parameter sind die Blöcke, auf die `A0` und `A6` zeigen.

Die Adressen dieser Routinen hat die Driver Task in feste Addressen in ihrem Taskworkspace einzutragen, bevor die Kontrolle an den Filehandler weitergegeben wird. Unter dem Label RDISCA hat die Addresse der Routine RDISC, unter WDISCA die Addresse von WDISC, unter FORMA die Adresse von FORM, unter DISPOA die Adresse von DISPOP und unter DESELA die Addresse von DSEL zu stehen. Neben den Adressen dieser Unterprogramme sind aber noch Zeiger auf einige Tabellen zu übergeben:

FOCTS ist die Tabelle der Beschreibung der einzelnen Formate für Single Density. Für jeden Formattyp ist ein Eintrag notwendig, der mindestens aus folgenden vier Worten bestehen muß:

1. Anzahl der Sektoren pro Block
2. Bytes je Sektor
3. Einem Infowort
4. Der Anzahl Blöcke pro Track −1

Der Filehandler überträgt diese vier Worte vor dem Aufruf des Formatters (FORM) in die Zellen NSEPT, SEL, FSELB und NBLPT im Taskworkspace. Das erste Byte von FSELB wird zusätzlich mit dem zur Laufwerknummer gehöhrenden Eintrag aus der FOCTU Tabelle verknüpft. Ein Zeiger auf diese Tabelle ist in FOCTSA im Taskworkspace einzutragen.

FOCTD ist die Beschreibungstabelle für der einzelnen Formate in Double Density. Sie is wie FOCTS aufgebaut. Der Zeiger auf diese Tabelle hat in FOCTDA im Taskworkspace zu stehen.

Weiter ist die Länge (Anzahl Bytes) eines Eintrages in den Tabellen FOCTS und FOCTD im Taskworkspace als Langwort bei FOCTEL einzutragen.

FOCXX ist die Tabelle der im FORMAT Befehl angebbaren Formattypen. Jeder Eintrag ist genau zwei Byte lang. Das Ende dieser Tabelle wird mit zwei Bytes Null gekennzeichnet. Die Formattypen sind in der Reihenfolge anzugeben, in der sie auch in den FOCTS und FOCTD Tabellen stehen, und für jeden Eintrag in der FOCXX Tabelle hat in den FOCTS und FOCTD Tabellen jeweils ein Eintrag vorhanden zu sein.

FOCTU enhält für jedes mögliche Laufwerk einen ein Byte langen Eintrag. Vor dem Aufruf des Formatters für das Laufwerk n wird das *n*-te Byte dieser Tabelle mit dem ersten Byte in FSELB oder Verknüpft. Auf die Tabelle FOCTU hat FOCTUA im Taskworkspace zu zeigen.

DRVMSA Maske im Taskworkspace des Drivers. Hier ist eine Maske einzutragen, die zur Ermittlung der gültigen Laufwerknummer dient. Die im Filehandler verwendete Laufwerksnummer entsteht durch eine „UND“-Verknüpfung dieser Maske mit der aus dem CE stammenden Drive Nummer.

DRVNAM ist die Tabelle der Laufwerknamen. In ihr ist für jedes nach der Ausblendung über DRVMSA mögliche Laufwerk ein Eintrag vorzusehen. Jeder Eintrag in diese Tabelle ist genau 4 Bytes lang. In den ersten beiden Bytes steht die Laufwerknummer ASCII codiert. Das dritte Byte enthält die LDN, für die dieser Driver arbeiten soll, plus `$80`, und im vierten Byte steht die Laufwerknummer. Die Tabelle ist mit einem Wort `$0000` abzuschließen. Wenn das System aus dieser Tabelle auch die Memo's für die Laufwerkbezeichnungen ermitteln soll, so muß vor der DRVNAM Tabelle die Scan-Marke (Scheibe 9) `$AEB1`, `$BF95`, `$02BF` stehen. Denken Sie auch daran, in dem Driver die Device Facilities der verwendeten „LDN“ auf `$C7F8` zu setzen. Ein Zeiger auf die DRVNAM Tabelle ist unter DRVNNA im Taskworkspace abzulegen.

Das Langwort INITYA und das Wort FTYMSA im Taskworkspace werden nur für alte Driver benötigt. Sie müssen aber mit `$00000004` bzw. `$0004` vorbesetzt werden.

In das Wort FLDN im Taskworkspace ist die LDN, unter der dieser Driver arbeitet, einzutragen.

Das Langwort FATTSA im Taskworkspace des Drivers ist nur für dem MSFM von Bedeutung. Ein gesetztes Bit bedeutet, daß der MSFM für das dem Bit entsprechende Laufwerk eine FAT mit 16 Bit langen Einträgen annehmen wird.

Die Routinen RDISC und WDISC erhalten ihre Parameter aus Datenblöcken, auf die die Register `A0` und `A6` zeigen. Die mehr allgemeinen Parameter für ein Laufwerk sind in dem über `A0` zu erreichenden Driveblock abgelegt. Dies sind:

`BLKLEN`	(Wort) Länge der zu lesenden oder schreibenden Daten.
`NOSEPB`	(Wort) Anzahl der Sektorn pro Block.
`SODSID`	(Wort) Anzahl der Oberflächen −1
`NOBLPT`	(Wort) Anzahl der Blöcke pro Track −1
`ERRPTC`	(Wort) Anzahl der Versuche

Das Byte `FPTFL(A0)` muß auf jeden Fall gelöscht werden. Wenn möglich, sollte hier auch bei jedem Leseauftrag eingetragen werden, ob der Schreibschutz für das Medium aktiviert ist: es ist dann auf `$FF` zu setzen.

Die über `A6` zu erreichenden Parameter betreffen den einzelnen Lese- oder Schreibauftrag:

`BLKNO`	(Wort)	Der zu bearbeitende Block.
`DADR`	(Langwort)	Adresse der Daten.
`ERRNO`	(Wort)	Fehlernummer.

Die Routinen RDISC und WDISC melden mit dem Zero Bit („`EQ`“) im Statusregister des Prozessors, ob der Auftrag erfolgreich ausgeführt werden konnte. „`NE`“ bedeutet, daß in ERRNO eine Fehlermeldung abgelegt worden ist. Es sind folgende Fehlernummern zugelassen:

4: Data Address Mark Error
8: Track 000 not found (Position Error)
12: Aborted Command Error
16: Controller fault
20: ID-Field not found
24: CRC-Error in ID or Data
28: Uncorrectable Data
32: Bad Block found
36: Drive not ready
40: Device Write protected
44: Disk Changed
48: Drive not present

Die Anzahl der Versuche in ERRPTC ist normalerweise auf 12 vorbesetzt. Nur beim Lesen, das zum Auffinden von defekten Blöcken dienen soll, wird hier 2 eingesetzt. Die Speicherzelle darf aber nicht verändert werden.

Driver, die Disks mit unterschiedlichen Formaten handhaben, müssen in der Lage sein, beim Parametersatz „Block 0, Blocklänge 1024 Bytes, 4 Sektoren/Block, 1 Block pro Track und 1 Oberfläche“ (BLKNO=0, BLKLEN=1024, NOSEPB=0, NOBLPT=0, SODSID=0) die ersten 1024 Byte vom Medium einzulesen. Nur bei dieser Kondition braucht damit gerechnet zu werden, daß eine Diskette mit einem anderen Format zu bearbeiten ist. Nach einem Format Befehl hat aber das Lesen mit der Sektorlänge, die beim Formatieren verwendet wurde, sofort möglich zu sein. Der Driver muß also, wenn es die Hardware erfordert, eine Information in eigenen Zellen speichern, an der er erkennt, welches Format zuletzt auf welchem Laufwerk verwendet wurde.

Das FORM Kommando erhält seine Daten über Speicherplätze im Taskworkspace. Diese sind:

FSELB	(Wort) Siehe FOCTS
SEL	(Wort) Sektor Länge in Bytes
MAXTRK	(Wort) Anzahl der Tracks
SINGFL	(Wort) 0: Single Density
NSEPT	(Wort) Sektoren pro Track
FERRC	(Wort) Fehlercode beim formatieren
FORMT	(Wort) Position des Formats in FOCXX

Im Fehlerfall wird die Fehlernummer bei FERRC eintragen. Die Nummern sind wie beim Schreiben/Lesen vergeben.

8.3 Das Communication Element

8.3.1 Benutzung und Aufbau des CE

Das Betriebssystem übernimmt die Betreuung von Warteschlangen an E/A-Bausteinen mit begrenzter Übertragungsgeschwindigkeit, wie z. B. UART (ACIA) oder solchen digitalen Ein-/Ausgaben, bei denen zwischen den Transfers aus anderen Gründen Wartephasen anfallen (z. B. Floppy-Koppler o. ä.). Für rein elektronische und damit aus der Sicht des digitalen Prozesses beliebig schnell mögliche E/A ist das Betriebssystem nicht zuständig, da keine Wartephasen anfallen, die einen Taskwechsel sinnvoll werden lassen.

Eine Task, die eine Ein-/Ausgabe beginnt, muß sich zuvor in den Besitz eines „Communication-Elements“ (CE) bringen - wenn sie nicht noch ein „leeres“ von einer vorhergehenden E/A besitzt. Dafür ist die Operation mit dem Trap „`FETCE`“ vorgesehen.

Das CE wird von der Task parametriert, z. B. durch Einsetzen der Warteschlangen-Nummer („LDN“), des Übertragungsmodes, der Satzlänge, des File-Namens usw.

Das gefüllte CE wird von der Task mit Hilfe der Trap-Operation „`XIO`“ oder „`MSGSND`“ dem Betriebssystem angeboten. Diese Traps bringen die Task in den Wartezustand „`I/O?`“, sofern bei der Parametrierung des CE das Wartebit im Übertragungsmode gesetzt wurde. Wenn das Freigabebit („Release“-Bit) von der Task gesetzt wurde, ist der Aufruf von `XIO` (bzw. `MSGSND`) die letzte zulässige Operation der Task mit diesem CE, da es nach Abarbeitung in der Warteschlange automatisch als freier Speicher in die Verwaltung von **RTOS–UH** zurückkehrt.

Wenn das Freigabebit nicht gesetzt wurde, muß die Task in geeigneter Weise mit dem CE weiter verfahren. Eine Neuparametrierung ist erst erlaubt, nachdem das CE abgearbeitet wurde. War beim Aufruf von `XIO` das Wartebit gesetzt, so kann die Task nach dem `XIO`/`MSGSND` sofort neu über das CE verfügen. Andernfalls ist mit der Systemfunktion „`IOWA`“ (I/O-Wait) das Ende der I/O-Operation abzuwarten. Ggf. kann noch vor einer Neuparametrierung nach dem `IOWA` der Rückmeldecode im CE analysiert werden.

Ein CE wird auf drei mögliche Arten wieder zu freiem Speicherraum:

1. Aufruf von „`XIO`“ mit gesetztem Freigabebit.
2. Die besitzende Task wird terminiert oder beendet sich selbst.
3. Die Task ruft die Operation „`RELCE`“ für dieses CE auf.

Der Aufruf von „`RELCE`“ (Release CE) ist immer möglich, solange die Task Besitzer des CE ist, also auch während eines noch laufende E/A-Vorganges über dieses CE. `RELCE` beeinflußt nicht die laufende oder durch `XIO` gerade veranlaßte E/A-Operation.

Bei der Terminierung einer Task ist zwischen Ein- und Ausgabe zu unterscheiden:

- Bei der Ausgabe verbleiben mit `XIO` bereits in die Ausgabewarteschlange gebrachte CEs dort, es wird nur das Freigabebit gesetzt und die Task-ID im CE gelöscht.
- Bei der Eingabe werden die CEs, die noch nicht in Bearbeitung durch die Betreuungstask der Warteschlange sind, aus der Schlange genommen und zu freiem Speicher umgewandelt. In Bearbeitung befindliche CEs verbleiben bei der Betreuungstask (die angefangene Eingabe muß also regulär zu Ende geführt werden) bis sie abgearbeitet sind und werden dann zu freiem Speicher umgewandelt.

Das CE ist ein wichtiger Bestandteil der Ein-/Ausgabe im Betriebssystem **RTOS–UH**. Dabei ist der genaue Aufbau nur für den Assemblerprogrammierer interessant. Für den Hochsprachanwender übernehmen Compiler und Laufzeitsystem die Parametrierung der CE's. In der folgenden Tabelle sind in der linken Spalte die Displacements der einzelnen Parameter relativ zu `A1`, dem Zeiger auf das CE nach „`FETCE`“, und daneben kurz ihre Bedeutung angegeben.

Name	68xxx	PowPC	len	Funktion
– –	$00	$00	–	Verwaltungszeiger, nicht verändern!
TIDO	$12	$14	4	Task-ID of owner, nicht verändern!
FORS	$16	$18	4	Verwaltungszeiger, nicht verändern!
BACKS	$1A	$1C	4	Verwaltungszeiger, nicht verändern!
PRIO	$1E	$20	2	Plazierungspriorität in der Queue, darf verändert werden. Defaultwert ist die aktuelle Laufprio. der Task.
BUADR	$20	$24	4	Buffer-Adresse. Darf verändert werden. Defaultbesetzung: zeigt auf den Platz FNAME + max. erlaubte Pfadlänge.
RECLEN	$24	$28	2	Blocklänge (Anzahl Bytes), I/O-Task gibt eine Null oder negative Zahl zurück, falls I/O nicht ausführbar.
STATIO	$26	$2A	1	Statusbyte
LDNIO	$27	$2B	1	LDN der Warteschlange, für die das CE bestimmt ist oder in der es steht.
MODE	$28	$2C	2	Betriebsart, Endebedingung etc.
DRIVE	$2A	$2E	2	Drive-Nr. linkes Byte: Time-Out wenn $80 addiert.
FNAME	$2C	$30	?	File-Name oder „Pathlist“, Autostop durch $FF, es sei denn, die maximale Länge ist bereits erreicht. Beachten Sie bitte die maximale Länge (implem. abhängig)

Tabelle 8.3: Aufbau des Communication Elementes

8.3.2 Die Modebytes

Die beiden Bytes des Mode-Wortes haben sehr unterschiedliche Bedeutung.

- Das linke Byte (Adresse+0) ist bitweise mit überlagerbaren Funktionen belegt. Das Setzen des entsprechenden Bits aktiviert die beschriebene Funktion. Die niederwertigen 3 Bit haben bei den seriellen Schnittstellen eine feste, beim Filesystem dagegen eine vom Betriebsbefehl abhängige (sehr oft auch keine) Bedeutung. Genauere Informationen dazu in der Tabelle 8.5.

- Das rechte Byte (Adresse+1) benutzt für die höchstwertigen 3 Bit ebenfalls eine bitweise Kodierung. Die verbleibenden rechten 5 Bit kodieren einen von 32 möglichen Betriebsbefehlen für den ausführenden I/O-Dämonen. Die Dämonen werten nur die für sie interessante Teilmenge der Betriebsbefehle aus.

Am besten versteht man die Funktion des gesamten I/O-Systemes, wenn man sich neben diesem Abschnitt die Beschreibung der Traps `FETCE`, `IOWA`, `MSGSND`, `RELCE` und `XIO` durchliest. Das CE ist ein Baustein eines leistungsfähigen und komplexen Kommunikationssystemes. Es ist keinesfalls nur für die echte „Ein“- oder „Ausgabe“ gedacht. Auch zum Rangieren von Botschaften und Befehlen zwischen verschiedenen Prozessen (Tasks) – auch über eine Vernetzung – bedient **RTOS–UH** sich dieses Werkzeuges.

Byte	Mnemo	Bedeutung
$80	MODMWA	Suspend (Wait) until ready
$40	MODMOU	Directionbit, set if Output-direction
$20	MODMCR	Auto-stop after transmission of a Carriage-return
$10	MODMLF	Auto-stop after transmission of a Line-feed
$08	MODMEO	Auto-stop after transmission of an EOT
$04	MODMSC	Beim Filesystem: Siehe Tabelle 8.5 Bei serieller Schnittstelle: Suppress Command, Cotr. `A`, `B`, `C` ohne Wirkung.
$02	MODMNE	Beim Filesystem: Siehe Tabelle 8.5 Bei serieller Schnittstelle: No echo on input
$01	MODMBI	Beim Filesystem: Siehe Tabelle 8.5 Bei serieller Schnittstelle: Binärer Transfer

Tabelle 8.4: Die Bits im linken Modebyte

Byte	Mnemo	Bedeutung
$07	FINDA	Bei DIR ($0E): FIND -A
$06	FIND	Bei DIR ($0E): FIND-Befehl
$04	NOCLO	Bei READ: kein automatisches Close am Ende des Files.
$03	DIREA	Bei DIR ($0E): DIR -EA
$02	BADBL	Bei READ raw: Badblock setting
-“-	DIRE	Bei DIR ($0E): DIR -E
$01	DIRA	Bei DIR ($0E): DIR -A Bei SAVE ($14): Rückgabe Filesize
-“-	RETA	Bei RETURN ($04): -A Parameter

Tabelle 8.5: Die unteren 3 Bits im linken Modebyte

Byte	Mnemo	Bedeutung
$80	IOCMEF	Rückmeldebit bei Eingabe: End-of-file
$40	IOCMNE	No Error messages by damon
$20	IOCMEX	Exclusiv access this task.
$1x	...	Betriebsbefehlkodierung
...	...	in diesen Bits gemäß
$0x	...	Tabelle 8.7 unten.

Tabelle 8.6: Die funktionellen Bits im rechten Modebyte

Byte	Bedeutung
$00	READ/WRITE „old“ File
$01	ERASE the File
$02	REPORT Error
$03	-
$04	RETURN the File (-a -> Modebyte)
$05	-
$06	CLOSE the File
$07	READ/WRITE „ANY“ File
$08	REWIND existing (OLD) File
$09	APPEND the File
$0A	-
$0B	-
$0C	(list) FILES (CE in the CE)
$0D	(list) FREE (CE in the CE)
$0E	(list) DIRectory (CE in CE) (-e -a -> Modebyte)
$0F	-
$10	SYNC (i.e. save on medium)
$11	TOUCH the File, (read or write by directionbit in Modebyte)
$12	LINK (additional name) to File
$13	SEEK (change position in file)
$14	SAVE next pos. to write to a file (SEEK's counterpart)
$15	REWIND any (install if necessary) File
$16	REWIND new (error if exists) File
$17	FORMAT single density, parameters by FNAME*
$18	FORMAT double density, parameters by FNAME*
$19	CF, change filesystemstate, parameters by FNAME*
$1A	MaKe DIRectory
$1B	ReMove DIRectory
$1C	RENAME (change name)
$1D	-
$1E	-
$1F	READ/WRITE rawblock, BADBLOCK by Modebyte

Tabelle 8.7: Die Betriebsbefehle im rechten Modebyte

* Zur Kodierung von FNAME siehe gleichnamigen Shellbefehl.

Byte	Mnemo	Bedeutung
$80	STABFL	„FLag“ zur freien Verwendung des Nutzers, wird vom System nicht verändert oder beachtet.
$40 $20 $10	... ...	Reserviert. Reserviert. Reserviert.
$08	STABOQ	„in Own Queue“. Der Trap `RELCE` setzt dieses Bit im Moment der Rückgabe (STABRT war gesetzt) des CE in die eigene Queue des Besitzers. Obwohl eingekettet, ist es in Wirklichkeit ungebunden.
$04	STABRT	„ReTurn“. Der Trap `RELCE` soll, wenn er vom I/O-Dämonen exekutiert wird, das CE in die eigene Queue des Besitzers zurückgeben.
$02	STABRE	„RElease“. Der Trap `RELCE` soll, wenn er vom I/O-Dämonen exekutiert wird, das CE in freien Speicher verwandeln („Verschrottungsbit“). Kann vom Nutzer vor dem `XIO` oder `MSGSND` gesetzt werden (dann ist das CE mit dem `XIO`/`MSGSND` für ihn gestorben). Ein von der Besitzertask auf das CE ausgeführter `RELCE` oder ein `TERMI` bzw. `TERME`, der auf die Besitzertask wirkt, setzt dieses Bit ebenfalls (für die interne Aufräumarbeit).
$01	...	Reserviert

Tabelle 8.8: Die Bits im Statusbyte des CEs

Man sollte die zur Zeit vom System nicht benutzten Bits in `STATIO` gemäß Tabelle 8.8 wirklich frei lassen. Bei zukünftigen Erweiterungen des Systemes könnte es sonst zu großen Komplikationen kommen.

8.4 Assemblerkodierte PEARL-Unterprogramme

8.4.1 Parameterübergabe bei PEARL90

Die Assembler und Transferassembler des **RTOS–UH**-Systemes sind sehr gut geeignet, um in Maschinensprache für Sonderzwecke Unterprogramme zu kodieren, die von PEARL-Programmen aufgerufen werden können. Mit dem Übergang auf PEARL90 und der Einführung der Formate wurde diese Aufgabe einerseits für den Programmierer erheblich erleichtert, andererseits aber auch die Effizienz dieses Anschlusses deutlich verbessert. An dieser Stelle wird zunächst nur noch der neue T–Code kompatible Anschluß beschrieben. Die alte PEARL80-Notation sowie Umstellhinweise von PEARL80 auf PEARL90 folgen in den nächsten Abschnitten.

Wir betrachten hier exemplarisch den Fall

`ABCD(p1, ... , pn);` gleichbedeutend zu `CALL ABCD(p1, ... , pn);`

wobei `ABCD` ein Assemblerunterprogramm mit n Parametern sein soll; das Symbol `ABCD` muß im aufrufenden PEARL-Programm als `... ENTRY GLOBAL` spezifiziert sein. Auf der Assemblerseite wird der Einstiegpunkt von `ABCD` durch Voranstellung des Zeichens „∼“ (Tilde) global deklariert. Dies ist ein wichtiger Unterschied zum alten PEARL80. Durch nunmehr zwei Sorten globaler PEARL-Symbole werden gefährliche Irrtümer mit Mixturen aus „alt“ und „neu“ ausgeschlossen.

Der PEARL90-Compiler erzeugt zu obiger Aufrufanweisung folgende 4 Kodeabschnitte:

1. Berechnung etwaiger Parameter oder Feldelementadressen
2. Bereitstellung und Vorbereitung von Prozedurarbeitsspeicher
3. Ablage der Parameterverweise im Prozedurarbeitsspeicher
4. Nur im Testmode: Signaturgenerierung
5. Native Sprung an die Stelle ∼`ABCD` oder ∼`ABCD-12`.

Im Testmode wird die um 12 reduzierte Einstiegsadresse als Sprungziel verwendet. Der Sprung erfolgt mit einem Befehl, der im T–Code als `XJRS` kodiert würde.

Im Gegensatz zum alten PEARL80 beginnen alle Unterprogramme heute im native mode des jeweiligen Prozessors. Virtuelle Befehle sind bei Prozeduraufruf und –Rückkehr nicht mehr beteiligt.

Die Schnittstellenbedingungen unmittelbar an der Einsprungstelle – bzw. am Signaturprüfungseinstieg – können wie folgt beschrieben werden:

D0	Signatur, wird nur im Testmode versorgt, bei dem die Routine 12 Bytes vor dem eigentlichen Einstieg angesprungen wird.
A2	enthält den potientellen neuen A5-Wert. Die Prozedur muß an Hand der Größe (mit Hilfe von A3) entscheiden, ob er verwendbar ist.
A3	zeigt auf das erste Byte hinter dem letzten nutzbaren Byte im per A2 angebotenen Workspace
A5	zeigt noch auf den lokalen Workspace des Aufrufers.
-4(A2)	enthält die 4 byte lange Adresse für die Wertrückgabe. Diese Zelle ist auch vorhanden, aber undefiniert besetzt, wenn es sich nicht um eine Funktionsprozedur handelt.
-4-x1(A2)	Wert oder Adresse (IDENT) des Parameters p1. Im Mode „per IDENT“ ist x1 bei skalaren Objekten genau 4, bei Feldern dagegen so lang wie der zugehörige Feldbeschreibungsblock. Im Mode „by value“ ist x1 die nach bestimmten Regeln aufgerundete Länge des Parameters p1. Besteht der Wert eines Parameters aus mehr als 256 Bytes, so wird vom Compiler statt des Parameterwertes die 4-Byte lange Adresse einer zuvor vom Compiler angefertigten Kopie übergeben.
-4-x1-x2(A2)	Wert oder Adresse (IDENT) des Parameters p2. Weiteres analog zum Parameter p1.

Tabelle 8.9: Parameterschnittstelle bei PEARL90

Man sieht, daß Parameterwerte oder Objektadressen in gegenüber der Prozedurdefinition umgekehrter Reihenfolge im A2-Space stehen. Dieser Bereich mit negativen Offsets ist auf 16 kB begrenzt. Bei einer extrem großen Parameteranzahl und/oder vielen großen Objekten knapp unterhalb der 256-Byte Grenze im „per value“-Mode kann es passieren, daß der Compiler einen Kapazitätsüberlauf beim Prozeduraufruf anzeigt. Bei ernsthaften Programmen wurde das allerdings noch nie gesehen.

Weil der Compiler so ausgelegt ist, daß stets mindestens 32 Bytes zwischen die in A2 und A3 angegebenen Adressen passen, läßt sich bei vielen Bibliotheksroutinen ein **schneller Sonderfall** konstruieren. Wenn dieser 32 Byte große Platz für interne Variablen reicht und keine anderen unterlagerten Unterprogramme aufgerufen werden, so kann ein Unterprogramm ohne weitere Prüfung sofort mit der Aktion innerhalb des A2-Space beginnen und direkt mit dem T–Code–Befehl `XRTS` beendet werden.

Im **Regelfall** muß jedoch zunächst sichergestellt werden, ob der Platz reicht. Außerdem müssen Vorbereitungen für unterlagerte Prozeduren getroffen werden. Auch das Verlassen der Routine ist nun mit einigen Maschinenbefehlen verbunden, weil A5 verändert wurde und wieder auf den Aufruferwert zurückgestellt werden muß.

Beide Fälle werden wir im folgenden betrachten.

Schneller Sonderfall

Wir kodieren eine beispielhafte simple Funktion, die weniger als 32 Bytes privaten Speicher benötigt und keine weiteren Unterprogramme aufruft. Es sei

```
XYZ: PROC((I1,I2) FIXED(15) RETURNS(FIXED(15));
     RETURN(I1+I2); END;
```

als Assemblerversion zu kodieren.

```
           .INCLUDE  .../PROCS.FOR          wegen SIGCHK
           .INCLUDE  .../COMEQU.NOL         wegen RGLR
retva      EQU     -4                       Offset returnvalue adr.
I1         EQU     -6                       Offset I1
I2         EQU     -8                       Offset I2
           ...
           LOCK     RGLR          =$0100    Linkregister-lock
           SIGCHK   $F175106F               wird später beschrieben
~XYZ       MOVE   I1(A2),D1                 Wert von I1 nehmen
           ADD    I2(A2),D1                 Wert von I2 addieren
           MOVEA.L retva(A2),A1             Adr Rückgabewert
          _MOVE   D1,(A1)                   Ergebnis ablegen
           XRTS                             schnelle Rückkehr
```

Das Beispiel enthält bereits den Signaturcheck, der später beschrieben wird. Würde man im obigen Beispiel noch ein paar private Speicherzellen benötigen, so könnte man diese Bytes im Bereich

`0(A2) ... $1F(A2)` ablegen.

Bitte beachten:

Mit Ausnahme der Register `A4`, `A5`, `A6` und `A7` dürfen alle Register innerhalb des Unterprogrammes frei verwendet werden. Beim PowerPC steht die Rückkehradresse zunächst nur im Linkregister. Der Transferassembler beklagt sich bei PC-relativen Adressierungen möglicherweise über das Linkregister-`LOCK`. Es kann entfernt werden, wenn am Prozedureingang ein `XSL` eingefügt und gleichzeitig der `XRTS` durch den gewöhnlichen `RTS` ersetzt wird. `A7` ist beim schnellen Sonderfall nur mit 1 weiteren BSR-Level (4 Bytes) belastbar!

Möglichst jedes PEARL-Unterprogramm sollte „`REENTRANT`“ sein! Es dürfen folglich keine statisch (auf festen Speicherzellen) angelegten Objekte im UP verändert werden.

Der Regelfall

Das obige Beispiel ist nicht typisch, denn es enthält beinahe gar keine Aktion innerhalb der Prozedur. Nur aus diesem Grund läuft das assemblerkodierte Unterprogramm tatsächlich deutlich (etwa um den Faktor 2) schneller als eine gleichwertige, vollständig in PEARL90 kodierte Variante. Der PEARL-Compiler zieht nämlich keinen Nutzen aus dem Spezialfall sondern kodiert stets den „Regelfall“ einer Prozedur, bei dem vorab nicht gesichert ist, daß der benötigte Platz für die internen Variablen tatsächlich in den vom Aufrufer angebotenen Raum paßt.

Den Regelfall können wir auch in Assemblersprache mit Hilfe des `PRODEC`-Formates kodieren. Dieses Format läd in jedem Fall A5 neu. Dabei ist der neue A5-Wert nicht identisch mit dem Eingangswert von A2 sondern um den Versatz `FLVA` größer. Die Parameter wurden also scheinbar etwas weiter in den negativen Bereich geschoben. Dies ist erforderlich, weil zum Beispiel auf -4(A5) in der PEARL-Welt mit der Zelle `BWIO` eine wichtige Pufferzelle für die formatierte Ein-Ausgabe steht. Der Versatz `FLVA` ist zwischen 68k und PowerPC unterschiedlich und muß im T–Code darum aus der Datei `COMEQU` geholt werden. Die wahrscheinlichen Werte sind

```
FLVA  EQU $22    bei der 68k-Familie
FLVA  EQU $24    bei der PowerPC-Familie
```

Reicht der mit A2/A3 angebotene Platz für die internen Prozedurobjekte nicht aus, so wird mit dem Maschinenkode des `PRODEC`-Formates automatisch eine neue **RTOS–UH**-Sektion mit Hilfe des `PENTR`-Traps angefordert. Durch die Compileroption „`/*+R=...*/`“ kann man aber einen genügend großen Taskworkspace vorhalten und damit diesen zeitlich sehr teuren Exkurs in die Speicherverwaltung vermeiden. (Im Beispiel kann durchaus ein Verlust um den Faktor 10 auftreten.)

Das `PRODEC`-Format hat zwei Parameter, nämlich die Gesamtgröße des in der Prozedur benötigten Workspace (im Bereich positiver A5-Displacements) und das Gesamtparametervolumen in Bytes. Bei letzterem darf man nur dann 0 einsetzen, wenn die Prozedur weder Parameter erhält noch ein Ergebnis zurückliefert.

Die Rückkehr aus dem Unterprogramm darf nun nicht mehr mit einfachem `XRTS` erfolgen sondern es muß das `PROCEX`–Format verwendet werden. In diesem Format wird sichergestellt, daß ein eventuell aus der **RTOS–UH**-Speicherverwaltung angeforderter Workspace an das System zurückgegeben wird.

Wir nehmen das gleiche Beispiel wie oben an, nun jedoch mit Extra-Workspace von 1kB. Im Bereich der positiven Displacements von A5 sind damit 1024 Bytes frei verfügbar (die im Beispiel allerdings nicht benutzt werden).

```
          .INCLUDE  .../PROCS.FOR        wegen SIGCHK
          .INCLUDE  .../COMEQU.NOL       wegen RGLR, FLVA
retva     EQU     -4-FLVA                Offset returnvalue adr.
I1        EQU     -6-FLVA                Offset I1
I2        EQU     -8-FLVA                Offset I2
WSPSZ     EQU     $400                   angenommener WSP
          ...
          SIGCHK   $F175106F             wird später beschrieben
~XYZ      PRODEC WSPSZ,-I2-FLVA          Workspace anlegen
          MOVE   I1(A5),D1               Wert von I1 nehmen
          ADD    I2(A5),D1               Wert von I2 addieren
          MOVEA.L retva(A5),A1           Adr Rückgabewert
          _MOVE   D1,(A1)                Ergebnis ablegen
          PROCEX                         Verlassen des UP
```

Dieses Beispiel läuft nun sehr genau mit der gleichen Geschwindigkeit wie die PEARL-kodierte Version. Die Assemblerkodierung lohnt sich also nur bei Systemroutinen, die mit dem Miniworkspace auskommen oder sehr spezielle Hardwareoperationen nutzen, die sich in PEARL nicht gut formulieren lassen.

8.4.2 Der Signaturcheck in PEARL90

Der PEARL90-Compiler prüft bekanntlich schon zur Compilezeit die Korrektheit der Aktualparameter eines Prozedur- oder Funktionsaufrufes. Bei im selben Modul deklarierten Prozeduren ist diese Prüfung immer aktiv und lückenlos. Werden jedoch globale Prozeduren aus externen Modulen aufgerufen, so kann der Compiler die Aktualparameter nur mit der vom Programmierer niedergeschriebenen Spezifikation vergleichen, welche natürlich fehlerhaft sein kann. Dieses Problem war in früheren Jahren bei der Umstellung von PEARL80 hinreichend bekannt. Das **RTOS**–**UH**-PEARL90 sieht einen mit dem Testmode „`/*+T */`“ zuschaltbaren besonderen Mechanismus vor, der auch derartige Fehler mit großer Wahrscheinlichkeit erkennen und gefährliche Nebenwirkungen verhindern kann, den **Signaturcheck**.

Wie bei den Beispielen für assemblerkodierte Unterprogramme deutlich zu sehen war, werden übergebene Parameter direkt und ohne Prüfung von ihren mutmaßlichen Ablageplätzen geholt. Eine individuelle Prüfung wie im alten PEARL80 wäre zeitlich viel zu teuer. Der Compiler berechnet darum zu jeder Prozedurdefinition eine 32-Bit lange Zahl, die mit sehr hoher Wahrscheinlichkeit (aber nicht sicher!) bei relevant anders definierten Prozeduren einen anderen Wert annimmt. Benutzt wird dabei eine Polynomformel für einen sog. 31 Bit Galoiskörper. In diese Formel gehen Art, Stellung und Zahl der Parameter auf recht komplizierte Weise ein. Sogar der innere Aufbau geschachtelter Strukturen und der Gleitkommatypus eines `FLOAT`-Objektes (IEEE long, IEEE short, RTOS) geht dabei mit ein. Man kann die Signatur leider nicht durch eigene Rechnung bestimmen, sondern muß den PEARL90- Compiler für diese Aufgabe einsetzen. Man kodiert dazu die Prozedurdeklaration in PEARL90 und übersetzt sie im „`/*+P */`-Mode. Man erhält dann beim 68k-Compiler etwa folgendes Protokoll:

```
=    1 MODULE TEST;/*+P */
=    2 PROBLEM;
001C: >>Check signature:F175106F
0028: PHDR N/2,
004A: PNTR FFF4 ,
=    3 XYZ: PROC((I1,I2) FIXED) RETURNS(FIXED);
004C: MOVW D0 ,I2  X16 LOC FFD6(WL) ,
0050: ADDW D0 ,I1  X16 LOC FFD8(WL) ,
0054: MOVX A0 ,#fretv  X16 LOC FFDA(WL) ,
0058: MOVW (A0) ,D0 ,
005A: RETN
006C: RETN
007E: >>VALUE 0000=>N/
007E: >>VALUE 0022=>N/2
007E: >>ESL
=    4 RETURN(I1+I2); END;
007E: >>CON-BLK
007E: >>MODEND
=    5 MODEND;
```

Beim PowerPC-Compiler – der ja nur einen anderen Codegenerator verwendet – sieht die entsprechende Sequenz wie folgt aus:

```
=    1 MODULE TEST;/*+P */
=    2 PROBLEM;
001C: >>Check signature:F175106F
0028: phdr N/2,
0076: pntr FFF4 ,
=    3 XYZ: PROC((I1,I2) FIXED) RETURNS(FIXED);
0078: movw r0 ,I2  X16 LOC FFD4(r13) ,
007C: addw r0 ,I1  X16 LOC FFD6(r13) ,
0084: movx r8 ,#fretv  X16 LOC FFD8(r13) ,
0088: movw (r8) ,r0 ,
008C: retn
00B0: retn
00D4: >>VALUE 0000=>N/
00D4: >>VALUE 0024=>N/2
00D4: >>ESL
=    4 RETURN(I1+I2); END;
00D4: >>CON-BLK
00D4: >>MODEND
=    5 MODEND;
```

In beiden Fällen finden wir die gleiche hier interessierende Zeile

```
>>Check signature:F175106F
```

aus der wir den Parameter für das `SIGCHK`-Format ablesen können.

Der Signaturcheck verbraucht relativ wenig Zeit, weil es nur ein einfacher 32-Bit Compare mit konditioniertem Trapaufruf ist. Da auch der Feldindextester sehr viel schneller als in der PEARL80-Welt geworden ist, kann in vielen Fällen der Testmode des Compilers sogar in der endgültigen Version eines Programmes belassen werden.

Natürlich kann man auch signaturlose Unterprogramme schreiben, zum Teil wird das von anderen Übersetzern (etwa IEP–C) offenbar auch ausgenutzt. In diesem Fall wird das `SIGCHK`-Format einfach durch eine 12 Byte lange Sequenz von `NOP`-Befehlen ersetzt. Jetzt muß freilich sehr viel mehr Aufwand in die Programmentwicklungssystematik gesteckt werden, denn falsch spezifizierte signaturlose Unterprogramme können neben Fehlfunktionen sehr gefährliche Seitenwirkungen verursachen.

Einige Systemprogramme akzeptieren auch Signaturen aus einem Tabellenvorrat – etwa weil es ihnen egal ist, ob eine Datenstation nur für Ausgabe, nur für Eingabe oder für beide Richtungen spezifiziert wurde. Dies wird dann durch einen entsprechenden Wegsprung von der um 12 reduzierten Einstiegadresse kodiert. Dabei wird ausgenutzt, daß der Compiler im Testmode eine Versorgung des Registers D0 (bzw. r0) mit der Signatur unmittelbar vor dem Sprung auf den Signatur–Entry generiert.

Vorsicht!

Man kann die Signatur nicht verwenden, um den aktuellen Parametertyp festzustellen! Wenn der Testmode nicht eingeschaltet ist, wird die Signatur nämlich auf der Aufruferseite gar nicht berechnet und das Register D0 (bzw. r0) ist undefiniert.

8.4.3 Der Feldbeschreibungsblock

PEARL90 verwendet einen gegenüber PEARL80 erheblich erweiterten Feldbeschreibungsblock. Er ermöglicht eine praktisch kaum begrenzte Zahl von Dimensionen, nicht bei 1 beginnende Indizes sowie eine volle 32-bit Feldadressierung.

Jede Zeigervariable vom Typ „array“ ist identisch zum hier beschriebenen Feldbeschreibungsblock. Auch Felder als Prozedurparameter und statisch (evtl. global) definierte Felder werden durch einen solchen Feldbeschreibungsblock repräsentiert. Bei lokalen Feldern innerhalb von Tasks oder Prozeduren sowie bei Feldern innerhalb von Strukturen sind dem Compiler sämtliche Daten bekannt und es gibt zunächst nur compilerintern einen Feldbeschreibungsblock. Der Compiler generiert aber bei der Übergabe solcher Felder an Prozeduren oder Zuweisungen an eine Zeigervariable automatisch ebenfalls ein solches zur Laufzeit existentes Datenobjekt.

PEARL verwendet bekanntlich das Prinzip der „mitreisenden Felddeskriptoren“, welches viele Probleme und Fehlerquellen anderer Sprachen – besonders bei C und auch C++ – vermeidet. Leider stellt man bei Umsteigern von diesen Sprachen immer wieder fest, daß sie diese Art der Objektorientierung gar nicht oder erst sehr spät schätzen lernen. Dabei ist der Zeitverlust der Methode durchaus vernachlässigbar.

Offset	Bedeutung des 32-Bit Objektes
0	Physikalische Speicheradresse des ersten Elementes
4	Anzahl der Dimensionen des Feldes
8	Gesamtzahl der Feldelemente – 1. Diese Information verwertet der Indextester.
12	Minimal zulässiger linearer Index. Bezeichnet den Versatz (in Feldelementen, nicht in Bytes!), den das Feldelement (0,0,0,...,0) gegenüber der Feldanfangsadresse besitzt. Hier steht bei Feldern, die sämtliche Startindizes 0 haben (wie bei C), logischerweise eine exakte 32-bit 0. Diese Information wird bei der Feldformel und vom Indextester benutzt.
16	Untergrenze des ersten Index
20	Obergrenze erster Index – Untergrenze erster Index +1
24	Untergrenze des zweiten Index (falls vorhanden)
28	Obergrenze zweiter Index – Untergrenze zweiter Index +1
...	usw. für alle folgenden Indizes jeweils ein Pärchen

Tabelle 8.10: Der Feldbeschreibungsblock in PEARL90

Am sinnvollsten ist es, wenn man sich an Hand obiger Angaben mit Hilfe der `/*+P .. */`-Option des Compilers die Feldbeschreibungsblöcke statisch (auf Modulebene) deklarierter Felder in der Praxis ansieht. Auch ein Studium der Feldzugriffsformel ist damit möglich. Dabei sollte man keine konstanten Indizes verwenden, weil sonst Rechnungen vom Compiler wegoptimiert werden können!

Wenn eine Prozedur als Aktualparameter ein komplettes Feld übergeben bekommt, so steht der Feldbeschreibungsblock direkt im A2-Space. Seine Anfangsadresse relativ zu A2 erhält man wie üblich nur durch Kenntnis der Objektlänge *Len*, die sich nach folgender Formel berechnet:

$$Len = 16 + Ndim * 8$$

Dabei ist $Ndim$ die Anzahl der Dimensionen des Feldes. Da beim Prozeduraufruf der Raum im A2-Space auf 16 kB beschränkt ist, kann es bei Mutwilligkeit passieren, daß bei zu vielen Parametern und/oder zu vielen Feldern mit zu vielen Dimensionen (mehr als 4 machen normalerweise in der Wissenschaft kaum Sinn!) der Compiler beim Prozeduraufruf einen Kapazitätsüberlauf anzeigt, weil die 16 kB-Grenze überschritten wird.

Wenn Sie selbst Zugriffsformeln in Assembler kodieren, denken Sie bitte daran, daß der zunächst berechnete lineare Index noch mit der Anzahl Bytes, die ein einzelnes Feldelement belegt, multipliziert werden muß.

8.5 Parameterübergabe im alten PEARL80

Von einer Neukodierung von Assemblerroutinen für das PEARL80-System wird dringend abgeraten. Die folgenden Angaben aus einer alten Handbuchversion werden nur noch zur Information wiedergegeben und können als Hilfestellung dienen, wenn alte assemblerkodierte PEARL80-Routinen vorhanden sind, die auf den T–Code der PEARL90–Welt umgestellt werden müssen.

Die Übergabe der Programmkontrolle an das Unterprogramm (UP) erfolgte im PEARL80-System mit Hilfe virtueller Befehle, die vom PEARL-Laufzeitsystem (Hyperprozessor) exekutiert werden. Wir betrachten hierzu den Fall

CALL ABCD(para-list);

wobei „`ABCD`“ ein Assemblerunterprogramm sein soll; das Symbol `ABCD` muß im aufrufenden PEARL-Programm als „`ENTRY GLOBAL`“ spezifiziert sein. Auf der Assemblerseite wird `ABCD` durch Voranstellung des Zeichens `'>'` global deklariert. Das PEARL-Run-time betritt das UP im virtuellen Mode an der Stelle „`>ABCD`“, unser UP muß also in jedem Fall mit einem virtuellen Befehl beginnen, wobei dies ggf. der Befehl „`V0`“ zum Abschalten des Hyperprozessors sein kann. Die weiteren Randbedingungen beim Eintritt sehen wie folgt aus:

`A0`	zeigt auf die Stelle, an der das Parameterlink beginnt.
`D2`	enthält das Register `A5` (=„WL“) des Aufrufers.
`A5`	ist noch identisch mit `D2`.

Wichtige Hinweise

Die Register `A4`, `A5`, `A6`, `A7` dürfen im Laufe des UP nicht vom Programmierer verändert werden. Kurzzeitige Veränderung ist zwar in gewissen Fällen möglich, wird aber nicht empfohlen. `A7` ist mit max. 2 BSR-Level (8 Bytes) belastbar, es muß beim UP-Austritt oder vor der Benutzung irgendwelcher virtuellen Befehle unbedingt wieder auf dem Eingangswert stehen.

Bitte denken Sie auch daran, daß es Ihre eigene Aufgabe ist, dafür zu sorgen, daß das UP „`REENTRANT`“ ist. Es dürfen also keine statisch allokierten Objekte im UP verändert werden, wenn Sie diese Bedingung einhalten wollen.

Im folgenden sollen 2 Fälle unterschiedlicher Schwierigkeit betrachtet werden:

Fall A (nicht für Neuentwicklung!)

Es sollen weder Parameter noch Funktionswerte übergeben werden. Die Prozedur benötigt außer einigen wenigen Registern keinen eigenen Arbeitsspeicher. Ein Transfer von Werten ist über globale PEARL-Objekte dennoch möglich.

Auf der PEARL-Seite generiert der Compiler aus `CALL ABCD;` die (virtuelle) Sequenz (Liste aller virtuellen Befehle siehe Abschnitt 8.7):

```
PROC  >ABCD            (PROC=V18)
EPAR                   (EPAR=V19)
```

Auf der Assembler-UP-Seite programmieren wir (ein Module-Head gem. Seite 446 ist nicht dargestellt):

```
>ABCD   V0                    Hyperproc. off
        CMPI.B  =19,(A0)      Vergl.auf EPAR
        BNE     ...           Error - Zweig
        ....                  68000-Usr-code
        ....                  PEARL-Assembler-UP
        ....
*                     D0 ... D7 zur freien Verfuegung
*                     A1 ... A3   -----  " -----
        ....
        JMP     2(A0)         Ruecksprung HP
        ....
```

Der Fall A ist die schnellste Form des Anschlusses überhaupt. Notfalls kann auf das Überprüfen der leeren Parameterliste (`CMPI.B ...`) auch noch verzichtet werden. Wenn Sie z. B. in dem UP Floatingbefehle benötigen, so muß zuvor der Hyperprozessor wieder angeworfen werden. Dies gelingt mit Hilfe des dem UH-Assembler bekannten Befehles `TV`, der nur ein bequemer Mnemo für den Trap 14 ist.

Fall B (nicht für Neuentwicklung!)

Es sollen Parameter und Funktionswerte übergeben werden. Außerdem nehmen wir an, daß unser UP zusätzlichen internen Arbeitsspeicher benötigt, dabei aber „reentrant“ sein soll. Wir betrachten wiederum zunächst die PEARL-Seite:

```
SPC ABCD ENTRY(FIXED,FLOAT IDENT) RETURNS(FLOAT(55)) GLOBAL;

   ......

Z=ABCD(3,Y); /* Der P80-Compiler generiert hieraus: */

        PROC  >ABCD                (V18=PROC)
        INVW  Xcon=3               (V14=INVW)
        VARF  Y                    (V12=VARF)
        VARD  resultcell           (V13=VARD)
        EPAR                       (V19=EPAR)
        ...
        MOVD  resultcell,Z        (MOVD ist ein Macro, 2x MOVE.L)
```

Beim Studium des Codes wird erkennbar, daß Funktions-UPs sich nur durch einen zusätzlich angehängten Übergabeparameter von den durch „`CALL`“ aufrufbaren UPs unterscheiden. Der Rückgabewert muß in eine vom Compiler beschaffte Zelle geschrieben werden.

Vor der Codierung unseres Assembler-UP muß zunächst bestimmt werden, wieviel Prozedurworkspace „PWSP“ gebraucht wird. In diesem PWSP müssen auch die Parameterwerte oder - bei „IDENT“ - die vier Byte langen Adressen der Objekte untergebracht werden. Sollen Felder übergeben werden, so müssen auch Kopien der Feldbeschreibungsblöcke (s. unten) darin Platz finden.

In unserem Beispiel sei angenommen, daß das UP intern das 4 Byte lange Objekt „`HILF1`„ und das 8 Byte lange Objekt „`HILF2`“ benötigt. Wir codieren:

```
RETN   OPD     $4E4C      Opcode  definition   =  TRAP 12
ENTR   OPD.V   29         Opcode proc-entry  (Hyperproc.)
INVW   OPD.V   14         Invariant 16 bit  fixed  =  V14
VARF   OPD.V   12         Variable Float 32 bit    =  V12
VARD   OPD.V   13         Variable Float 64 bit    =  V13
EPAR   OPD.V   19         End of parameter Xfer    =  V19
*      PWSP-Allocation
PAR1   EQU     0             2 bytes positioned to  0(A5)
PAR2   EQU     PAR1+2        4 bytes (adr) pos. to  2(A5)
WERT   EQU     PAR2+4        4 bytes (adr) pos. to  6(A5)
HILF1  EQU     WERT+4        4 bytes (internal) to 10(A5)
HILF2  EQU     HILF1+4       8 bytes (internal) to 14(A5)
WSPSZ  EQU     HILF2+8       Total Size of  Procedure-WSP
*
>ABCD  ENTR    WSPSZ.L     Fetch  storage  from 'RTOS-UH'
       INVW    PAR1.X      xfer 16-bit FIXED  by  'value'
       VARF    PAR2.Z      xfer 32-bit FLOAT  by  'Ident'
       VARD    WERT.Z      xfer 64-bit FLOAT  by  'Ident'
       EPAR                End of param., + hyperproc off
       ....
*      reeller 68000-code. D0...A3 sind  frei verfuegbar.
*      A5  zeigt auf Ablageplatz  des  Wertes  von  PAR1.
*      Beispielhafter Zugriff auf die Objekte:
       MOVE.L  =$01400000,HILF1(A5)  oder (gleichwertig):
       MOVE.L  =$01400000,HILF1.X
       MOVE    PAR1.X,D4   Die Zahl 3 des obigen Bsp.->D4
       ....
       MOVEA.L PAR2.X,A0   Zeiger laden, da  IDENT  xfer!
       MOVE.L  =$01400000,(A0)  Wert-> 'Y' des Beispieles
       ....
       MOVEA.L WERT.X,A0   Zeiger auf Ergebniszelle laden
       MOVE.L  HILF2.X,(A0)+  xfer Bytes  1..4   Ergebnis
       MOVE.L  HILF2+4.X,(A0)  ''         5..8      ''
       ....
*      Bsp. letzte Sequenz unter Verwendung des Hyperproc
       TV                     Hyperproc hier  'anwerfen'
       V1     WERT.Z,HILF2.X  (Reihenfolge: dest,source)
       ....                   evtl. andere virt. Befehle
       V0                     Hyperproc  hier abschalten
       ....                   evtl.  weiterer  Realcode
       RETN                   Realer Befehl, exit des UP
       END
```

Man beachte dabei, daß das Verlassen des UP nur mit dem „RETN-Trap“ möglich ist. Dieser muß naturgemäß im Bereich realen Maschinencodes stehen.

Die Verwendung des V1-Befehles, der auch als MOVU OPD.V 1 hätte deklariert werden können, dient nur der Demonstration, da der Befehl inzwischen von den Compilern nicht mehr benutzt wird. (s. Liste in Abschnitt 8.7). MERKE: Hilfszellen und per „value“ übergebene Objekte werden mit „.X oder (nur bei realen Befehlen möglich) „...(A5)“ adressiert. Durch Zeiger vertretene Objekte werden bei realen Befehlen durch Laden eines Adreßregisters und bei virtuellen Befehlen durch die „.Z-Adressierung“ erreicht. Bei den Parameterübergabebefehlen im Kopf des UP wird durch .Z der Ident- und durch .X der Valuemode der Übergabe festgelegt. Bei Identmode darf das Objekt der Aufrufseite nicht invariant sein, sonst wird wie bei falschem Parameterdatentyp ein Laufzeitfehler angezeigt. Grundsätzlich kann man die Übergabe auch ohne den Hyperprozessor schaffen, wenn der Aufbau der Aufrufseite durch das UP selbst interpretiert wird.

Feldbeschreibungsblock (alt, PEARL80)

Beim Zugriff auf in der PEARL80-Welt global deklarierte Felder sind einige Besonderheiten zu beachten, da dem eigentlichen Variablenfeld im Speicher ein Vorspann (Feldbeschreibungsblock) vorangestellt ist. Aus

DCL A(bound1,bound2,bound3) ... GLOBAL wird im Speicher

```
>A        DC.W  bound3                          2 Bytes
          DC.W  bound2                          2 Bytes
          DC.W  bound1                          2 Bytes
          DC.L  offset+adr                      4 Bytes
          DS    (Speicherbereich des Feldes)  size Bytes
```

Bei weniger als 3 Dimensionen entfällt die Ablage der nicht angegebenen Feldgrenzen. Die Werte von bound1 etc. sind mit der „Global+offset“-Option des UH-Assemblers ohne Probleme adressierbar. Die 32-bit-zahl „offset+adr“ gibt die physikalische Adresse an, auf der das Feldobjekt A(0,0,0) (welches nicht existiert) stehen würde. Damit wird die Indexformel beim Feldzugriff durch den Compiler (und den Assemblerprogrammierer) schneller und kürzer. Bei Unklarheiten ist ein Studium des vom Compiler generierten Codes (/*+P*/-Option) sehr zu empfehlen.

`FIXED` und `FLOAT` Parameterbefehle (nicht für Neuentwicklung!)

Diese Befehle haben jeweils nur einen Operanden, der mit `.X` oder `.Z` Adressierung (s. o.) angesprochen wird.

INVD	V17	Invariant	Float (55)	=	8 Byte Float
INVF	V16	Invariant	Float (23)	=	4 Byte Float
INVW	V14	Invariant	Fixed (15)	=	2 Byte Fixed
INVX	V15	Invariant	Fixed (31)	=	4 Byte Fixed
VARD	V13	Variable	Float (55)	=	8 Byte Float
VARF	V12	Variable	Float (23)	=	4 Byte Float
VARW	V10	Variable	Fixed (15)	=	2 Byte Fixed
VARX	V11	Variable	Fixed (31)	=	4 Byte Fixed

Skalare Parameter außer `FIXED`, `FLOAT` (nicht für Neuentwicklung!)

Die seltener auftretenden Parametertypen werden durch etwas längeren Code übergeben. Für alle restlichen Datentypen außer `FIXED` und `FLOAT` steht dafür nur ein gemeinsamer virtueller Befehl zur Verfügung:

`MPXF OPD.V 139` „Miscellaneous parameter xfer“

Der `MPXF`-Befehl hat 2 Operanden, von denen einer für den Objekttransfer und der zweite für die Typprüfung nötig ist.

Anwendung: `MPXF dtyp,object.X` oder `...object.Z`

Die Information auf der Adresse `dtyp` besteht aus 2 oder 4 aufeinanderfolgenden Bytes. Sie haben folgende Bedeutung:

Byte	Code	Bedeutung
1	$xx	Länge (1...255) bei CHAR und (1...32) bei BIT
	$00	Bei den andern Datentypen, DUR etc.
2	$08	CHAR variable
	$09	INV CHAR
	$0A	BIT(1...16) variable
	$0B	INV BIT(1...16)
	$0C	BIT(17...32) variable
	$0D	INV BIT(17...32)
	$0E	DURATION variable
	$0F	INV DURATION
	$10	CLOCK variable
	$11	INV CLOCK
	$12	STRUCT variable
	$13	INV STRUCT
3	$xx	Nur bei STRUCT: High byte of obj.length
4	$yy	Nur bei STRUCT: Low byte of obj. length

Wie man richtig vermutet, wird beim Transfer von Strukturen nicht die innere Typuntergliederung geprüft, sondern lediglich eine Prüfung der Länge in Bytes ausgeführt.

Beispiel:

```
MPXF  OPD.V 139
      ....
      MPXF  C7,TEXT.X    'Value'-Uebergabe  String.
      MPXF  B16,MASK.Z   'Ident'-Uebergabe  Bitmask
      ....
C7    DC.B  7,$09        CHAR(7)  invariant object.
B16   DC.B  16,$0A       BIT(16)  variable  object.
      ....
*     Fuer MASK muessen 4 Byte (Adresse!) und fuer
*     TEXT 7 Bytes im PWSP vorgesehen sein.
*     Bei 'value'-Uebergabe findet ggf. eine Anpas-
*     sung der CHAR-Laengen statt.
```

Felder vom Typ „`no string`“ als Parameter (nicht für Neuentwicklung!)

Mit dem Befehl `ARNS` (Array no string) werden Felder der Datentypen `FIXED`, `FLOAT`, `DURATION`, `CLOCK` und `STRUCT` transferiert. Beispielprogramm:

```
     ARNS  OPD.V  122      'Array no string' 3 Operanden.
           ....
           ARNS  offs+adr-cell,boundlistcells,descr.mask
           ....
           ARNS  OFFS.X,BOUNDS.X,$0812.X    (2dim DUR-Feld)
```

Für OFFS sind 4 Bytes bereitzustellen, für die Liste der „bounds“ (2·Anz. Dimensionen) Bytes. Die Feldgrenzen werden in der Reihenfolge des Feldbeschreibungsblockes abgelegt. Auf `OFFS` steht die (fiktive) Speicheradresse des Feldelementes (0) bzw. (0,0) oder (0,0,0). Wenn `OFFS` unmittelbar hinter `BOUNDS` liegt, entsteht somit ein kompletter neuer Feldbeschreibungsblock.

Die „descriptionmask“ enthält Informationen über die Anzahl der Dimensionen und den Datentyp wie folgt:

Typ	1 dim var,inv	2 dim var,inv	3 dim var,inv
FIXED(15)	`$0401,$0541`	`$0801,$0941`	`$0C01,$0D41`
FIXED(31)	`$0402,$0542`	`$0802,$0942`	`$0C02,$0D42`
FLOAT(23)	`$0404,$0544`	`$0804,$0944`	`$0C04,$0D44`
FLOAT(55)	`$0408,$0548`	`$0808,$0948`	`$0C08,$0D48`
DURATION	`$0412,$0552`	`$0812,$0952`	`$0C12,$0D52`
CLOCK	`$0416,$0556`	`$0816,$0956`	`$0C16,$0D56`
STRUCT	`$0415,$0555`	`$0815,$0955`	`$0C15,$0D55`

Bekanntlich ist der Feldtransfer nur im „`IDENT`“-Mode möglich, so daß es keine weitere Fallunterscheidung wie bei den Skalaren gibt.

Bei den Verbundobjekten (`STRUCT`) folgt dem `ARNS` noch eine Längenüberprüfung `LTST` als weiterer Parameterübergabebefehl. Er hat einen Operanden, der unmittelbar die Anzahl Bytes, aus der die Struktur besteht, angibt.

```
LTST  OPD.V  72               Definition LTST
      ARNS   ...,...,$0415.X  Uebergabe Verbundfeld.
      LTST   25.X             Verbundtyp besteht aus 25 Bytes.
```

Felder der Typen `BIT` und `CHAR` als Parameter (nicht für Neuentwicklung!)

Hierfür sind die Befehle `ARBS` (Bitstring) und `ARCS` (Charstring) vorgesehen. Wir studieren eine beispielhafte Anwendung:

```
ARBS   OPD.V  123      'array bitstring',  3 Operanden
ARCS   OPD.V  124      'array char.string',3 Operanden

ARCS   offs+adr-cell,boundlist-cells,descr.mask
ARBS                    ''

ARCS   OFFS.X,BOUNDS.X,$0812.X     = ..(,) CHAR(18)
ARBS   OFF2.X,BOU2.X,$0C09.X       = ..(,,)  BIT(9)
```

Es gelten sinngemäß die gleichen Bedingungen wie bei der Instruktion `ARNS`; im Gegensatz zum `ARNS` enthält die „description-mask“ nun im rechten Byte die Länge in Bytes (`CHAR`) oder in Bits (`BIT`) des Feldelementes. Im Einzelfall ist wie folgt zu kodieren:

Typ	Befehl	1 dim var,inv	2 dim var,inv	3 dim var,inv
BIT(xx)	ARBS	`$04xx,$05xx`	`$08xx,$09xx`	`$0Cxx,$0Dxx`
CHAR(yy)	ARCS	`$04yy,$05yy`	`$08yy,$09yy`	`$0Cyy,$0Dyy`

Datenstationen als Parameter (nicht für Neuentwicklung!)

Die Übergabe von Stationen ist nur im Ident-Mode möglich. Dafür ist der Befehl „DMYD“ (Dummy Dation) vorhanden:

```
DMYD   OPD.V  137           Dationuebergabe,  1 Operand
       ....
       DMYD   adrptr.X      Uebergabeparameter ist  die
                            Adresse des Dation-Blockes
       ...                  Aufbau s.u.
       MOVEA.L adrptr.Z,A1  Dation-Blockadresse holen
*      ...
*      Es gab einen fehlerhaften Transfer, was nun ?
       MOVE    TFU(A1),D7   Hole erstes Wort fuer test
       BMI.S   neset        B: NE-option ist gesetzt
       ...
*      NE-option war nicht gesetzt, gebe Meldung aus
       ERROR                 setze Trap ab
       DC     $1234          irgend einen Code
       ...
neset  ...                  Weiter mit irgendwas
       RTS
```

Aufbau des PEARL-Dation Blockes

Mnemo	offs	len	Bedeutung
DIOFAC	0	2	I/O Facility für das Laufzeitsystem. Bit 2: End-of-File Bit
DLDN	2	1	aktuelle LDN der Dation
DDRIVE	3	1	aktuelle Drive-Nummer der Dation
DSTAT	4	2	Status des I/O-Transfers (hier greift ST() zu)
DTFU	6	2	Transferlänge der Dation (max. 32 Kbyte, oberstes Bit ist NE-Flag)
DINFO	8	2	Zelle für die aktuellen AI- bzw. MB-Parameter
DPATH	10	??	Pathlist ohne den Hardwarenamen der Dation, wird durch $FF beendet. Am Ende folgt noch ein $FE, welches niemals überschriebn werden darf, denn es markiert das Ende des vom Compiler vorgesehen PLatzes für die pathlist.

8.6 Umstellung von alten Assemblerunterprogrammen auf PEARL90

Diese Anleitung soll dazu dienen, vorhandene P80-Assemblerprozeduren so zu verändern, daß sie sowohl in der alten P80- als auch in der neuen PEARL90-Welt benutzt werden können. Für die Kodierung neuer, nur für PEARL90 ausgelegter Routinen ist sie nicht gut geeignet. Dazu wurde ja bereits eine vollständige Erläuterung ab Seite 566 gegeben. Die folgenden Seiten enthalten zum Teil bereits Bekanntes, denn sie sollen auch „stand alone“ aus dem Handbuch herauskopiert verstanden werden können.

Wenn Sie die empfohlenen Formate und `INCLUDE`-Files benutzen, werden Ihre Routinen mit hoher Wahrscheinlichkeit problemlos transferassemblier sein und sind damit auch auf anderen RTOS-Hardwareplattformen, z.B. dem PowerPC, direkt einsetzbar.

Bei kleinen Prozeduren mit nur ca. einer Seite Assemblertext ist es fast immer günstiger, den eigentlichen Kode der Originalroutine zu duplizieren und mit eigenem Vor- und Nachspann in eine *reinrassige* PEARL90-Version umzuwandeln. Diese kann dann einfach neben die unveränderte alte Routine gestellt werden.

Größere Anpassungsarbeiten gibt es allenfalls bei der Benutzung von Feldern als Parameter, da man hier den wesentlich üppigeren P90-Feldbeschreibungsblock erst auf den alten P80-Block umfingern muß. Dabei gehen natürlich alle in P80 nicht möglichen Optionen (etwa nicht bei 1 beginnende Untergrenzen) verloren. Probleme bereiten auch die sehr seltenen Fälle, bei denen eine formatierte Ein-/Ausgabe (d.h. `PUT`, `GET` oder `CONVERT`) mit den Hyperprozessorbefehlen innerhalb der Routine nachgebildet wird, da die dafür nötige Zelle `BWIO` auf `-4(A5)` bei dieser rigiden Lösung nicht angelegt wird.

Prinzip der Parameterübergabe im PEARL90-System.

Der P90-Compiler benutzt ausschließlich reellen Code, eine *Übergabe* im früheren Sinne gibt es nicht mehr. Die meiste Arbeit wird auf der Aufruferseite erledigt. Dies vereinfacht das Kodieren von Assemblerroutinen erheblich und führt zu einem sehr deutlich schnelleren Laufzeitverhalten. Der Compiler arbeitet auf der Aufruferseite mit dem sogenannten A2-Space, in dem er Objekte oder deren Adressen bereitstellt.

Wenn die aufzurufende Routine nur einen minimalen privaten Workspace von 24 oder weniger Bytes benötigt, so kann sie auch direkt diesen A2-Space in einen A5-Space umarbeiten. Dieser Sonderfall verursacht noch einmal erheblich kleinere Ein- und Ausstiegszeiten. Er wird später behandelt.

Man beachte folgende Regeln:

- Jede Prozedur besitzt eine sogenannte *Signatur*. Das ist ein 32 Bit langes Muster, welches mit hoher Wahrscheinlichkeit (aber nicht sicher) nur dieser einen Prozedurspezifikation zugeordnet ist. (Für Insider: Berechnung nach der Polynomformel für einen 31 Bit Galoiskörper). Diese Signatur wird nur beim Testmode benötigt, um eventuell falsche Spezifikationen der externen Prozedur entdecken zu können. Die Signatur Ihrer Routine erhalten Sie mit Hilfe des Compilers (s.u.).

- Die Parameter werden in umgekehrter Reihenfolge im A2-Space (und auch im späteren normalen A5-Space) abgelegt. Am oberen Ende auf -4(A2) steht die Adresse, auf der der Rückgabewert abzulegen ist. Diese Zelle ist auch vorhanden, aber undefiniert, wenn die Prozedur keine Werte (oder Pointer) zurückgibt. Zur Prozedur

  ```
  X:PROC(A FIXED,B FLOAT,C CHAR(1) IDENT) RETURNS(FIXED);
  ```

  ergibt sich dann z.B. folgendes A2-Space-Layout:

`-4(A2)`	Adr. Rückgabewert (immer 4 Bytes)
`-6(A2)`	Wert des FIXED(15)-objektes A
`-10(A2)`	Wert des FLOAT(23)-objektes B
`-14(A2)`	Adresse der Variablen C, CHAR(1)

  Objekte, deren Länge in Bytes nicht durch 2 teilbar ist, werden durch Auffüllen so plaziert, daß folgende Objekte wieder auf geraden Adressen stehen.

- Adreßzeiger und Array-pointer werden mit ihrer jeweils benötigten Länge direkt im A2-Space abgelegt. Gleiches gilt für alle *per value* transferierten Objekte, solange sie nicht mehr als 256 Bytes benötigen. Objekte mit mehr als 256 Bytes werden auch im *per value* -Mode durch einen Pointer

(4 byte) repräsentiert; dieser Zeiger zeigt auf eine temporäre Kopie, die der Compilercode auf der Aufruferseite angefertigt hat und nun von der Routine ohne Auswirkungen auf das Original verändert werden darf.

- Der P90-Compiler generiert am Prozeduranfang hinter dem Signaturcheck das sogennante `PRODEC-`Format. Es hinterläßt die Parameter in gleicher Reihenfolge im A5-Space, allerdings noch weiter im negativen Bereich als im A2-Space. Dieser zusätzliche Versatz ist zwischen der 68K- und den RISC-Implementierungen unterschiedlich und wird bei reinrassigen P90-Prozeduren durch die Verwendung der Assembliervariablen `FLVA` (first local variable address) aus dem File `COMEQU` ausgeglichen (zur Zeit der Drucklegung `$22`=34 beim 68K und `$24`=36 beim PowerPC).

  Das hier vorgeschlagene spezielle `P8090-`Format schiebt im Gegensatz zum `PRODEC-`Format die Parameter scheinbar (also nicht durch Move-Befehle zur Laufzeit) in einen bei `0(A5)` beginnenden Prozedur-Workspace. Dies bedingt, daß die Zelle `BWIO` (Bufferpointer Workspace IO) auf `-4(A5)` nicht angelegt ist und formatierte PEARL-Ein-/Ausgabe mit nachgebildeten `PUT-`, `GET-` oder `CONVERT-`Anweisungen innerhalb der Routine nicht zugelassen ist. Zum Verlassen der Routine ist das reinrassige P90-Format `PROCEX` ebenfalls nicht geeignet, dafür ist in der Mischwelt das Format `X8090` zuständig.

- Die hier vorgeschlagenen Formate aus der Datei `PROCS.FOR` passen sich automatisch der Zielhardware an. Assembler und Transferassembler picken sich automatisch die richtigen Befehlssequenzen heraus. Verändern Sie bitte daher auf keinen Fall diese Datei!

Wir betrachten exemplarisch die Umstellung der Prozedur

```
X:PROC(A FIXED,B FLOAT,C CHAR(1) IDENT) RETURNS(FIXED);
```

Dazu hatten wir in der P80-Welt evtl. wie folgt kodiert:

```
AIN    EQU   0          Input para A
BIN    EQU   2          Input para B
CIN    EQU   6          Adr. von para C
RESLT  EQU   10         Result pointer
LORG   EQU   14         origin local wsp
MY_X   EQU   LORG       My own cell
....   ....  ..         Andere lokale Objekte
WSPSZ  EQU   LORG+100   Angenommene 100 locals
*
>X     ENTR  WSPSZ.L     Auftakt para xfer
       VARW  AIN.X       Value 16 bit fixed
       VARF  BIN.X       Value 32 bit float
```

```
        MPXF  C1,CIN.Z Ident char(1)
        VARW  RESLT.Z  Result pointer
        EPAR           End para-list
XCDE    MOVE  ....     Code von X
        ....           "
EXIT    RETN           Exit von X
```

Wir wollen den Code zwischen den Marken `XCDE` und `EXIT` unverändert lassen, aber der Routine einen zweiten PEARL90-Einstieg hinzufügen. Dazu verfahren wir in folgenden Schritten:

1. **Bestimmung der Signatur.**

   Wir kodieren das folgende PEARL-Programm:

   ```
   MODULE;PROBLEM;/*+P*/
   X:PROC(A FIXED,B FLOAT,C CHAR(1) IDENT) RETURNS(FIXED);
     RETURN(5);/*-P*/
   END;
   MODEND;
   ```

   Bei der Übersetzung im IEEE-Float-Modus (bitte beachten!) steht in der Zeile mit dem Signatur-check `$C5D74285`, bei Softfloat `$C5D54285`.

2. **Umordnen der Parameter-EQUs.**

   Die Parameter werden in neuer P90-Reihenfolge abgelegt. Diese Änderung sollte bei normaler Kodierung des Innenlebens von X keine Auswirkung haben. Außerdem müssen wir eine Flag einführen und den lokalen eigenen Workspace um `FLVA` hochschieben:

   ```
   CIN    EQU   0         Adr. von para C
   BIN    EQU   4         Input para B
   AIN    EQU   8         Input para A
   RESLT  EQU   10        Result pointer
   FLAG9  EQU   RESLT+4   Flag für P90
   LORG   EQU   FLAG9+FLVA Hochschieben
   MY_X   EQU   LORG      My own cell
   ....   ....  ..        Andere lokale Objekte
   WSPSZ  EQU   LORG+100  Liegt jetzt höher
   ```

   Bitte beachten: Das hier beispielhaft `FLAG9` genannte Objekt muß exakt 4 Bytes hinter dem Ergebniszeiger stehen! Die Zellen zwischen `FLAG9` und `LORG` dürfen nicht benutzt werden! Dort liegen interne Daten.

   Prozeduren ohne Parameterliste, die auch kein Ergebnis abliefern, setzen `FLAG9` wie folgt:

```
FLAG9 EQU   0          Flag für P90
```

3. **P90-Entry anlegen.**

   Wir kodieren einen zusätzlichen P90-Einstieg wie folgt:

```
      SIGCHK $C5D74285       Signature check
~X    P8090  WSPSZ,FLAG9     Aus PROCS.FOR
      BRA    XCDE            To common code
```

4. **Veränderung hinter dem EPAR.**

   Vor der Stelle XCDE muß die Flag gesetzt werden, an der das X8090-Format erkennt, daß die Routine im alten P80 Modus zu beenden ist:

```
>X    ENTR  WSPSZ.L   Auftakt para xfer
      VARW  AIN.X     Value 16 bit fixed
      VARF  BIN.X     Value 32 bit float
      MPXF  C1,CIN.Z  Ident char(1)
      VARW  RESLT.Z   Result pointer
      EPAR            End para-list
     _CLR.L FLAG9(A5) *** Neu ***
XCDE  MOVE ....       Code von X
```

   Das Zeichen '_' vor dem CLR.L ist ein Hinweis für den Transferassembler, daß der Statusregister-Update entfallen kann. Es wird vom 68K-Assembler ignoriert.

5. **Ersetzung aller RETN-Traps.**

   Alle RETN-Traps werden durch das Format X8090 ersetzt:

```
EXIT  X8090  FLAG9    Exit, A5-Bezug implizit.
```

Umstellung bei sehr kleinem Workspace.

Wie oben erwähnt, kann eventuell mit dem vom Compiler bereitgestellten A2-Space auch direkt gearbeitet werden, so daß ein A5-Space weder im vom Compiler angebotenen Raum noch als RTOS-Sektion angelegt werden muß. In diesem Fall müssen die *privaten* lokalen Objekte in den kleinen Raum hinter `FLAG9+8` passen, der bei allen RTOS-Varianten mindestens 24 Byte groß ist. Bei dieser Lösung wird also kein Workspace neu angelegt, so daß lediglich das Retten von A5 und das Neuladen wegen der P80-Kompatibilität erforderlich ist. Die Durchlaufzeit der Routine kann dadurch merkbar verkürzt werden: es wird nahezu die Geschwindigkeit des auf Seite 569 beschriebenen Sonderfalles errreicht.

Bedingung für diese Lösung ist allerdings, daß kein weiteres Unterprogramm von der Routine aufgerufen wird, ausgenommen interne Routinen, die mit BSR aufgerufen werden (Stackplatz reicht nur für einen weiteren Level!).

Die EQUs werden nun erneut geringfügig geändert und die Formate `P8090` sowie `X8090` werden durch ihre schnelleren und einfacheren Brüder `QP8090` sowie `QX8090` ersetzt. Das komplette Ergebnis sieht dann wie folgt aus:

```
CIN   EQU   0          Adr. von para C
BIN   EQU   4          Input para B
AIN   EQU   8          Input para A
RESLT EQU   10         Result pointer
FLAG9 EQU   RESLT+4    Flag für P90
      ....             4 Bytes intern
LORG  EQU   FLAG9+8    *** Neu ***
MY_X  EQU   LORG       My own cell
....  ....  ..         Andere lokale Objekte
WSPSZ EQU   LORG+24    *** Maximum! ***
*
      SIGCHK $C5D74285        Signature check
~X    QP8090 WSPSZ,FLAG9      *** Neu ***
      BRA    XCDE             To common code
*
>X    ENTR  WSPSZ.L  Auftakt para xfer
      VARW  AIN.X    Value 16 bit fixed
      VARF  BIN.X    Value 32 bit float
      MPXF  C1,CIN.Z Ident char(1)
      EPAR           End para-list
     _CLR.L FLAG9(A5) *** bleibt ***
XCDE  MOVE ....      Code von X
      ....            "
EXIT  QX8090 FLAG9   *** Neu ***
```

Bitte keinesfalls die Quick-Versionen mit den normalen Versionen kombinieren!

Wenn `WSPSZ` den zugelassenen Wert überschreitet, erzeugt der `FORMAT`-Prozessor des Assemblers bzw. Transferassemblers bei der Generierung des `QP8090` eine Fehlermeldung `/LIMIT/`.

```
*-----------------------------------------------*
*     D E M O : M I X E D  P 8 0 / P 9 0        *
*-----------------------------------------------*
*      Using the quick model with few locals    *
*      T-Code used                              *
*                                               *
      .INCLUDE PROCS.FOR     load Formats       *
*                                               *
CIN    EQU    0              ** adapted **      *
BIN    EQU    4              ** adapted **      *
AIN    EQU    8              ** adapted **      *
RESLT  EQU    10             ** adapted **      *
FLAG9  EQU    RESLT+4        **  new!   **      *
LORG   EQU    FLAG9+8  FLVA  ** adapted **      *
*...                                            *
WSPSZ  EQU    LORG+24        not changed        *
*                                               *
       DC      0,0,0,0,$0010 not changed        *
       DC.B    'AAABBB'      not changed        *
                                               *
      .IF_PROCTYPE  M68K  -> No P80 on PowerPC!! *
*                                               *
ENTR   OPD.V  29             not changed        *
VARW   OPD.V  10             not changed        *
VARF   OPD.V  12             not changed        *
MPXF   OPD.V  139            not changed        *
EPAR   OPD.V  19             not changed        *
*                                               *
>X     ENTR   WSPSZ.L        not changed        *
       VARW   AIN.X          not changed        *
       VARF   BIN.X          not changed        *
       MPXF   C1,CIN.Z       not changed        *
       VARW   RESLT.Z        not changed        *
       EPAR                  not changed        *
       CLR.L  FLAG9(A5)   *** P90FLAG,new!! **** *
      .FIN                *** End 68K,new!! **** *
*..... Result=TOFIXED(CIN)+AIN         .........*
XCDE   movea.l CIN(A5),A0    not changed        *
       clr.l   d0            not changed        *
       move.b  (A0),d0       not changed        *
      _add     AIN(A5),d0    not changed        *
       movea.l RESLT(A5),A0  not changed        *
```

```
      _move    d0,(A0)       not changed         *
EXIT  QX8090   FLAG9     *** was a RETN before! *
*                                                *
C1    DC    $0108          not changed         *
*................................................*
*..... Additional P90-Entry with IEEE-Float obj: *
*                                 ----------     *
      SIGCHK $C5D74285      *** new! *** (IEEE) *
~X    QP8090 WSPSZ,FLAG9    *** new! ***        *
      BRA    XCDE           *** new! ***        *
*                                                *
      END                   That's all          *
```

8.7 Hyperprozessorbefehle

Die virtuellen Maschinenbefehle des Laufzeitsystemes bilden den sogenannten „Hyperprozessor“. Die Benutzung der Befehle ist zwar auch dem Assemblerprogrammierer möglich, doch wird keine Gewähr für langfristige Unveränderlichkeit des Hyperprozessors gegeben. Im Zuge von Compilerverbesserungen können immer auch Veränderungen des Befehlssatzes auftreten. Man überzeuge sich daher genau, ob die aktuelle Variante noch alle benutzten Befehle enthält. Die Hyperprozessorbefehle können die Register `D0-A3` zerstören.

No.	X	Mnemo	Bemerkungen	*
V0	-	TOREAL	Abschalten des Hyperprozessors	*
V1	2	MOVU	Nicht für Neuentwicklung (Move 8 bytes)	*
V2	1	(ADDF)	Nicht für Neuentwicklung (Add Float 23)	*
V3	1	(ADDD)	Nicht für Neuentwicklung (Add Float 55)	*
V4	1	(SUBF)	Nicht für Neuentwicklung (Sub Float 23)	*
V5	1	(SUBD)	Nicht für Neuentwicklung (Sub Float 55)	*
V6	1	(MULX)	Nicht für Neuentwicklung (Mul Fixed 31)	*
V7	1	(MULD)	Nicht für Neuentwicklung (Mul Float 55)	*
V8	1	(DIVX)	Nicht für Neuentwicklung (Div Fixed 31)	*
V9	1	(DIVD)	Nicht für Neuentwicklung (Div Float 55)	*
V10	1	VARW	Param. XFER P80: Variable Fixed 15	*
V11	1	VARX	Param. XFER P80: Variable Fixed 31	*
V12	1	VARF	Param. XFER P80: Variable Float 23	*
V13	1	VARD	Param. XFER P80: Variable Float 55	*
V14	1	INVW	Param. XFER P80: Konstante Fixed 15	*
V15	1	INVX	Param. XFER P80: Konstante Fixed 31	*
V16	1	INVF	Param. XFER P80: Konstante Float 23	*
V17	1	INVD	Param. XFER P80: Konstante Float 55	*
V18	1	CALL	Call P80-procedure. (Im Comp. als PROC)	*
V19	-	EPAR	P80: End of Parameterlist + Hyperproc. off	*
V20	2	SHFS	Shift 1 TO 16 BIT Object	*
V21	2	SHFL	Shift 17 to 32 BIT Object	*
V22	-	CVXF	Nicht für Neuentwicklung, Fix31 to Flo23	*
V23	-	CVWD	Nicht für Neuentwicklung, Fix15 to Flo55	*
V24	-	CVXD	Nicht für Neuentwicklung, Fix31 to Flo55	*
V25	-	CVFD	Nicht für Neuentwicklung, Flo23 to Flo55	*
V26	4	CATC	Concatenation von 2 CHAR-strings	*
V27	3	PUT	Eröffnung einer Liste für PUT	*
V28	3	GET	Eröffnung einer Liste für GET	*
V29	1	ENTR	P80: Prozedureintritt mit PWSP-Erzeugung	*

V30	2	EFR2	E-(E/A)Format mit 2 Parametern	*
V31	3	EFR3	E-(E/A)Format mit 3 Parametern	*
V32	1	PAGE	PAGE-(E/A)Format	*
V33	1	XFOR	X-(E/A)Format	*
V34	1	RLFO	Remote-Left-Bracket in (E/A)Format	*
V35	1	EALW	E/A of Fixed15	*
V36	1	EALX	E/A of Fixed31	*
V37	1	EALF	E/A of Float23	*
V38	1	EALD	E/A of Float55	*
V39	1	EADU	E/A of Duration	*
V40	1	EACL	E/A of Clock	*
V41	-	LBRK	Left Bracket in (E/A)Format	*
V42	-	RBRK	Right Bracket in (E/A)Format	*
V43	-	FESP	(E/A)Format-End-specification	*
V44	1	FACT	(E/A)Format Wiederholfaktor	*
V45	1	AFOL	A-(E/A)Format	*
V46	-	LIFO	List-(E/A)Format	*
V47	1	SKFO	SKIP-(E/A)Format	*
V48	3	OPN3 **	OPEN 3 Param. (BY IDF(...))	*
V49	1	RWND **	REWIND Dation and open	*
V50	1	SYNC **	Synchronize Dation	*
V51	1	APND **	Append to a File	*
V52	1	EORL	EOR with 32 Bit Obj.	*
V53	2	CSHL	Cyclic Shift 17 to 32 Bit	*
V54	2	IOBS	I/O Bit-string	*
V55	2	CSHS	Cyclic Shift 1 to 16 Bit	*
V56	2	SEEK **	Seek a position in a file	*
V57	2	SAVP **	Save a position in a file	*
V58	-	----	Nicht mehr besetzt	*
V59	3	NEBS	Not equal for Bit-strings	*
V60	4	NECS	Not equal Character-string	*
V61	-	----	Nicht mehr besetzt	*
V62	-	----	Nicht mehr besetzt	*
V63	1	EQUD	Equal long Float	*
V64	1	SUSE	Suspend external (given Task)	*
V65	-	----	Nicht mehr besetzt	*
V66	1	NEQD	Not equal long Float	*
V67	-	----	Nicht besetzt	*
V68	-	----	Nicht besetzt	*
V69	-	----	Nicht besetzt	*
V70	-	----	Nicht besetzt	*

V71	-	----	Nicht besetzt	*
V72	1	LTST	P80: Length-test (Struct-param xfer)	*
V73	1	(LTHF)	Nicht für Neuentwicklung (LT Float(23))	*
V74	1	(LTHD)	Nicht für Neuentwicklung (LT Float(55))	*
V75	-	- - - -	Nicht mehr besetzt	*
V76	-	- - - -	Nicht mehr besetzt	*
V77	-	- - - -	Nicht mehr besetzt	*
V78	-	- - - -	Nicht mehr besetzt	*
V79	-	- - - -	Nicht mehr besetzt	*
V80	-	- - - -	Nicht mehr besetzt	*
V81	-	- - - -	Nicht mehr besetzt	*
V82	-	- - - -	Nicht mehr besetzt	*
V83	1	POWW	Power Fixed(15)	*
V84	1	POWX	Power Fixed(31)	*
V85	1	POWF	Power Float(23)	*
V86	1	POWD	Power Float(55)	*
V87	1	TERM	Terminate (given Task)	*
V88	1	PREV	Prevent (given Task)	*
V89	1	CONT	Continue (given Task)	*
V90	1	LIMV	Line-marker in virtual environment	*
V91	1	ODAT	Open Dation (no operation)	*
V92	1	CDAT	Close Dation (if closable)	*
V93	1	WCON	(When) ... Continue	*
V94	1	TCON	(Timed) ... Continue	*
V95	2	ACTI	Activate with priority	*
V96	2	WACT	(When) ... activate with priority	*
V97	2	TACT	(Timed) ... activate with priority	*
V98	-	WRES	(When) ... Resume (own Task)	*
V99	-	TRES	(Timed) ... Resume (own Task)	*
V100	-	STSC	Start schedule-definition	*
V101	1	ATCL	AT (clock) schedule param. set	*
V102	1	AFTR	AFTER (duration) schedule param. set	*
V103	1	ALLD	ALL (duration) schedule param. set	*
V104	1	UNIL	UNTIL (clock) schedule param. set	*
V105	1	DUDU	DURING (duration) schedule param. set	*
V106	1	WHEV	WHEN (event) schedule param. set	*
V107	3	WRIT	WRITE-Instruction	*
V108	3	READ	READ-Instruction	*
V109	-	----	Nicht mehr besetzt	*
V110	-	----	Nicht mehr besetzt	*

V111	1	EOLI	End I/O-List (escape-label)	*
V112	-	RFEN	Remote-(E/A)Format end	*
V113	2	EACS	E/A Character-string	*
V114	3	FFOR	F-(E/A)Format	*
V115	-	----	Nicht besetzt	*
V116	-	----	Nicht besetzt	*
V116	-	----	Nicht besetzt	*
V118	4	EQBS	Equal Bit-strings	*
V119	4	EQCS	Equal char. strings	*
V120	3	MVBS	Move Bit-string	*
V121	3	MVCS	Move character-string	*
V122	3	ARNS	P80: Array-param.xfer 'no string'	*
V123	3	ARBS	P80: Array-param.xfer 'bitstring'	*
V124	3	ARCS	P80: Array-param.xfer 'char.string'	*
V125	2	BFOR	B-(E/A)Format	*
V126	2	DFOR	D-(E/A)Format	*
V127	2	TFOR	T-(E/A)Format	*
V128	-	----	Nicht mehr besetzt	*
V129	-	----	Nicht mehr besetzt	*
V130	-	----	Nicht mehr besetzt	*
V131	1	(MULF)	Nicht für Neuentwicklung (MUL Float(23))	*
V132	1	(DIVF)	Nicht für Neuentwicklung (DIV Float(23))	*
V133	1	EORW	Excl.Or 32 Bit (Fehlt in 68000 Hardw.)	*
V134	-	(CVWF)	Nicht für Neuentwicklung (Fix15 to Fl23)	*
V135	1	(LDAD)	Nicht für Neuentwicklung (Load acc.8 by)	*
V136	1	(STAD)	Nicht für Neuentwicklung (St. accu 8 by)	*
V137	1	DMYD	P80: (Dummy)-Dation parameter xfer	*
V138	1	DMYI	P80: (Dummy)-Interrupt parameter xfer	*
V139	2	MPXF	P80: Miscellaneous Parameter X-fer	*
V140	-	(ABSF)	Nicht für Neuentwicklung (ABS accu Fl23)	*
V141	-	(ABSD)	Nicht für Neuentwicklung (ABS accu Fl55)	*
V142	-	(ENTI)	Nicht für Neuentwicklung (ENTIER)	*
V143	-	(ROUN)	Nicht für Neuentwicklung (ROUND)	*
V144	-	(SIGN)	Nicht für Neuentwicklung (SIGN)	*

Im Folgenden sind alle Befehle, die unsere Compiler erzeugen, in alphabetischer Reihenfolge aufgelistet.

Mnemo	X	Operation	Bemerkung	*
ABAL	-	ADD.L A5,D0	Add base adr long	*
ABSD	-	JSR -72(A6)	Abs Wert double float	*
(ABSD)	-	V141	Nicht für Neuentwicklung (ABS accu Fl55)	*
ABSF	-	JSR -68(A6)	Abs Float23	*
(ABSF)	-	V140	Nicht für Neuentwicklung (ABS accu Fl23)	*
AC-1	-	SUBQ =1,D0	Dekrement D0	*
ACTI	2	V95	Activate with priority	*
ADDA	-	ADD.L A1,D0	add address	*
(ADDD)	1	V3	Nicht für Neuentwicklung (Add Float 55)	*
ADDD	1	Macro	LEA obj,A1 + JSR -20(A6)	*
(ADDF)	1	V2	Nicht für Neuentwicklung (Add Float 23)	*
ADDF	1	Macro	LEA obj,A1 + JSR -4(A6)	*
ADDI	2	ADDI.W =con,xxx		*
ADDM	1	ADD.W D0,xxx		*
ADDW	1	ADD.W	Native Add	*
ADDX	1	ADD.L	Native Add long	*
ADIL	1	ADD.L =xx,D0	Add immediate long	*
AD+2	-	ADDQ.L =2,A1	Quick adr. shifter	*
AFOL	1	V45	A-(E/A)Format	*
AFTR	1	V102	AFTER (duration) schedule param. set	*
ALLD	1	V103	ALL (duration) schedule param. set	*
ANDW	1	AND.W	Native AND Word	*
ANDL	1	AND.L	Native AND Long	*
ANIW	1	ANDI.W =xx,D0		*
ANIL	1	ANDI.L =xx,D0		*
APND	1	V51 **	Append to a File	*
ARBS	3	V123	Array-param.xfer 'bitstring	*
ARCS	3	V124	Array-param.xfer 'char.string'	*
ARNS	3	V122	Array-param.xfer 'no string'	*
ATCL	1	V101	AT (clock) schedule param. set	*
BFOR	2	V125	B-(E/A)format	*
BGE4	-	BGE.S $+4		*
BGEL	1	BGE.L xxx	Conditioned branch ge	*
BGTL	1	BGT.L xxx	Conditioned branch gt	*
BLTL	1	BLT.L xxx	Conditioned branch lt	*

```
CALL    1   V18          Call procedure. (Im Comp. als PROC)      *
CASE    -   JMP -96(A6)       For CASE-construct                  *
CATC    4   V26          Concatenation von 2 CHAR-strings         *
CDAT    1   V92          Close Dation (if closable)               *
CLRL    1   CLR.L xx                                              *
CMPW    1   CMP.W        Native Compare to D0                     *
CMPL    1   CMP.L        Native Compare long to D0                *
CONT    1   V89          Continue (given Task)                    *
CSHL    2   V53          Cyclic Shift 17 to 32 Bit                *
CSHS    2   V55          Cyclic Shift 1 to 16 Bit                 *
CVBL    -   Macro        Convert Bitstring to long SWAP D0 + CLR  *
CVFD    -   CLR.L D1          Convert Flo23 to Flo55              *
(CVFD)  -   V25          Nicht für Neuentwicklung, Flo23 to Flo55 *
CVWD    -   JSR -56(A6)       Convert Fix15 to Flo55              *
(CVWD)  -   V23          Nicht für Neuentwicklung, Fix15 to Flo55 *
CVWF    -   JSR -64(A6)       Convert Fix15 to Flo23              *
(CVWF)  -   V134         Nicht für Neuentwicklung (Fix15 to Fl23) *
CVWX    -   EXT.L D0          Convert Fix15 to Fix31              *
CVXD    -   JSR -60(A6)       Convert Fix31 to Flo55              *
(CVXD)  -   V24          Nicht für Neuentwicklung, Fix31 to Flo55 *
CVXF    -   JSR -52(A6)       Convert Fix31 to Flo23              *
(CVXF)  -   V22          Nicht für Neuentwicklung, Fix31 to Flo23 *
D0A0    -   MOVEA.L D0,A0     Quickload of adr-reg (optim.)       *
D0A1    -   MOVEA.L D0,A1                                         *
D0*2    -   ADD D0,D0         Verdoppele D0                       *
D0*4    -   LSL =2,D0         Vervierfache D0                     *
D0*8    -   LSL =3,D0         Verachtfache D0                     *
DFOR    2   V126         D-(E/A)Format                            *
DISA    -   TRAP $A034        Disable Interrupt                   *
DIVD    1   Macro        LEA obj,A1 + JSR -32(A6)                 *
(DIVD)  1   V9           Nicht für Neuentwicklung (Div Float 55)  *
DIVF    1   Macro        LEA obj,A1 + JSR -16(A6)                 *
(DIVF)  1   V132         Nicht für Neuentwicklung (DIV Float(23)) *
DIVW    1   DIVS         Native divide                            *
DIVX    1   Macro        LEA obj,A1 + JSR -48(A6)                 *
(DIVX)  1   V8           Nicht für Neuentwicklung (Div Fixed 31)  *
DMYD    1   V137         (Dummy)-Dation parameter xfer            *
DMYI    1   V138         (Dummy)-Interrupt parameter xfer         *
DSCO    -   SUBQ =1,D2        Decrement Shift count one           *
DUDU    1   V105         DURING (duration) schedule param. set    *
```

EACL	1	V40	E/A of Clock	*
EACS	2	V113	E/A Character-string	*
EADU	1	V39	E/A of Duration	*
EALD	1	V38	E/A of Float55	*
EALF	1	V37	E/A of Float23	*
EALW	1	V35	E/A of Fixed15	*
EALX	1	V36	E/A of Fixed31	*
EFR2	2	V30	E-(E/A)Format mit 2 Parametern	*
EFR3	3	V31	E-(E/A)Format mit 3 Parametern	*
ENAB	-	TRAP $A032	Enable Interrupt	*
ENTI	-	JSR -76(A6)	ENTIER-Funktion	*
ENTI)	-	V142	Nicht für Neuentwicklung (ENTIER)	*
ENTR	1	V29	Prozedureintritt mit PWSP-Erzeugung	*
EOLI	1	V111	End I/O-List (escape-label)	*
EORL	1	V52	EOR with 32 Bit Obj.	*
EORW	1	V133	Excl.Or 16 Bit (Fehlt in 68000 Hardw.)	*
EPAR	-	V19	End of Parameterlist	*
EQBS	4	V118	Equal Bit-strings	*
EQCS	4	V119	Equal char. strings	*
EQUD	1	V63	Equal long Float	*
EQUF	1	Macro	Konstruktion mit CMP.L + versch. ff	*
EQUW	1	Macro	Konstruktion mit CMP.W + versch. ff	*
EQUX	1	Macro	Konstruktion mit CMP.L + versch. ff	*
EXIT	-	TRAP =1	Terminate own Task	*
EX20	-	EXG D2,D0	(in FOR ... BY variable ... REPEAT)	*
FACT	1	V44	(E/A)Format Wiederholfaktor	*
FESP	-	V43	(E/A)Format-End-specification	*
FFOR	3	V114	F-(E/A)Format	*
FLFR	-	JSR -88(A6)	Float to fraction conv. ('SEND')	*
FRFL	-	JSR -92(A6)	Fraction to Floating conv. ('TAKE')	*
GET	3	V28	Eröffnung einer Liste für GET	*
INVD	1	V17	Param. XFER: Konstante Float 55	*
INVF	1	V16	Param. XFER: Konstante Float 23	*
INVW	1	V14	Param. Xfer: Konstante Fixed 15	*
INVX	1	V15	Param. XFER: Konstante Fixed 31	*
IOBS	2	V54	I/O Bit-string	*
ITS1	-	TRAP $A040	Index-test 1-dim.	*
ITS2	-	TRAP $A042	Index-test 2-dim.	*
ITS3	-	TRAP $A044	Index-test 3-dim.	*

JRSI	-	JSR (A1)	Native pointered jump subroutine	*
JSR	1	JSR xxx	Native jump subroutine	*
LBRK	-	V41	Left Bracket in (E/A)Format	*
LDAD	1	V135	Nicht für Neuentwicklung (Load acc.8 by)	*
LDIL	1	MOVE.L =xx,D0	(Load immediate long)	*
LEFA	1	LEA xx,A1		*
LIFO	-	V46	List-(E/A)Format	*
LIMR	-	TRAP $A036	Line-marker in real environment	*
LIMV	1	V90	Line-marker in virtual environment	*
LTHD	1	Macro	LEA obj,A1 + JSR -40(A6)	*
(LTHD)	1	V74	Nicht für Neuentwicklung (LT Float(55))	*
LTHF	1	Macro	LEA obj,A1 + JSR -36(A6)	*
(LTHF)	1	V73	Nicht für Neuentwicklung (LT Float(23))	*
LTST	1	V72	Length-test (Struct-param xfer)	*
MMBY	2	Macro	Move multibyte, schnelle DBF-Konstrukt.	*
MOVB	?	MOVE.B	Native Move byte	*
MOVF	?	MOVE.L	Native Move long	*
MOVD	?	Macro	Verschiedene Sequenzen native Code	*
MOVU	2	V1	Nicht für Neuentwicklung (Move 8 bytes)	*
MOVW	?	MOVE.W	Native Move	*
MOVX	?	MOVE.L	Native Move long	*
MPXF	2	V139	Miscellaneous Parameter X-fer	*
MULD	1	Macro	LEA obj,A1 + JSR -28(A6)	*
(MULD)	1	V7	Nicht für Neuentwicklung (Mul Float 55)	*
MULF	1	Macro	LEA obj,A1 + JSR -12(A6)	*
(MULF)	1	V131	Nicht für Neuentwicklung (MUL Float(23))	*
MULU	1	MULU	Native Multiply 16 bit unsigned	*
MULW	1	MULS	Native Multiply 16 bit	*
MULX	1	Macro	LEA obj,A1 + JSR -44(A6)	*
(MULX)	1	V6	Nicht für Neuentwicklung (Mul Fixed 31)	*
MVBS	3	V120	Move Bit-string	*
MVCS	3	V121	Move character-string	*
NEBS	3	V59	Not equal for Bit-strings	*
NECS	4	V60	Not equal Character-string	*
NEGW	1	NEG.W xx		*
NEGL	1	NEG.L xx		*
NEQF	1	Macro	Konstruktion mit CMP.L + versch. ff	*
NEQW	1	Macro	Konstruktion mit CMP.W + versch. ff	*
NEQX	1	Macro	Konstruktion mit CMP.W + versch. ff	*

```
NOT     1   NOT.W        Native NOT Word                          *
NOTW    1   NOT.W        Native NOT Word                          *
NOTL    1   NOT.L        Native NOT Long                          *
ODAT    1   V91          Open Dation (no operation)               *
OPN3    3   V48 **       OPEN 3 Param. (BY IDF(...))              *
ORW     1   OR.W         Native OR                                *
ORL     1   OR.L         Native OR long                           *
PAGE    1   V32          PAGE-(E/A)Format                         *
PHDR    1   Macro        Vorderer Teil des PRODEC-Formates        *
PNTR    1   DC.W ...          Only Extension Word                 *
POWD    1   V86          Power Float(55)                          *
POWF    1   V85          Power Float(23)                          *
POWW    1   V83          Power Fixed(15)                          *
POWX    1   V84          Power Fixed(31)                          *
PREV    1   V88          Prevent (given Task)                     *
PROC    1   V18          Call procedure                           *
PUT     3   V27          Eröffnung einer Liste für PUT            *
QSHC    1   MOVEQ ...,D2      Quick shiftcount                    *
RBRK    -   V42          Right Bracket in (E/A)Format             *
READ    3   V108         READ-Instruction                         *
REQU    -   TRAP =6           Request SEMA (adr. by A1)           *
RELA    -   TRAP =7           Release SEMA (adr. by A1)           *
RETN    -   Macro             In PEARL80: Trap =12                *
RETN    -   Macro             In PEARL90: PROCEX-Format           *
RFEN    -   V112         Remote-(E/A)Format end                   *
RLFO    1   V34          Remote-Left-Bracket in (E/A)Format       *
ROLL    -   ROL.L D2,D0                                           *
ROUN    -   JSR -80(A6)       ROUND-function                      *
(ROUN)  -   V143         Nicht für Neuentwicklung (ROUND)         *
RWND    1   V49 **       REWIND Dation and open                   *
SAVP    2   V57 **       Save a position in a file                *
SEEK    2   V56 **       Seek a position in a file                *
SHFL    2   V21          Shift 17 to 32 BIT Object                *
SHFS    2   V20          Shift 1 TO 16 BIT Object                 *
SIGN    1   Macro        LEA obj,A1 + JSR -84(A6) SIGN-function   *
(SIGN)  1   V144         Nicht für Neuentwicklung (SIGN)          *
SKFO    1   V47          SKIP-(E/A)Format                         *
SSHC    -   MOVE D0,D2        Store Shiftcount in D2              *
STAD    1   V136         Nicht für Neuentwicklung (St. accu 8 by) *
STAL    1   MOVE.L D0,xx(A5)                                      *
STSC    -   V100         Start schedule-definition                *
SUBD    1   Macro        LEA obj,A1 + JSR -24(A6)                 *
```

(SUBD)	1	V5	Nicht für Neuentwicklung (Sub Float 55)	*
SUBF	1	Macro	LEA obj,A1 + JSR -8(A6)	*
(SUBF)	1	V4	Nicht für Neuentwicklung (Sub Float 23)	*
SUBW	1	SUB.W	Native Subtract Word	*
SUBX	1	SUB.L	Native Subtract Long	*
SUSE	1	V64	Suspend external (given Task)	*
SUSP	-	TRAP $A028	Suspend self	*
SU20	-	SUB D2,D0	(In FOR ... REPEAT)	*
SWAP	-	SWAP D0		*
SWLT	-	Macro	SLT D0 + EXT D0	*
SYNC	1	V50 **	Synchronize Dation	*
TACT	2	V97	(Timed) ... activate with priority	*
TCON	1	V94	(Timed) ... Continue	*
TERM	1	V87	Terminate (given Task)	*
TFOR	2	V127	T-(E/A)Format	*
TRES	-	V99	(Timed) ... Resume (own Task)	*
TSTW	-	TST.W D0		*
TSTL	-	TST.L D0		*
UNIL	1	V104	UNTIL (clock) schedule param. set	*
VARD	1	V13	Param. XFER: Variable Float 55	*
VARF	1	V12	Param. XFER: Variable Float 23	*
VARW	1	V10	Param. XFER: Variable Fixed 15	*
VARX	1	V11	Param. XFER: Variable Fixed 31	*
WACT	2	V96	(When) ... activate with priority	*
WCON	1	V93	(When) ... Continue	*
WHEV	1	V106	WHEN (event) schedule param. set	*
WRES	-	V98	(When) ... Resume (own Task)	*
WRIT	3	V107	WRITE-Instruction	*
XFOR	1	V33	X-(E/A)Format	*

8.8 E/A in Assemblersprache

Vor dem Studium der Codierung eigener E/A-Treiber sollte man sich mit der typischen Maschinencode-Sequenz einer Ein- oder Ausgabe vertraut machen. Dazu betrachten wir den Fall einer E/A über den ACIA/SCC mit der Warteschlangennummer 2. Weil sich die Displacements zwischen der 68k- und der PowerPC-Familie unterscheiden, wird dringend empfohlen, die Datei „`COMEQU`“ per `.INCLUDE` einzubinden. Die nachfolgend angegebenen `EQU`s sind nur zur Information angegeben. Bitte lesen Sie dazu auch auf Seite 559 die Beschreibung des Communication Elements (CE) nach.

```
*       Define Systemtraps:
FETCE   OPD    $4E48   Fetch communication-element (CE) Systrap
IOWA    OPD    $A00A   I/O wait function
RELCE   OPD    $4E49   Release the CE
XIO     OPD    $4E4A   Xfer communication-element for I/O
*       Displacements (PowerPC rechts, sofern abweichend):
BUADR   EQU    $20 $24 4 byte long buffer-address (from FETCE)
FNAME   EQU    $2C $30 File-name
LDNIO   EQU    $27 $2B Logical dation number (=Queue-number)
MODE    EQU    $28 $2C Mode-Byte of Communication-Element
DRIVE   EQU    $2A $2E Driver-Number or ACIA-Mode
RECLEN  EQU    $24 $28 Record length (16 bit)
STATIO  EQU    $26 $2A Statusbyte of communication-element
*       Symbolic masks and bit-positions:
MODMOU  EQU    $40     Mode-mask  for  Output
MODMCR  EQU    $20     Mode-mask  'carriagereturn ends record'
MODMWA  EQU    $80     Mode-mask 'wait for completion'
MODM..                 Other masks when used
STABRE  EQU    1       'Verschrottungsbit'
```

Fall A

Es soll eine Ausgabe gestartet werden und während des Transfers noch etwas anderes gemacht werden. Das CE soll anschließend für einen weiteren E/A-Vorgang erneut benutzt werden. Textlänge sei 50 Characters. Wir codieren:

```
  ...    ...         Arbitrary code before write sequence
 _MOVEQ  =50,D1      Communication-elem. with 50 char. info-len.
  FETCE              Get space from RTOS-UH, A1 is loaded
  MOVE.W =MODMOU*$100,MODE(A1) Xfer-mode: out,no wait,by cnt.
  MOVE.B =2,LDNIO(A1)   Queuenumber = 2  (Dation=ACIA2)
 _MOVE   =0,DRIVE(A1)   Ax (not Bx or Cx)
```

```
  MOVEA.L BUADR(A1),A2  pointer 1st character in buffer
  ...
  MOVE.B ...,(A2)+      put info into communication elem.
  ...
 _MOVE   =50,RECLEN(A1)  all 50 characters to write
  XIO                Make the output
  ...                other activities while output is running
  ...                A1 must be saved and reloaded !!
  IOWA               Wait for completion of I/O with A1-com.el
  ...                from here on:  A1-CE may be used again
  RELCE              Release the CE in A1. A1 is invalid now
```

Fall B

Über B2 soll ein Stück Text gelesen werden und dabei noch Programmaktivität während des Lesens stattfinden. Außerdem soll kein Echo erzeugt werden, der Zugang zum Bedieninterface ist zu verriegeln, damit die Zeichen $01 etc. gelesen werden können. Maximale Textlänge sei 40 und beim ersten auftretenden Carriagereturn soll der Transfer beendet werden (Das Zeichen CR steht dann als letztes im Puffer). Wir codieren zusätzlich zum Universalvorspann:

```
MODMCR  EQU    $20     End Transfer with 1st carriagereturn.
MODMNE  EQU    $02     'No echo' (Only ACIA/SCC)
MODMSC  EQU    $04     Suppress Command (only ACIA/SCC)
     ...
    _MOVEQ  =40,D1  D1.L     = length of info in CE
     FETCE                   load A1 by RTOS-UH
     MOVE.B =2,LDNIO(A1)     Queue-Number is 2
     MOVE.W =(MODMCR+MODMSC+MODMNE)*$100,MODE(A1) Xfer mode setup
     MOVE   =40,RECLEN(A1)   max. number of char's
    _MOVE   =2,DRIVE(A1)     'B' (buffered) instead of 'A'
     XIO            Make the Xfer
     ...            Arbitary-code, command-if remains blocked.

     IOWA           Wait until record ready (CR or 40 char's).

*    Die Daten liegen nun bereit. In RECLEN(A1) ist die Anzahl
*    Bytes zu finden. Auf BUADR(A1) die 4 bytes lange  Adresse
*    des ersten Zeichens. Nach Auswertung der Daten:

     RELCE          Release CE. A1 invalid from now on!
```

```
*    Anmerkungen zum hier demonstrierten Beispiel:
*    Damit das Bedieninterface wieder 'befreit' wird, muss ein
*    ACIA2-XIO   ohne  den 'suppress command-if'- Mode  folgen
*    oder der Terminalbediener drueckt die 'break'-taste.
```

Fall C

„Ausgeben und vergessen“ eines Konstantenstring. Zusätzlich zum Universalvorspann codieren wir nun:

```
  ...
 _CLR.L  D1      Length of textbuffer = 0
  FETCE          Fetch a CE
  LEA    textadr,A0
  MOVE.L A0,BUADR(A1)   Start adr of text to write
  MOVE.W =MODMOU+MODMCR+$0,MODE(A1) Output until CR
  BSET   =STABRE,STATIO(A1)   'Verschrottungsbit'
  MOVE   =100,RECLEN(A1)      Max. number of char if no CR
  MOVE.B =2,LDNIO(A1)         queue-number
 _CLR    DRIVE(A1)            Set to Ax
  XIO                         A1 no longer valid!

*    Die Ausgabe laeuft, das CE ist fuer diese  Task nun nicht
*    mehr erreichbar, da  nicht  festzustellen ist, ob es noch
*    existiert.
```

8.9 Ergänzung von E/A-Treibern

In der 68k-Version des Betriebssystemes **RTOS–UH** sollten alle Peripheriegeräte, bei denen Wartezeiten einmalig > 0.6 ms bzw. wiederholt > 80 μs anfallen, durch sog. „I/O-Dämonen“ (Treibertasks) betreut werden. Für die schnellen PowerPC-Versionen sind die obigen Zeitrichtwerte erheblich zu verkleinern. Beim E/A-Vorgang wird durch die Usertask mit dem „`XIO`“-Trap ein Eintrag in eine prioritätentengeordnete Warteschlange angelegt. War die Warteschlange vorher leer, so aktiviert `XIO` über eine `LDN`-Tabelle den für diese Warteschlange zuständigen I/O-Dämonen.

Für Peripherie, die dem Prozessor keine verwendbaren Zeitreste läßt, macht diese Konstruktion keinen Sinn. So wird z. B. über den VME- oder PIA-Bus mit `SEND` und `TAKE` direkt und ohne Taskwechsel kommuniziert.

Wenn Sie Ihr System mit weiteren Peripheriegeräten ausbauen und dafür keine der im **RTOS–UH** vorhandenen Treibertasks benutzt werden kann – z. B. neues V24-Port – dann ist das System in der hier angegebenen Weise zu erweitern. Die Aufgabe, die hierbei zu erledigen ist, gliedert sich in drei Phasen:

Phase 1: Festlegung einer „`LDN`“ für das neue Peripheriegerät.
Phase 2: Codierung des I/O–Dämonen.
Phase 3: Codierung des Interruptprozesses.

Phase 1: Festlegung einer `LDN`

a) Das System soll fortan dauerhaft die neue Station enthalten: Der Bootblock oder das ROM muß erweitert werden. Zweckmäßigerweise wird die nächste freie `LDN` belegt. Geht man nach oben darüber hinaus, so wird vom Nukleus beim Autolinking die `LDN`-Lücke für nachladbare I/O-Tasks verwaltungstechnisch aufbereitet. (Tabellenplatz) Die Station muß mit einem üblichen alphanumerischen Namen versehen werden, der die Verwendung durch das Bedieninterface ermöglicht. Dazu ergänzen wir eine „Scheibe“ vom Typ 9. Die Stationseigenschaften werden mit einer Scheibe des Typs 10 vorbesetzt, sie wirkt wie ein ganz zu Anfang ausgeführter „`SD`“-Bedienbefehl. Die notwendigen Informationen sind ab Seite 635 bei der „Scheibenstruktur“ zu finden.

b) Das System soll nur vorübergehend um die neue Datenstation erweitert werden: In diesem Falle müssen Sie eine `LDN` aus einer der vorhandenen Lücken wählen. Die Station ist bei dieser Form der Erweiterung allerdings immer nur als „`LD/x.y/`“ vom Bedieninterface errreichbar. Die Stationseigenschaften können nur mit Hilfe des „`SD`“-Bedienbefehles besetzt werden.

Phase 2: Codierung des I/O-Dämonen

Der Taskname (Name des Dämonen) ist frei wählbar. Allerdings sollte bei einer Boot/ROM-Erweiterung dieser mit „#“ beginnen, um den Dämonen vor dem `UNLOAD` zu schützen. Beim Entladen eines I/O-Dämonen ist in jedem Fall besondere Vorsorge zu treffen, so daß der Eintrag in der „LDN-TID“-Tabelle des Nukleus wieder gelöscht wird.

Damit der von irgendwo exekutierte „`XIO`“-Trap sein Communication-Element (siehe dazu Seite 559) weiterleiten kann, benötigt er eine Zuordnungstabelle, die die `LDN` in den Task-Identifier (`TID`) umwandeln kann. Beim Autolinking baut der Nukleus diese Tabelle auf (siehe Scheibe 1). Bei den nachmontierten Tasks müssen Sie (oder jemand anders) diesen Eintrag selbst erledigen. Nicht belegte Tabellenplätze sind durch den Eintrag einer (4-Byte) 0 markiert.

```
*      Task-head mit Namen,Prioritaet etc.            *
* Oder: Scheibencode der Scheibe no. 1                 *
*      Hohe Prioritaet wird empfohlen, z.B. -1 oder    *
*      dynamische Prioritaet durch Angabe von 0        *
*                                                      *
*        System-traps needed here:                     *
DPC      OPD     $4E43      Dispatcher-caller         *
OFF      OPD     $4E4F      All interrupts 'off'      *
RELCE    OPD     $4E49      Release Comm.element      *
TOQ      OPD     $4E4D      Take of queue             *
TERMI    OPD     $4E41      Terminate (self)          *
*
*        Displacements (PowerPC rechts) use COMEQU !! *
EXCORG   EQU     0    $4000 Exception origin          *
BLOCK    EQU     $24  $22   Block-byte of a Task      *
BLKBSU   EQU     4    4     Suspend-bit-no. in BLOCK  *
TID      EQU     $802 $5000 Actual running Task ident.*
SIOLDT   EQU     $852 $50B0 Start i/o-LDN to TID table*
IDP1     EQU     $832 $508C Interr. data buffer 1     *
...      ...
IDP7     EQU     $84A $50A4 Interr. data buffer 7     *
*
*        Interrupt-buffer                              *
* a)  Platz ueber Scheiben 2 ... 8                     *
*     Interrupt-Vector anschliessen ueber Scheiben-
*     nummer 14                                        *
* b)  Platz im RAM freihalten:                         *
```

```
IRLINK   DC      0,0,0,0,...0  as used by ir-process  *
*
START:                        ;                        *
  nur b) MOVEA.L SIOLDT,A1  Tab-pointer LDN-TID       *
  nur b) MOVE.L  TID,ldn*4(A1)  Montieren der I/O-Task*

TAKE:    TOQ                Take of queue            *
         BRA.B   EXIT (muss .B sein!)Wenn Schlange leer
         BRA.B   DOIT       CE aus Schlange gefunden  *

EXIT:    TERMI               Ende weil Schlange leer  *

DOIT:
  nur a) MOVEA.L IDP1...IDP7,A0   Buffer-pointer IDPx *
  nur a) LEA     OFFS(A0),A0  benutzten Ber. auslassen*
  nur b) LEA     IRLINK,A0  access to ir-link-block   *
         MOVE.L  A1,(A0)+        Comm.element for ir  *
         MOVEA.L TID,A2     Link to Task-identifier   *
         MOVE.L  A2,(A0)+   fuer ir-process           *
         CLR     (A0)+      Character-index reset     *
         ....               hier moeglicherweise andere*
         ....                Interruptunkritische Op.  *
*                                                     *
*        Beginn der unteilbaren Sequenz:              *
*
         OFF                Alle Unterbrechungen sperr*
        ....    (coupler)  Interrupt-process mit Para-
        ....               metern  versorgen, Hardware
        ....       ''      'scharf' machen, IR-PC     *
        ....               versorgen etc.             *
  nur b) MOVE.L  =IRxy,IVEC+EXCORG  Vector  anschl.   *
       ....               !!  privilegierter Mode ! !*
       _BSET   =BLKBSU,BLOCK(A2) Task suspendieren
        DPC                Ende Off-sequ. Disp. start*
*                                                     *
*        Der Daemon schlaeft nun bis der IR-Process   *
*        ihn wieder freischaltet.                     *
*                                                     *
         RELCE              Freigabe des Comm.elem.   *
         BRA     TAKE       naechstes El. aus Schlange*
```

Phase 3: Codierung des Interruptprozesses

Typische Aufgabe des Interruptprozesses wird die Behandlung des Dateninterrupts oder die Beendigung der Blockierphase des zugehörigen Dämonen aus irgendwelchen Gründen sein. Nach Abschluss oder Abbruch der Datenübertragung muß der suspendierte Dämon wieder freigegeben werden. Damit auf Interruptebene der Zugriff auf dessen Blockbyte möglich ist, muß bereits auf Taskebene vor der Selbstsuspendierung der `TID` des Dämonen im Interruptdatenblock abgelegt werden.

RTOS–UH besitzt im Gegensatz zu fast allen anderen Betriebssystemen einen speziellen Reparaturmechanismus für unerwartete Fehler innerhalb von Interruptantwortroutinen. Um solche Fehler wie `wrong opcode`, `wrong address` etc. auf eine interruptspezifische Art in einen geordneten Rückfall ableiten zu können, gibt es die Zelle „`IID`“ (`$7FE` bei 68k, `$5004` bei PowerPC) = „Interruptidentifier“. Der Nukleus bestimmt im Fehlerfalle mit Hilfe von `IID`, die aktuelle Ansprungadresse für den Rückfallmechanismus. Die genaue Kodierung von `IID` ist zwischen der 68k- und der PowerPC-Version dabei unterschiedlich, weil die Prozessoren sich im Supervisormode zu stark unterscheiden – so besitzt der PowerPC keinerlei Äquivalent zum Vectorbaseregister des 68k. Aus diesem Grund werden für die Versorgung von `IID` in der Datei `SUPERVIS.FOR` Formate angeboten, die sich automatisch an die Zielhardware (68k oder PowerPC) anpassen.

In jedem Fall muß die Zelle `IID` zunächst gerettet werden, denn der Interruptprozeß könnte ja einen anderen unterbrochen haben, dessen Vektorlink am Ende wiederhergestellt werden muß. Entsprechend muß am Ausgang der Interruptroutine die Zelle `IID` wieder auf den alten Wert zurückgestellt werden. (Wenn man mit Hilfe der Shell auf `IID` nachsieht, wird man dort stets und alle Zeit den Wert 0 finden, der angibt, daß man sich nicht in einer Interruptroutine befindet.)

Man beachte, daß beim PowerPC ein besonderer „Interruptpreprozessor“ die Register `r31`, `ccr` und `lr` zunächst automatisch freistellt. Danach wird `lr` allerdings mit der Rückehradresse in das Interruptgate neu geladen und muß notfalls gerettet werden. Weil der Transferassembler für die Umsetzung des T–Codes auf den PowerPC neben den freien Registern `r31` und `ccr` im Extremfall noch 9 weitere Hilfsregister benötigt, werden diese vom Format `IRENTC` zu Beginn der Interruptroutine neben `IID` ebenfalls gerettet. Mit dem Format `IREXTC` werden sie wieder zurückgeladen. Will man „native“ PowerPC kodieren, so sind die Formate `IRENPP` und `IREXPP` statt dessen zu verwenden. Letztere retten die Register `r25` ... `r30`, `ctr`, `xer` und `lr` nicht, sind ansonsten aber funktionsidentisch.

Die „Malfunction“-Routine muß dafür sorgen, daß z. B. alle Register restauriert

werden und ggf. der Interruptverursacher (Coupler etc.) in einen normierten Zustand versetzt wird. Als Minimum ist die Ableitung der Malfunction auf den regulären Interruptausgang anzusehen.

Warnung 1

Die Benutzung von Traps auf Interruptebene ist generell verboten – obwohl es mit einigen möglich wäre. Auch die Verwendung von „BSR“–Befehlen ist wegen der damit verbundenen Paralysierung des „Rückfallmechanismus“ gefährlich!! BSR/RTS kann durch LEA ...,Ax und JMP (Ax), eine obendrein schnellere Lösung, ersetzt werden, oder aber man rettet nach dem IID- und Registersave das System-A7 in ein eigenes Register, um es bei Malfunction als erstes wieder auf den korrekten Wert zu bringen. Jedes Verlassen der Interruptroutine darf nur unter Einschaltung des Prozeßumschalters erfolgen, damit während der IR-Prozedur aufgelaufene Taskzustandsänderungen nicht „verschlafen“ werden: Verwenden Sie also in der 68k-Welt nie den RTE-Befehl direkt!

Warnung 2

Die Interruptantwortroutine liegt typischerweise im gleichen Programmtext wie die Grundebenentask, bedenken Sie aber, daß zum Zeitpunkt des Interrupteintrittes irgendein völlig fremder Prozeß die Prozessorregister etc. besitzt. Sie müssen größte Aufmerksamkeit darauf verwenden, daß der unterbrochene Prozess korrekt fortgesetzt werden kann! Man legt hier sonst eine extrem gefährliche Zeitbombe in das System!

Wir studieren nun den schematischen Aufbau einer T–kodierten Interrupt-Antwortroutine:

```
         .INCLUDE   ../COMEQU        EQUs
         .INCLUDE   ../SUPERVIS.FOR  Supervisorformate *
IVEC     EQU        $200       Assumed IR-Vector adr   *
*
*.....   Here is the Interrupt entry point ...........*
IRxy     IRENTC    malfxy,IVEC Malfunc Anschluss etc. *
         MOVEM.L   D0...,-(A7)    Save Registers used *
nur b)   LEA       IRLINK,A0 Parameter-feld anschl.   *
nur a)   MOVEA.L   IDP1...IDP7,A0     IDPx laden      *
nur a)   LEA       OFFS(A0),A0    fremden Bereich skip*
         .......              Von der Task-Grundebene  *
         .......              angelegte Daten, z.B. die*
         .......              Adresse des CE, sind ueber
```

```
          .......              xx(A0) erreichbar        *
          MOVEA.L   (A0),A1      access to comm.elem.   *
          MOVE.B    ...,...    Daten ueber Coupler      *
          .......                                       *
          CMP       ...,...    Test ob Transfer fertig  *
          BNE       Exitxy     Sprung wenn nicht fertig *
*
*......   Transfer fertig: Daemon wieder freigeben ... *
          MOVEA.L   4(A0),A0     access Task-id         *
          _BCLR     =BLKBSU,BLOCK(A0) continue task     *
          DPCALL               Alert dispatcher         *
Exitxy    MOVEM.L  (A7)+,D0...  Register rueckladen     *
          IREXTC               korrekter Ausgang        *
*                                                       *
malfxy    IRMALF   IVEC        bei 68k Leeroperation    *
          ....                 Controller normieren     *
          BRA      Exitxy      Exit by Disp. test       *
```

Phase 3: Andere Konstruktion ohne Interrupts

Liefert der Coupler keine Interrupts, so kann auf der Ebene der Task selbst das Communicationelement bearbeitet werden. Um Verluste durch Abfrageschleifen zu vermeiden, bediene man sich einer „durchlöcherten“ Schleife, etwa, indem die Task alle 4 msec nachschaut und sich bei fehlender Bereitschaft des Couplers erneut selbst suspendiert für 4 msec. Bei hohen Datenraten geht es natürlich auch ganz ohne Suspendierung der I/O-Task.

Beispiel für einen E/A-Treiber

Es handelt sich um ein einfaches Programm, welches einen normalen CE-Transfer zu einem ACIA-Baustein gestattet. Zur Demonstration wurde auch die typische ausgereizte Sequenz zur Erkennung von `CR`, `LF` oder `EOT` angegeben.

Das Maschinenprogramm braucht nur noch zugeladen zu werden. Die Task ist einmal zu aktivieren, damit der Tabelleneintrag erfolgen kann. (Wegen leerer Schlange keine Aktion). Anschließend ist über „`SD /LD/5/ xx`“ das Port zu parametrieren. Bevor die Task mit `UNLOAD` entfernt wird, muß entweder über eine Hilfstask oder mit Hilfe des `SM`-Befehles (Vorsicht! sorgfältig rechnen) wieder eine Null an Stelle des `TID` eingesetzt werden. (4 Byte Null)

```
*************************************************
*      Demonstrationsprogramm   'eigene  E/A'    *
*      HIER: Nachmontierte Version               *
*                                                *
*      Taskname: Queue5, LDN=5                   *
LDN    EQU    5              For assembler       *
*-----------------------------------------------*
      .INCLUDE  .../COMEQU.NOL   passende EQUs   *
      .INCLUDE  .../GENERAL.FOR  Task-DCB etc    *
      .INCLUDE  .../SUPERVIS.FOR fuer IR-Prozess *
*-----------------------------------------------*
*..... TASK-HEAD for RTOS-UH:                    *
       DC.B   0,0,0,0,0,0,0,0,0,1,'Queue5'       *
       TSKDCB 0,WSPMIN,START prio=0=dynamic      *
*-----------------------------------------------*
*      Coupler- and interrupt- addresses:        *
*      *** Depending upon actual hardware ***    *
*          ----------------------------          *
ACST   EQU    $50041          Statusreg ACIA      *
ACDT   EQU    $50043          Datareg ACIA        *
IVEC   EQU    $210            Interrupt-link      *
*-----------------------------------------------*
*      System traps:                             *
DPC    OPD    $4E43           Dispatcher-caller   *
OFF    OPD    $4E4F           All interrupts off  *
RELCE  OPD    $4E49           Release comm.elem.  *
TOQ    OPD    $4E4D           Take off queue      *
TERMI  OPD    $4E41           Terminate self      *
*-----------------------------------------------*
*..... Link-cells daemon<->IR-process ...........*
IRLINK DC.L   0               Actual Text-address *
       DC.L   0               Task-ID of this Task*
       DC     0               Length-control-word *
*-----------------------------------------------*
*      T A S K  -  C O D E:                      *
*                                                *
TERMEX TERMI                  Used from below     *
START  MOVEA.L SIOLDT,A0      Table-address       *
      _MOVE.L TID,LDN*4(A0)   Nachmontage LDN     *
       TOQ                    Inspect the queue   *
       BRA.B  TERMEX          B: queue empty      *
*..... Queue is not empty ......................*
```

```
        LEA     IRLINK,A0       For rapid access    *
       _MOVE.L BUADR(A1),(A0)+  Start adr of Text *
        MOVEA.L  TID,A2         Task Id for access  *
        MOVE.L A2,(A0)+         save for ir-process *
*                                                   *
*....   Determine number of chars to transmit       *
*                                                   *
        MOVE    RECLEN(A1),D2   Assumed length      *
        MOVEQ   =MODMCR+MODMLF+MODMEO,D6  Testmask *
        AND.B   MODE(A1),D6     Quick check         *
        BEQ.S   A06             b: no mode specified*
        MOVEA.L BUADR(A1),A3    Text-org            *
        CLR     D3              Reset Record-length *
A02     CMP     D2,D3           Reclength test      *
        BGE.S   A06             b:all done          *
        ADDQ    =1,D3           Move counter        *
        MOVE.B  (A3)+,D0        Inspect the byte    *
        MOVEQ   =MODMCR,D7      Test-mask           *
        SUB.B   =$0D,D0         Test for Carr. rtn  *
        BGT.S   A02             b: not cr,lf or eot *
        BEQ.S   A04             b:is cr             *
        MOVEQ   =MODMLF,D7      Testmask            *
        ADDQ.B  =$0D-$0A,D0     Test for LF         *
        BEQ.S   A04             b:is LF             *
        ADDQ.B  =$0A-$04,D0     Test for EOT        *
        BNE.S   A02             b:not eot           *
        MOVEQ   =MODMEO,D7      Testmask            *
A04     AND.B   D6,D7           Mode-match?         *
        BEQ.S   A02             b:no match          *
        MOVE    D3,D2           result length       *
A06     MOVE    D2,(A0)+        Store length        *

        LEA     IRENT,A3        Ir-entry address    *
*
        OFF                     Disable interrupts  *
        MOVE.B  =$35,ACST       New coupler status  *
        MOVE.L  A3,IVEC+EXCORG  Ir-vector connection*
       _BSET    =BLKBSU,BLOCK(A2)  suspend the task*
        DPC                     Call dispatcher     *
*       Now the task is suspended for last interr.*
        RELCE                   Release Caller      *
        BRA     START           Repeat queue-op.    *
```

```
*-----------------------------------------------*
*                                               *
*      I N T E R R U P T - P R O C E S S        *
*                                               *
*                                               *
IRENT  IRENTC IRMAL,IVEC     IR-Header          *
       MOVEM.L A0/A1,-(A7)     Save registers    *
       LEA    IRLINK,A0      Link to parameters *
       SUBQ   =1,8(A0)       Counter control    *
       BMI.S  IRCOD4         b:end of transm.   *
       MOVEA.L (A0),A1       Buffer-address     *
       MOVE.B  (A1)+,ACDT    Send data to periph.*
      _MOVE.L  A1,(A0)       Restore new pointer *
IRCOD0 MOVEM.L  (A7)+,A0/A1        reload reg's  *
IRCOD1 IREXTC                 IR-Exit            *
*                                               *
*      End of transmission                      *
IRCOD4 MOVE.B =$15,ACST       Switch coupler off *
       MOVEA.L 4(A0),A0       Access task-id     *
      _BCLR   =BLKBSU,BLOCK(A0)  Continue        *
       DPCALL                 Flag dispatcher-call*
       BRA    IRCOD0          Exit               *
*                                               *
*      Malfunction recovery-exit                *
IRMAL  IRMALF IVEC            Whatever is necessar*
       BRA.S  IRCOD4          Make daemon runnable*
*                                               *
       END
```

8.10 Exception-Handler

8.10.1 Einführung

Programmierfehler können nicht gewollte asynchrone Traps auslösen, wie z.B. den Bus-Error-Trap bei versuchtem Zugriff auf nicht vorhandene Speicheradressen. **RTOS–UH** kann diese Fehlermeldungen über den Error-Dämon ausgeben. Der Error-Dämon ist gewöhnlich die Task mit der höchsten Priorität im System. Dadurch erfolgt die Ausgabe immer mit höchster Priorität.

Anstelle des Error-Dämons als zentrale Task für die Fehlerbehandlung kann man jeder Task taskindividuell eine Prozedur, den sogenannten „Exception-Handler“ zur Verfügung stellen. Dieser behandelt dann alle Ausnahmen einschließlich der vom Anwender selbst programmierten Aufrufe des „Error-Traps“ (siehe Seite 475).

Eine Restaurierung aller Register in den Zustand vor der Ausnahme ist dabei wegen der Fehleranalyse und einer eventuellen Fortsetzung der Task wünschenswert. Prinzipiell kann der Anwender einen Exception-Handler selbst schreiben und anschließen (siehe Unterabschnitt „Interna“). Wegen der Restaurierung der Prozessorregister sowie wegen der unterschiedlichen Behandlung von Ausnahmen durch die verschiedenen Prozessoren stellt **RTOS–UH** den sogenannten „**RTOS–UH**-internen Exception-Handler“ zur Verfügung. Er besitzt gegenüber dem Error-Dämon folgende Vorteile:

- Die Fehlermeldung besitzt die Priorität der aufrufenden Task. Dadurch werden höherpriorisierte Tasks nicht behindert.
- Löst eine Task mehrere Ausnahmen so schnell hintereinander aus, daß die vorherigen Fehlermeldungen vom Error-Dämon noch nicht ausgegeben sind, erfolgt beim Exception-Handler im Gegensatz zum Error-Dämon eine korrekte Ausgabe der Zeilennummer nicht nur bei der ersten Meldung.
- Läuft nach einer Ausnahmebehandlung eine Task weiter, wie es z. B. häufig bei Floating-Point-Exceptions der Fall ist, werden zwei Taskwechsel gespart.
- Der Anwendungsprogrammierer bekommt die Möglichkeit, verschiedene Ausnahmen selbst zu bearbeiten.

Der Error-Dämon hat allerdings auch Vorteile. Zum einen ist es nicht immer erwünscht, daß die Priorität der Fehlermeldung die der auslösenden Task ist. Zum anderen sendet die auslösende Task bei einem Exception-Handler die Fehlermeldung selbst an den I/O-Dämon, was mit einem Warten auf das Ende der Ausgabe verbunden ist.

8.10.2 Anschluß des Exception-Handlers

Bevor auf die innere Struktur eingegangen wird, soll zunächst der Anschluß des **RTOS–UH**-internen Exception-Handlers beschrieben werden. Das Shell-Subroutine-Package stellt dafür die Routine `EXLKL` bereit. Im einfachsten Fall ist sie wie folgt zu parametrieren:

`D7.W` `0`

`A1.L` Zeiger auf den Arbeitsbereich des Exception-Handlers, der bei diesem Aufruf dem Exception-Handler fortan für seine spätere Tätigkeit zur Verfügung gestellt wird. Bei diesem sogenannten „Exception-Frame“ handelt es sich typischerweise um einen reservierten Bereich aus Taskworkspace oder Taskkkopf. Die minimale Größe ist durch das in COMEQU definierte Label `SGEFFS` festgelegt (68xxx: 82Bytes, PowerPC: 132 Bytes).

`A2.L` Zeiger auf den Error-Pfad. Ausschnitt aus einem CE von `STATIO` bis zum Ende des Names.

`A6.L` TID der Task, die einen Exception-Handler bekommen soll. Meistens montieren sich Tasks selber einen, aber bei Debuggern o. ä. kann es sich auch schon eimal um eine andere Tasks handeln.

Beispiel: Eine Task möchte sich selber einen Exception-Handler montieren und will Fehlermeldungen immer auf den „Permanent-Error“ Pfad von User 1 ausgeben.

```
*-------------------------------------------------------*
       .include COMEQU.NOL                              *
*-------------------------------------------------------*
EXLKL  EQU    124                                       *
*-------------------------------------------------------*
*  Taskworkspace-Layout                                 *
*  PMBUF: erste freie Speicherstelle im Taskworkspace   *
FRAME  EQU    PMBUF         Platz fuer Exception-Frame   *
FFREE  EQU    FRAME+SGEFFS  Ab hier eigene Variablen     *
*-------------------------------------------------------*
       ....                  u.a. Stack initialisieren   *
*                                                       *
      _MOVEQ  =0,D7          einfachster Anschluss       *
       MOVEA.L TID,A6        Eigener Taskkopf-Zeiger     *
       LEA    FRAME.T,A1     Zeiger auf den Exception-Frame *
*     Standard-Error-Pfad holen                         *
```

```
       MOVEA.L UITIDP,A2     USER TO TID-TABLE              *
       MOVEA.L (A2),A2       Taskkopf User 1                *
       ADDA   PTHLEN,A2      A2:Shell-Environment-Zeiger    *
       MOVEA.L STDELP(A2),A2 Fehler-Pfad                    *
       MOVEA.L CIADR,A0      Zeiger auf SSRP-Tabelle        *
*     A4 muss auf eigenen Taskworkspace zeigen!             *
       JSR    EXLKL(A0)      Exeption Handler anschliessen  *
                                                           *
       ...                                                 *
*-----------------------------------------------------------*
```

Nach Ausführung dieser Sequenz ist der Exeption-Handler montiert. Er übernimmt anstelle des Error-Dämons nun die Fehlerbehandlungen. Auf den ersten Blick verhält sich **RTOS–UH** genauso wie vorher. Die oben angegebenen Vorteile sind nun jedoch vorhanden.

8.10.3 Selbstverarbeitete Ausnahmebehandlungen

Der Vorteil des im vorherigen Unterabschnitts vorgestellten Exception-Handlers liegt in der Erweiterbarkeit durch ein Anwendungsprogramm. Dieses kann wahlweise einige oder alle Ausnahmen selbst verarbeiten. Als Schlüssel für die Zuständigkeit dient das Error-Codewort, welches der Exception-Handler an Hand der Tabelle auf Seite 467 auswertet.

Beim Aufruf der Routine `EXLKL` können folgende Fälle unterschieden werden:

`D7.W 0`	Keine Erweiterung durch Nutzer.
`D7.W 1`	Eine gemeinsame Prozedur für alle selbstbehandelten Ausnahmen. In D6.L steht ein Zeiger auf eine Liste mit Error-Code-Wörten, die mit 0 abgeschlossen ist. Die anzuspringende Adresse, falls das Code-Wort in der Liste enthalten ist, steht in A3.L.
`D7.W 2`	Eine individuelle Prozedur für jede Ausnahme. In D6.L steht ein Zeiger auf eine Liste. Ein Listeneintrag sieht wie folgt aus: 2 Bytes Error-Code Wort, gefolgt von einem Langwort, das die für dieses Code-Wort zugehörige Adresse enthält. Das Codewort `$FFFF` steht für ein beliebiges Code-Wort, das Codewort `$0000` schließt die Liste ab.
`D7.W 3`	Eine gemeinsame Prozedur für alle Ausnahmen. In A3.L steht die anzuspringende Adresse.

Weiterhin können noch folgende funktionelle Bits in `D7` gesetzt werden: `SGCBFM+8` bewirkt, daß vor Ansprung der externen Routine die Fehlermeldung ausgeben wird. Ist Bit `SGCBBR+8` gesetzt, wird auch bei Breakpoints der Exception-Handler aufgerufen (Sonst macht der Error-Dämon die Meldung —

unabhängig davon, ob ein Handler angeschlossen ist oder nicht.) Um das Trace-Bit des Prozessors zu löschen, während der Exception-Handler selbst läuft, kann das Bit `SGCBNT+8` gesetzt werden. Beim Verlassen des Handlers mit der im folgenden beschriebenen Prozedur `EXRTN` setzt diese das Trace-Bit in den Zustand vor dem Auslösen der Exception.

Um seine eigenen Prozeduren korrekt zu beenden, stellt das SSRP die Routine `EXRTN` zur Verfügung, die verschiedene Arbeitsregister in den Zustand vor der Ausnahmebehandlung zurückversetzt (siehe Beispiel). Sie ist wie folgt zu parametrieren:

`A2.L`	Zeiger auf den Exception-Frame
`D5.B`	MI: Task wird immer suspendiert.
`D5.B`	EQ: Task wird nie suspendiert.
`D5.B`	GT: Task wird suspendiert, falls es vom Error-Codewort vorgesehen ist.
`D5.W`	MI: Fehlermeldung wird nachgeholt.
`D5.W`	PL: Normaler Ausstieg.

Beispiel: Das obige Programm ist so zu erweitern, daß bei einem Bus-Error, falls er bei einem bestimmten PC ausgelöst wird, die auslösende Task fortgesetzt wird. Dadurch laßt sich überprüfen, ob ein bestimmter Baustein oder eine bestimmte Einschubkarte vorhanden ist. Ideal wäre eine Unterdrückung der Fehlermeldung im Falle des Fortsetzens.

```
*-------------------------------------------------------------*
       .include COMEQU.NOL                                   *
*-------------------------------------------------------------*
EXLKL  EQU    124                                            *
EXRTN  EQU    128                                            *
*-------------------------------------------------------------*
*  Taskworkspace-Layout                                       *
*  PMBUF: erste freie Speicherstelle im Taskworkspace         *
FRAME  EQU    PMBUF          Platz fuer Exception-Frame      *
CODLST EQU    FRAME+SGEFFS   Code-Wort-Liste                 *
FFREE  EQU    CODLST+4       Ab hier eigene Variablen        *
*-------------------------------------------------------------*
       ....                   u.a. Stack initialisieren      *
                                                             *
       MOVEQ  =1,D7           Liste mit Sammel-PC            *
      _MOVE.L =$80150000,CODLST.T  Bus-Error & Listenende    *
       LEA    CODLST.T,A6     Zeiger auf Liste               *
      _MOVE.L A6,D6           Fuer SSRP-Ansprung             *
       LEA    BUSEPC,A3       Bus-Error PC                   *
```

```
       MOVEA.L TID,A6        Eigener Taskkopf-Zeiger        *
       LEA    FRAME.T,A1     Platz fuer Arbeitsbereich      *
*     Standard-Error-Pfad holen                             *
       MOVEA.L UITIDP,A2     USER TO TID-TABLE              *
       MOVEA.L (A2),A2       Taskkopf User 1                *
       ADDA   PTHLEN,A2      A2:Shell-Environment-Zeiger    *
       MOVEA.L STDELP(A2),A2 Fehler-Pfad                    *
       MOVEA.L CIADR,A0      Zeiger auf SSRP-Tabelle        *
*     A4 muss auf eigenen Taskworkspace zeigen!             *
       JSR    EXLKL(A0)      Exeption Handler anschliessen  *
       ...                                                  *
      _MOVEQ  =0,D0          Annahme: kein Bus-Error        *
NOP_PC MOVEA.L (A3),A3       Falls hier Bus-Error,fortsetzen*
       TST    D0             Test: Bus-Error??              *
       BEQ    ???            B: War keiner                  *
       ...                                                  *
*-----------------------------------------------------------*
*Eigene Ausnahmebehandlung fuer Bus-Error                   *
*Folgende Register stehen zur eigenen Verfuegung:           *
*68xxx:   SR                                                *
*PowerPC: r25...r31, xer,lr,cr                              *
*Beide:   D1,D5,D6,D7,A1,A2,A3,A5,A7                        *
*                                                           *
BUSEPC LEA    NOP_PC,A1      PC fuer Fortsetzen holen       *
       MOVEA.L D7,A2         Zeiger auf Exception-Frame     *
       MOVE.L SGOLPC(A2),D7  Ausloesender PC                *
       MOVEQ  =1,D5          Annahme: anderer PC            *
       CMP.L  A1,D7          Test: Anderer PC               *
       BNE.S  BUSEP4         B: Anderer PC=>Suspendierung   *
*     PC war bekannt, Task fortsetzen                       *
       ADDQ.L =PCSKIP,SGOLPC(A2) PC-erhoehen                *
       MOVEQ  =0,D5          Immer fortsetzen               *
      _MOVEQ  =1,D0          Fuer Ausloesende Task:Bus-Error*
*     Ausstieg                                              *
BUSEP4 MOVEA.L CIADR,A1      SSRP-Adresse                   *
       JMP     EXRTN(A1)     Ausstieg                       *
*-----------------------------------------------------------*
```

Da `D0` nicht zu den Registern gehört, die von `EXRTN` restauriert werden, kann die eigene Ausnahmebehandlungsroutine hier sogar der auslösenden Task eine Nachricht hinterlassen.

Dieses Programm ist übrigens komplett transferassemblierbar, so daß innerhalb der **RTOS–UH**-Welt prozessorunabhängig programmiert werden kann.

So wie das Programm dort steht, hat es noch einen entscheidenen Nachteil: Je nachdem, ob das Bit `SGCBFM` in `SGCNTL` gesetzt ist, wird bei einem Bus-Error immer oder nie eine Fehlermeldung abgesetzt. Es fehlt noch die Möglichkeit, im nachhinein, also beim Ausstieg über `EXRTN` die Meldung nachzuholen. Anstelle des `MOVEQ =1,D5` codieren wir `MOVE =$8001,D5`. Damit ist auch das letzte Manko unseres Anwendungsbeispiels behoben.

8.10.4 Interna

Im Taskkopf gibt es eine Speicherzelle für den „Exception-Frame.“ Ist dieses Langwort nicht gelöscht, zeigt es auf die in Tabelle 8.11 dargestellte Struktur.

`SIGTOT` ist der minimal benötigte Platz, der dem Betriebssystem zur Verfügung gestellt werden muß, falls man einen eigenen Exception-Handler schreiben möchte. Bei Ansprung des Exception-Handlers sind alle Prozessor-Register restauriert. `D7` steht zur Verfügung, da es über `SGOLD7` jederzeit restauriert werden kann.

Der **RTOS–UH**-interne Exception-Handler ist in der Lage, nach dem Absenden der Fehlermeldung alle Prozessor-Register mit den Werten zu belegen, die zum Zeitpunkt der Prozessorausnahme in den Registern standen. Dieses ist für eine Fehleranalyse und ein eventuelles Fortsetzen wichtig. Der Handler benötigt daher den hinter `SIGTOT` liegenden Speicherbereich. Der Gesamtbedarf ist durch das Label `SGEFFS` festgelegt und hat bei 68xxx-Prozessoren den Wert `SGESSR`, bei PowerPC-Prozessoren den Wert `SGEPFS`.

Die ersten 6 Bytes der 12 Byte langen Fehlermeldung haben folgenden Aufbau: Zuerst kommt das 2 Byte lange Error-Codewort, gefolgt vom TID der auslösenden Task. Steht im letzten Wort `$0001`, steht im Langwort hinter dem TID eine auszugebende 32-Bit Hexadezimalzahl. Lautet das letzte Wort `$0000`, folgt dem TID ein Zeiger auf ASCII-Text, der auszugeben ist. In allen anderen Fällen werden die letzten 6 Byte als ASCII-Text interpretiert und ausgegeben. Bei den ASCII-Texten gelten die Zeichen „;␣“ und die ASCII-Werte `$00` bis `$1F` als Begrenzer. Der „_“ wird durch ein „␣“ ersetzt.

Label	Offset	Länge	Bedeutung
SGAEB1	0	2	Hier steht immer ein $AEB5 zur Validierung.
SGCNTL	2	2	Kontrollwort. Im Wesentlichen das bei Aufruf von EXLKL übergebene D7.
SGRESV	4	4	1 Langwort Reserve für Erweiterungen. Z. Z. 0
SGTGPC	8	4	Der Zeiger auf den Exception-Handler.
SGOLPC	12	4	PC, der die Ausnahme auslöste.
SGOLSR	16	2	68xxx Prozessor: Status-Register vor Ausnahme. PPC: Trace-Bit steht im MSB
SGOLD7	18	4	D7 vor der Ausnahmebehandlung. Dadurch hat der Exception-Handler erst einmal ein Register zum Arbeiten.
SGEM12	22	12	12 Bytes Fehlermeldung. Erklärung weiter unten.
SIGTOT	34	0	Minimaler Speicherplatz für einen Exception-Frame.
SGERPT	34	4	Zeiger auf den Error-Pfad für die Ausgabe von Fehlermeldungen.
SGEXPC	38	4	Zeiger auf den externen PC, falls ein Anwender bestimmte Ausnahmen selbst bearbeiten möchte.
SGEXEL	42	4	Zeiger auf Liste mit Code-Wörten, die der Anwender selbst bearbeiten möchte.
SGSTAC	46	4	1 Langwort Stack.
SGSREG	50	32	Speicherplatz zum Retten verschiedener Register.
SGESSR	82	0	Speicherplatzbedarf des **RTOS–UH**-internen Handlers bei einem 68xxx-Prozessor.
SGRES2	82	10	PPC: Reserve für zukünftige Erweiterungen. Z. Z. 0
SGPCR	92	4	PPC: Condition-Register vor der Ausnahme.
SGPXER	96	4	PPC: xer-Register vor der Ausnahme.
SGPLR	100	4	PPC: link-Register vor der Ausnahme.
SGPREG	104	28	PPC: Platz für r25...r31 vor der Ausnahme.
SGEPFS	132	0	Speicherplatzbedarf des **RTOS–UH**-internen Handlers bei einem PowerPC-Prozessor.

Tabelle 8.11: Struktur von Exception-Frames

Kapitel 9: Das Scheibenkonzept

9.1 Die Systemkonfigurierung

Das Betriebssystem besteht aus einer Vielzahl kleiner Module, die vollkommen lageunabhängig sind. Die gegenseitigen Verbindungen zwischen den Modulen werden erst nach Einschalten des Systemes im Grundmodul, dem „Nukleus“, hergestellt. Dazu baut der Nukleus im unteren Speicherbereich ab `$800` eine Vielzahl von Tabellen auf, in denen die Querbezüge ihren Niederschlag finden. Die Querbezüge werden durch Abtasten („Scanning“) eines vereinbarten EPROM (oder RAM–) Bereiches aufgespürt. Signalmarke ist dabei die auf gerader Adresse beginnende Bytesequenz `$AEB1BF95`, gefolgt von einem Wort mit einem ungeraden Vielfachen der Primzahl 37. Wir sprechen von der *Scheibe x*, wenn dieses Wort genau den Wert `(x*2+1)*37` besitzt.

Die Signalmarke ist so gewählt, daß sie im normalen Assemblercode sowie im VCP–Code nicht vorkommen kann. Man hüte sich aber, ungenügend vorbesetzte Speicherbereiche abtasten zu lassen. Auch besteht eine gewisse Gefahr, wenn große Datenfelder im Betriebssystembereich abgelegt werden sollen.

Das *Zusammenbauen* eines Betriebssystemes besteht nun einfach in einer möglichst lückenlosen Hintereinanderreihung der einzelnen Module ('Scheiben') und dem Besetzen des Scanbereiches in den dafür vorgesehenen Zellen des Nukleus.

Das 68k–System startet auf der Adresse `Nukleus+$18` im Supervisormode.
Das PowerPC–System startet auf der Adresse `Nukleus+$20` im Supervisormode.

Der Scanbereich muß immer mit der Anfangsadresse des Nukleus beginnen. Er kann absolut oder relativ zur Nukleusanfangsadresse definiert werden. Dabei sind maximal zwei getrennte Bereiche möglich. Dies geschieht durch Eintragung von 4 Byte–Adressen auf den Zellen `Nukleus+$20` beim 68k und auf `Nukleus+$28` beim PowerPC.

Beispiel 68k:

```
Nukleus+$20   DC.L $00000001,$0000FFFF    relativ
       +$28   DC.L $F60000,$F7FFFE        absolut
```

Beim PowerPC sind die Ablageadressen um 8 höher. Gesetztes LS–Bit heißt „relativ zum Nukleus“. In der ersten Sektion werden also 64 kB, in der zweiten 128 kB abgetastet. Fehlt der zweite Bereich, so sind dort 4 bytes Nullen abzulegen.

Denken Sie also bitte daran, daß bei einer Erweiterung des Systemes ggf. der Scanbereich im Nukleus angepaßt werden muß. Dabei genügt es, wenn die letzte Signalmarke noch innerhalb des Bereiches liegt, der Code darf also darüberhinausragen.

Beim Scanning darf kein Bus–Error auftreten, sonst läuft das System nicht an.

Das kleinste denkbare System besteht nur aus dem Nukleus. Allerdings leistet es nichts, außer daß die Task `#IDLE` dauernd läuft. Durch Hinzufügen einer oder mehrer Scheiben wird daraus entweder ein kleines reines Laufsystem oder ein komfortables Entwicklungssystem, je nach dem beabsichtigten Einsatz.

9.2 Modifikation eines Systems

Ein normal aufgebautes Entwicklungssystem startet mit einer Überschrift, an der man in etwa erkennen kann, aus welchen „Scheiben“ das System besteht. Gedacht ist diese Überschrift auch zur Erkennbarmachung der jeweiligen Revisionsstufen, die meist durch ein „=“ Zeichen angehängt wird. Allerdings gibt es wesentlich mehr Scheiben, als sich in der Überschrift melden. Die folgende Tabelle kann daher nur einen groben Überblick geben:

`Nuc`	Betriebssystemkern, enthält Speicherverwaltung, Taskmanagement, stellt Systemtraps zur Verfügung
`Daemon`	Systemdaemon, höchstpriore Systemtask, ist für korrekte Auswertung und Ausgabe von im NUKLEUS erkannten Fehlersituationen zuständig und startet Bedieneingriffe
`EdFm`	ED-Filemanager, die Datenstation /ED
`Vi/Vo`	/VI und /VO (Pipe) Datenstationen
`Math`	Das mathem. Paket mit SIN, COS etc.
`Hyp`	Der Hyperprozessor, PEARL-Laufzeitsystem
`Editor`	Der kleine Standardeditor, Bedienbefehl ED
`Help`	Kurzhilfe, der HELP-Bedienbefehl
`Sh/sr`	Das Shell-subroutine package, SSRP
`Shell`	Die Grundshell mit den Standardbedienbefehlen
`XC`	Die Datenstation /XC (remote command)
`Loader`	Der Systemlader
`copy`	Bedienbefehl COPY
`Fm`	Filemanagement, UHFM, evtl. auch zusätzlich MSFM
`P`	PEARL-Compiler
`Imp`	Die zur Hardware gehörende Implementierungsscheibe
`EX`	Bourne-Shell Interpreter
`Net`	Netzwerkscheibe.

Sämtliche dieser Module sind aus Scheiben gemäß der in diesem Kapitel folgenden Beschreibung aufgebaut. Zusätzlich zu den Modulen umfaßt eine Implementation die Integration eines Schutzmechanismus, der Veränderungen des Codes im EPROM registriert.

Die Implementierung eines neuen Systems beginnt mit der Erstellung des (rechnerspezifischen) `Imp`–Moduls. Danach folgt die Zusammenstellung der einzelnen Module in geeigneter Reihenfolge sowie die Festlegung des zu schützenden Code–Bereiches. Es sind hierbei verschiedene Systemkonfigurationen mit unterschiedlichem Leistungsumfang denkbar:

- Minimalsystem
  umfaßt nur das Modul `Nuc`. Das Betriebssystem läuft vollständig, alle Traps sind angeschlossen, Schnittstellen sind nicht ansprechbar.

  Ein derartiges System hat praktisch keinen sinnvollen Einsatz, da Kommandointerpreter und Schnittstellentreiber fehlen. Es kann jedoch Programme, die keine Unterstützung durch `EdFm`, `Math` und `Hyp` benötigen, aus dem EPROM heraus exekutieren.

- Laufzeitsystem für Assemblerprogramme
  umfaßt `Nuc`, `Daemon`, `Sh/sr`, `Shell` und `Imp`. Das System kann nach au-

ßen kommunizieren, Kommandos empfangen und exekutieren. Es leistet ähnliches wie das Minimalsystem, ist jedoch von außen steuerbar.

- komplettes Laufzeitsystem
  ist erweitert um `EdFm`, `Math` und `Hyp`, ggf. `Fm`. Das System kann sämtliche in PEARL oder Assembler erstellten Programme bearbeiten und ggf. den Massenspeicher verwalten. Da der Lader noch nicht vorhanden ist, können nur EPROM–residente Programme ausgeführt werden.
- potentielles Entwicklungssystem
  ist erweitert um `Loader`, `EdFm` und die nicht in der Überschrift stehende `VCP`-Scheibe. Das System kann Programme von Massenspeicher oder Schnittstellen laden und exekutieren. Damit kann das System beliebige Programme ausführen.
- komplettes Entwicklungssystem
  enthält zusätzlich Compiler, Assembler und Windoweditor `WE`, entweder EPROM–resident oder von Massenspeicher nachgeladen. Hiermit ist der volle Leistungsumfang eines Standard–**RTOS–UH**/PEARL–Systems erreicht.

Bei einer regulären Implementierung eines Entwicklungssystems werden nur bestimmte Module im unteren Systembereich vom Schutzmechanismus erfaßt, d. h. es ist möglich, ein System oberhalb dieser Schutzgrenze zu modifizieren. Je nach Hardware kann dies durch Hinzufügen oder Ersetzen von Scheiben im EPROM oder im Bootbereich erfolgen. Die hinzugefügten Scheiben können auch Anwenderprogramme sein.

9.2.1 Beispielhafte Systemerweiterung

Betrachten wir als Beispiel ein System mit folgender Hardware–Konfiguration:

Prozessor:	68000
EPROM–Bestückung:	4x 27512, organisiert in zwei Bänken
EPROM–Adressen:	Bank1 : `$F80000` – `$F9FFFE`
	Bank2 : `$FA0000` – `$FBFFFE`

Bei einer derartigen Konfiguration enthält im Regelfall Bank1 das potentielle Entwicklungssystem (mit Schutzmechanismus) und Bank2 Compiler und Assembler (ungeschützt). Compiler und Assembler mögen die Adressen `$0000`–`$AECB` in Bank2 (relativ zu `$FA0000`) belegen.

Aufgabe 1

Als einfachen Einstieg betrachten wir die Umdefinition der Eigenschaften einer Datenstation. Erforderlich hierzu ist die Kenntnis der `LDN` dieser Station (s. Kapitel Datenstationen) sowie der Device–Parameter (s. Befehl `SD`, Seite 203). Die zweite serielle Schnittstelle (`/A2/`), die vom System standardmäßig als Datenschnittstelle mit den Device–Parametern `$0B00` vorbesetzt wird, sei beim Kaltstart des Systems als Terminal–Schnittstelle mit den Device–Parametern `$3300` zu initialisieren (Standard bei `/A1/`). Erforderlich hierzu ist die Erstellung einer 10–er Scheibe gemäß der Beschreibung am Ende dieses Kapitels.

Wir kodieren hierzu in Assembler

```
RORG   0            relativierende Assemblierung
DC     $AEB1        erstes Wort der Signalmarke
DC     $BF95        zweites Wort der      "
DC     (10*2+1)*37  Kennung Scheibe 10

DC     2            LDN der A2-Schnittstelle
DC     $3300        gewuenschte Device-Parameter
DC     0            Endekennung dieser Scheibe

 END                Ende Assemblierung
```

Wird dieser Text assembliert, so erhält man

1. in der Assemblerliste eine auf Adresse 0 beginnende Speicherbelegung `$AEB1`, `$BF95`, `$0309`, `$0002`, `$3300`, `$0000`
2. die hierzu gehörenden S–Records in der CO–Datei des Assemblers.

Wird der generierte Code in der EPROM–Bank2 ab Adresse `$AECC` abgelegt, so übernimmt das System die gewünschte Device–Parametrierung beim Kaltstart.

Aufgabe 2

Als etwas aufwendigere Aufgabe betrachten wir die Einbindung eines PEARL–Programmes in das System zur Ausführung als Auto–Start–Task nach Kalt– und Warmstart des Systems. Gleichzeitig sollen Compiler und Assembler aus dem System entfernt werden, um den verfügbaren EPROM–Platz zu vergrößern. Voraussetzung hierfür ist das Vorhandensein des `PROM`–Befehls.

a) nicht elegant, aber einfach:
Ist das PEARL–Programm geschrieben und übersetzt, so laden wir es (möglichst mit Size–Angabe gemäß Compilerbilanz) an die Speicheradresse, auf der es im späteren Betrieb ausgeführt werden soll. Anschliessend kann mit der Anweisung

`AUTOSTART` *`modulname,taskname`*

die gewünschte Task in eine Autostart–Task umgewandelt werden. (Bitte jetzt nicht `ABORT` betätigen — die Task würde sonst als Autostart–Task sofort starten!). Es ist allerdings sinnvoller, gleich beim Codieren in PEARL der Task das „MAIN“-Attribut zu geben. Mit dem Befehl

`PROM` *`modulname*`*

können wir nun (s. Beschreibung Befehl PROM, Seite 186) S–Records erzeugen. Diese S–Records sollten vom EPROMmer ab Adresse `$0` der Bank2 in einen EPROM–Satz programmiert werden. Werden die so programmierten EPROMs statt der Compiler/Assembler–EPROMs eingesetzt, so startet das System sofort nach Kalt– oder Warmstart unsere Autostart–Task. Das PEARL–Programm wird nun allerdings nicht aus dem EPROM heraus exekutiert, sondern beim Kaltstart auf die Adresse kopiert, auf der wir es beim PROMmen geladen hatten.

b) eleganterer Weg:
Nach einem ersten Compilerlauf merken wir uns die Größenangabe aus der Compilerbilanz (`$`*`xxxx`* `BYTES`), z. B. `$E78`. Für einen zweiten Compilerlauf fügen wir vor den Anfang des PEARL–Moduls die Zeile

```
SC=$E78,CODE=$FA0000,VAR=$3000;
```

in den Quelltext ein. Der Compiler generiert nun Code, der aus dem EPROM heraus exekutiert werden kann (beginnend auf Adresse `$FA0000`), lediglich der Modulvariablenbereich muß hierzu im RAM vorhanden sein. Nach dem Laden (auf beliebige Adresse) können wir (falls wir das „MAIN“-Attribut vergessen hatten) mit der Anweisung

`AUTOSTART` *`modulname,taskname`*

die gewünschte Task in eine Autostart–Task umwandeln. Mit dem Befehl

```
O /ED/SR;PROM modulname*
```

können wir nun (s. Beschreibung PROM-Befehl, Seite 186) S–Records erzeugen. Im File `/ED/SR` erscheinen zwei `S0...S2...S9` Blöcke (z. B. Zeile 1 – 27 und Zeile 28 – 67), die wir mit der `SC`–Option des `COPY`–Befehls in getrennte Dateien kopieren (zwei Durchgänge).

Die S–Records des zweiten Blocks müssen mit einem EPROMmer ab Adresse `$0` der Bank2 (entspricht der physikalischen Adresse `$FA0000`, s. `CODE=` Angabe) in einen Satz EPROMs programmiert werden. Die S–Records des ersten Blocks können in einen beliebigen, ausreichend großen Freiraum des EPROM–Satzes gebrannt werden. Haben wir sichergestellt, daß unsere EPROM–Daten noch vom Scan–Bereich des Systems erfaßt werden (ggf. `Nukleus+$20` aufwärts ändern), so können wir unsere EPROMs statt Compiler und Assembler einsetzen.

Statt der obigen Aktion mit Laden und `PROM`-Befehl ist normalerweise die Verwendung der `PROM`-Option des Linkers bequemer.

Beim Kaltstart des Systems richtet der Nukleus nun im RAM folgende Abschnitte ein:

1. den Modulvariablenbereich, beginnend bei Adresse `$3000` gemäß der `VAR=` Angabe,
2. und Taskköpfe für alle Tasks unseres Moduls ein. Unsere Autostart–Task wird nach der Warmstartphase gestartet.

Aufgabe 3

Als aufwendigsten Fall betrachten wir die Einbindung eines E/A–Treibers gemäß dem Beispiel von Seite 614.

Zunächst kodieren wir einen Scheibenkopf für eine 1–er Scheibe (Systemtask–Definition):

```
DC    $AEB1,$BF95,(1*2+1)*37
DC.B  $01,$80     Normale Task als Betreuungstask
DC.B  'MYQUE '    Name der Task
DC    -1          sehr hohe Prioritaet
DC.L  $100        256 Byte Workspace noetig
                  (je nach Bedarf, muss aber > $66)
DC.L  START-$     relativierte Startadresse
DC    ldn         LDN wie im Beispiel
DC    0           Endekennung fuer diese Scheibe
```

Da wir unsere Routine im EPROM ablegen wollen, können wir eventuellen Speicherbedarf der Interrupt–Routine nicht über DC–Anweisungen reservieren. Daher müssen wir beim Kaltstart einen Speicherbereich als Interrupt–Puffer anfordern, d. h. als Speicherbereich, der nur unserer Interrupt–Routine zur Verfügung steht. Wir entnehmen mit der auf Seite 641 beschriebenen Methode die schon belegten Interrupt–Puffer und wählen einen der freien Interrupt–Puffer. In diesem Beispiel verwenden wir den IDP5. Zur Anforderung kodieren wir daher eine 6–er Scheibe:

```
DC  $AEB1,$BF95,(6*2+1)*37 6-Scheiben-Marke
DC  100                    z.B. 100 Byte Puffer
```

Das System reserviert hierdurch 100 Byte Speicherplatz. Die Adresse des ersten Bytes dieses Speicherbereiches finden wir nach dem Kaltstart des System auf der Adresse $842.

Weiterhin soll unsere selbstdefinierte Datenstation auch unter einem Namen (Mnemo) vom Bedieninterface her erreichbar sein (von der PEARL–Ebene her ist unsere Datenstation nur über /LD/*ldn.drive/* erreichbar). Wir kodieren daher weiterhin eine 9–er Scheibe:

```
DC    $AEB1,$BF95,(9*2+1)*37  9-er Scheiben-Marke
DC.B  'MQ'              Mnemo
```

```
DC.B  ldn+$80          LDN und Textende-Markierung
DC.B  0                Laufwerksnummer, wir nehmen
                       hier Drive 0
DC.B  0                Ende der Scheibe
```

und haben hiermit den Mnemo–Anschluß installiert.

Wollen wir auch noch Eigenschaften der Datenstation vorbesetzen, so können wir dies mit einer 10–er Scheibe (s. Aufgabe 1) tun.

Sollte unsere Datenstation über einen Peripheriebaustein mit der Außenwelt kommunizieren, so muß dieser Baustein im Regelfall beim Warmstart (Abort, oder nach der Reset–Kaltstart–Phase) initialisiert werden. Hierzu kodieren wir eine 15–er Scheibe:

```
DC      $AEB1,$BF95,(15*2+1)*37
...             Maschinenkode zur Initialisierung
RTS             Ruecksprung
```

Der Nukleus springt diese Code–Sequenz im Supervisor–Mode an. Bei der Erstellung des Maschinencodes müssen wir die bei der Beschreibung der Scheibe gemachten Einschränkungen beachten. Nun können wir mit der Programmierung der Datenstation gemäß dem Beispiel von Seite 614 beginnen:

```
START TOQ             Maschinencode wie im
      ...             Beispiel
      ...
      RELCE
      BRA    START
```

Da der von unserer Datenstation angesprochene Peripheriebaustein Interrupts auslösen kann, müssen wir noch einen Interrupt–Antwortroutine erstellen. Hierzu kodieren wir zunächst eine 14–er Scheibe:

```
DC  $AEB1,$BF95,(14*2+1)*37  Kennmarke
DC  $220,IRENT-$-2   setzt Interruptvektor auf
                     unsere Interrupt-Routine
DC  0                Endemarke
```

wobei wir annehmen, daß unser Peripheriebaustein genau diesen Interrupt erzeugen kann. Selbstverständlich müssen wir hierzu die Eigenschaften der Hardware kennen und wissen, welche Interruptvektoren uns zur Verfügung stehen.

Nun können wir den eigentlichen Interrupt–Prozeß wie im Beispiel gemäß Seite 614 kodieren (Formatdatei `SUPERVIS.FOR` included):

```
IRENT IRENTC IRMAL,$220  Rette alten IID, lege Zeiger auf
                         IRMAL davor etc.
      ...                Maschinencode wie im Bei-
                         spiel, merke: Interruptpuffer
                         ueber IDP1 ... IDP7 erreichbar
      ...
IRMAL ...                Malfunction, wie im Beispiel
```

Damit ist der gesamte Code für unsere Datenstation erstellt. Wir brauchen nur noch zu assemblieren und können den erhaltenen Code ins EPROM programmieren.

Beachten wir, daß ggf. der Scan–Bereich des Nukleus angepaßt werden muß, um alle Scheibenköpfe zu erfassen, und haben wir bei der Programmierung darauf geachtet, daß unser Programm frei im Speicher verschieblich ist (keine absoluten Sprünge etc.), so sollte unsere Datenstation einsetzbar sein.

9.3 Beschreibung der Scheiben

Im folgenden werden die Scheiben in numerischer Reihenfolge beschrieben. Dies bedeutet jedoch keineswegs, daß sie in dieser Reihenfolge vom Abtaster erfaßt werden oder in dieser Reihenfolge im abgetasteten Speicher stehen müssen.

Scanbereich bedingt überspringen **Scheibe: -1**

Signalmarke: `$AEB1`, `$BF95`, `$FFDB` $(-1 \cdot 2 + 1) \cdot 37 =$`$FFDB`

Diese Scheibe ist nur wirksam, wenn nicht der nukleuseigene normale Scan–Trap, sondern die Hochgeschwindigkeitsscanscheibe (Scan–accelerator) benutzt wird. Der Scan–accelerator sammelt nämlich zunächst alle Adressen mit Signalmarken in einer eigenen Liste und beim Sammeln dieser Adressen können mit Hilfe dieser (–1)–Scheibe einzelne Adreßbereiche in Abhängigkeit von bestimmten Systemreaktionen übersprungen werden. Dazu wird während der Kaltstartphase ein in der Scheibe vereinbartes Unterprogramm exekutiert, welches mit `EQ =` nicht springen oder mit `NE =` springen antworten muß. Da der Adreßsammler diese Operation ausführt, können logischerweise keine 0–er Scheiben damit übersprungen werden. Auch können nur solche Kaltstartscheiben (Nr. 18) übersprungen werden, die später als die Scanaccelerator–Scheibe vom Abtaster erfaßt werden.

Aufbau hinter der Signalmarke (`COMEQU` included):

```
Entry DC.L   Cont-$    Langrelative Sprungweite
      MOVEA.L A7,A6    Stack retten
     _MOVE.L BUSELK,D7 Buserror-Link retten
      LEA    Exit,A0   Aussprung
      MOVE.L A0,BUSELK neuer Buserror
      MOVEQ  =1,D0     Set to 'NE' = Scanfortsetzung bei Cont
      ...              Alle Register frei bis auf D7,A6,A7
      .....            1. Statement der Testroutine
      MOVEQ  =0,D0     Set to 'EQ' = Scanfortsetzung bei Entry
Exit _MOVE.L D7,BUSELK Buserror restaurieren
      MOVEA.L A6,A7    Stack zurueckladen
      TST    D0        Trap-Antwort
      RTS              Rueckehr in den Scanaccelerator
      ...
      ...              Hier liegen evtl. ueberspringbare Scheiben
      ...
Cont  EQU     $        Marke zur Scanfortsetzung
      END              Normalerweise Ende der Scheibe hier
```

Scheibe: -1 **Fortsetzung**

---> Die Scheibe ist für den Sonderfall gedacht, wenn z. B. ein System mit unterschiedlichen Kartenbestückungen hochgefahren werden soll und **RTOS–UH** sich selbsttätig an die aktuellen Gegebenheiten anpassen muß. Wenn etwa eine serielle Schnittstellenkarte nicht eingesteckt ist, so kann mit Hilfe der Scanacceleratorscheibe und einer solchen (–1)–er Scheibe verhindert werden, daß die fehlende Karte initialisiert wird. Auch der Einbau des zugehörigen Treibers in das System läßt sich unterdrücken.

---> Ein evtl. Buserror muß durch Retten und vorübergehende Änderung des Links (`BUSELK`) unbedingt selbst abgefangen und in `NE` verwandelt werden. Sonst wird beim Buserror der ganze gerade aktuelle Scanbereich vorzeitig beendet. (`BUSELK` liegt beim 68k auf `$8`, beim PowerPC auf `$4008`) Es muß auch der Stack geordnet verlassen werden (`A7` muß richtig stehen).

Erweiterte Scan-Tabelle anschließen **Scheibe: 0**

Signalmarke: `$AEB1`, `$BF95`, `$0025` $(0 \cdot 2 + 1) \cdot 37 =$`$0025`

Bevor überhaupt irgendeine Abtastung beginnt, inspiziert der Nukleus, ob in seiner Scan–Bereichsbeschreibung auf den Zellen `Nukleus+$20` (beim PowerPC 8 höher!) nicht evtl. die beiden ersten Langworte Null sind. Ist dies der Fall, dann nimmt er die Langadresse auf `Nukleus+$28` (beim PowerPC 8 höher!) als absolute (1–er Bit ist Null) oder zum Nukleus relative (1–er Bit ist 1) Startadresse, um von dort aus die erste 0–er Scheibe zu suchen. Weiter dahinter liegende 0–er Scheiben werden nicht mehr berücksichtigt. Hinter der 0–er Signalmarke folgt nun eine beliebig lange Tabelle mit absoluten (gerade) oder relativen (ungerade) Adressen, die die einzelnen Scan–Bereiche festlegen.

Aufbau hinter der Signalmarke:

```
DC.L  Start1,End1     Bereich 1
DC.L  Start2,End2     Bereich 2
DC.L  Start3,End3     Bereich 3
....
DC.L  0               Stop-marke
```

Die Tabelle selbst hat also genau die gleiche Struktur, wie jene, die gewöhnlich im Nukleus steht — sie ist jetzt aber nicht mehr auf max. 2 Scanbereiche beschränkt.

Beispiel:
```
DC  $AEB1,$BF95,$0025  Signalmarke
DC.L $1,$1FFFF         Relativ zum Nukl. 128 kByte
DC.L $800000,$80FFFE   absolut 64 Kbyte dort oben
DC.L $30001,$5FFFF     relativ zum Nukl.  192 kByte
DC.L 0                 Stop-marke
```

`--->` Nicht vergessen, auf `Nukleus+X` die 3 magischen Langworte einzusetzen (`X=$20` beim 68k, `$28` beim PowerPC!):

```
Nukleus+X     $00000000   Triggert Suche der 0-er-Slice
  -''- +X+4   $00000000          -----  '' -----
  -''- +X+8   $0001F001   obige 0-er Scheibe wird hier
                          ab  Nukleus+$1F000 gesucht
```

Falls bei dieser Besetzung der Zellen keine 0–er Scheibe gefunden wird, läuft das System nicht an. Die Verwendung der 0–er Scheibe ist nur sinnvoll, falls mehr als 2 Scan–Bereiche benötigt werden oder andere Gründe für eine außerhalb des Nukleus liegende Tabelle sprechen.

Scheibe: 1 — Systemtask definieren

Signalmarke: `$AEB1`, `$BF95`, `$006F` $(1 \cdot 2 + 1) \cdot 37 =$`$006F`

Bytes hinter der Signalmarke:

`$0`	Type (s. u.)
`$1`	Class (s. u.)
`$2 ... $7`	Name-info
`$8 ... $9`	Priority
`$A ... $D`	Size of required Workspace, 4 bytes.
`$E ... $F`	Start-PC minus location of this word.
`($10 ... $11)`	Only if class is $80 or $01: `LDN`.
`$10 (or $12)`	Type (next task)
`$11 (or $13)`	Class (next task)
`...`	
`...`	Start-`PC` relative or `LDN` of last task.
`DC.W 0`	$0000 Stop-marker this slice.

Type Ist das spätere 2. Byte des Typwortes im **RTOS–UH**:

`$01`	Normale Task.
`$81`	Residente Task.
`$41`	Normale Task mit Autostart.
`$C1`	Residente Task mit Autostart.
`--> andere`	nicht erlaubt, reserviert

Class Gibt an, welche Sonderform der Task vorliegt:

`$00`	Normale Usertask.
`$01`	Primaere Shell für `LDN` (s. o.)
`$02`	Der Systemdämon `#ERRDM`.
`$80`	Eine I/O-Queue-Betreuungstask für die Warteschlange der `LDN` (s. o.) `MSB` oder `LDN` null oder 2. `LDN` (z. B. `VI`/`VO`)
`--> andere`	nicht erlaubt, reserviert

Fortsetzung **Scheibe: 1**

--> Es darf im System nur einen Dämonen `#ERRDM` geben. Ein Austausch des standardmäßig vorhandenen ist nur bei sehr kleinen reinen Laufsystemen sinnvoll - etwa um eine LED im Fehlerfall aufleuchten zu lassen etc. Die Übergabedaten finden sich im ringförmigen Errorpuffer des Systemes.

--> Bei den I/O-Tasks wird der entsprechende Anschluß für den Trap `XIO` automatisch hergestellt. Zum Bekanntmachen des Stationsnamens und der Eigenschaften sind besondere Scheiben vorgesehen.

Name-info Ist auf 2 Arten wahlweise kodierbar:

1. Genau 6 ASCII-Bytes (wie im Beispiel)
2. Relativierter 4-Byte-Zeiger auf den Namensstring, der durch `$FF` beendet wird. In diesem Fall haben die restlichen 2 Bytes keine Bedeutung, müssen aber als Nullwort angelegt werden. Die Relativierung erfolgt über

```
        ...
        DC.L  Textad-$
        DC.W  0
        ...
Textad  DC.B 'mueller',$FF
        ...
```

Beispielscheibe

```
        DC    $AEB1,$BF95,(1*2+1)*37          Signalmarke.
        DC.B  $C1,0                     Resident, Autostart, normal
        DC.B  '#SELFT'                      6 bytes name of task
        DC    $7FF0                     Extremely low priority.
        DC.L  $66                       Workspace ($66 is minimum!!)
        DC    START-$                       Start-PC relative
        DC    $0                        Stop-marke für diese Scheibe.
        ...
START   MOVE  ...                       Anfang der Systemtask.
```

Scheibe: 1 **Fortsetzung**

Hinweis 1

Alle residenten Systemtasks erhalten bereits beim Kalt- oder Warmstart ihren Taskworkspace im Bereich unterhalb des dynamisch verwalteten Speicherbereiches. Damit soll der Verwaltungsaufwand reduziert werden.

Hinweis 2

Tasks mit gesetztem Autostart-Bit (wie im Bsp.) laufen beim Aufsetzen des Systemes unter Berücksichtigung der festgelegten Priorität sofort los. Auch I/O-Betreuungstasks dürfen ggf. vom Autostart-Typ sein, obwohl eine etwaige Peripheriebaustein-Initialisierung in einer Warminitialisierungsscheibe besser aufgehoben ist.

Interruptbuffer installation **Scheiben: 2 ... 8**

Signalmarken: `$AEB1`, `$BF95`, $(x \cdot 2 + 1) \cdot 37$ $x = 2, 3, 4, 5, 6, 7, 8$

Das System stellt einen Interruptpuffer für den Level $(x-1)$ zur Verfügung des Systemprogrammierers. Dabei wird während des Autolinking für jeden Level nach der größten Anforderung gesucht. Nur dieser Maximalwert wird berücksichtigt. Auf den Zellen „`IDP`y“ legt das System einen 4 Byte Zeiger ab, der auf den Anfang des Puffers für den Level y zeigt.

Wort hinter der Signalmarke: 2 Byte lange Puffergröße, z. B.:

```
DC   $AEB1,$BF95,(4*2+1)*37    Level 3
DC   100                       100 Bytes
```

Falls nirgendwo im abgetasteten Bereich für den Level 3 eine größere Anforderung gefunden wird, so werden genau 100 Byte Puffer für den Level 3 bereitgestellt.

Die Verwaltung der Displacements muß der Systemprogrammierer selbst übernehmen. Insbesondere dürfen bei Einbau dieser Scheibe schon angeforderte Zellen nicht benutzt werden. In Zweifelsfällen kann man aus den `IDP`-Differenzen im Zielsystem ablesen, ab welchen Displacements Platz definiert werden kann. (Im obigen Bsp: Inhalt `IDP4` - Inhalt `IDP3` = Puffergr. 3)

Die 7 `IDP-`Zeiger (4 byte) stehen im System unmittelbar hintereinander, ihre tatsächliche Adresse wird mit der Datei `COMEQU` included. Dabei gilt:

`IDP1=$832` beim 68k
`IDP1=$508C` beim PowerPC

Beispiel zur Verwendung des Interruptpuffers:

```
HILF1    EQU     30                          Start-displacement
HILFXY   EQU     34                          Irgendeine Variable
SIZE     EQU     38                Letztes Byte auf displacem. 37
* ...
         DC      $AEB1,$BF95,(6*2+1)*37                 Level 5
         DC      SIZE
* ...
IRENTR   IRENTC    MALF,Vectoradr T-Code-Format IR-Entry
         MOVEM.L   A0-A3,-(A7)          angenommenes Beispiel.
         MOVEA.L   IDP5,A0                          Basiszeiger
         .....
         MOVE.L    HILFXY(A0), ...                  typ. Zugriff.
```

Scheiben: 2 ... 8 **Fortsetzung**

Hinweise

Der Zeiger `IDP1` wird im System zum Löschen des gesamten Interruptpuffers (bis zum Ende des `IDP7`) bei jedem Warm- oder Kaltstart benutzt, darf also nicht vom Programmierer „verbogen“ werden.

Theoretisch würde ein einziger `IDP` für alle Interrupts genügen, die Aufteilung nach Levels ist nur zur besseren Softwarestrukturrierung gedacht.

Weil sämtliche Zellen in den Interruptbuffern bei jedem System-Neustart und -Abort auf Null zurückgesetzt werden, eignen sie sich auch hervorragend für systeminterne Semaphor- oder Boltvariable.

Definition einer Datenstation **Scheibe: 9**

Signalmarke: `$AEB1`, `$BF95`, `$002BF` $(9 \cdot 2 + 1) \cdot 37 =$`$02BF`

Mit dieser Scheibe werden dem System ein oder mehrere Namen von neu definierten Datenstationen hinzugefügt. Das Bedieninterface kann mit Hilfe der Informationen dieser Scheibe aus einem Textstring dann die zugehörige Warteschlangennummer (`LDN`) und die zugehörige Untergliederungsnummer (`DRIVE`) ermitteln.

Aufbau hinter der Signalmarke:

`$0 ... ?`	Alphanumerischer Textstring beginnend mit Buchstaben.
`$x ...`	1 Byte `LDN` mit aufgeodertem Bit `$80`. (Dient gleichzeitig als Endemarke für den Textstring).
`$x+1`	1 Byte `DRIVE`, z. B. Laufwerksnummer.
`$x+2`	Nächster Textstring wie oben etc.
`$y`	1 Byte `LDN` plus `$80`
`$y+1`	1 Byte `DRIVE`.
`...`	Letzter Block mit Textstring, ldn,drive.
`$00`	Stopmarke, Ende dieser Scheibe.

Beispiel: Sie wollen die Station „PLOTTER“ für die „`LDN 5`“ einrichten, Laufwerksnummer sei 0 bzw. don't care.

```
DC    $AEB1,$BF95,(9*2+1)*37      Signalmarke 9-er slice
DC.B  'PLOTTER',$80+5,$00                 Name, ldn, drive
DC.B  0                  Null markiert das Ende der Scheibe.
```

Scheibe: 9 **Fortsetzung**

--> Damit die Station wirklich benutzt werden kann, müssen Sie noch mit Hilfe der Scheibe 1 einen zur gewählten LDN passenden I/O-Dämonen bereitstellen. (s. Scheibe 1)

--> Wählen Sie die LDN dabei nicht größer als nötig. Der Nukleus sucht im abgetasteten Speicher nämlich nach der höchsten LDN aller 1-er Scheiben (Nicht der 9-er). Diese bestimmt den freigehaltenen Platz in der Tabelle „LDN to TID“ (Zeiger auf Anfang der Tabelle heißt SIOLDT, siehe Datei COMEQU). Diese Tabelle enthält jeweils 4 Byte (den TID) pro LDN).

--> Man kann allerdings auch gezielt Lücken bei den LDNs lassen. Diese Tabellenlücken können durch Anschluß von nachladbaren I/O-Tasks (im verwalteten RAM) sinnvoll genutzt werden. So wird die I/O-Task „nachmontiert“:

```
LDN     EQU     5                       Beispiel-LDN
TID     EQU     $802 bzw. $5000         Actual Task-ID
SIOLDT  EQU     $852 bzw. $50B0         I/O LDN Table start
        ....
START   MOVEA.L SIOLDT,A1               Tab-Zeiger
        MOVE.L  TID,LDN*4(A1)           eigene TID
        TOQ                             aus Schlange
        BRA.B   EXIT                    leer: Ende
        ...                             hier Aktion der I/O-Task
```

Task einmal von Hand starten, ist dann betriebsbereit.

Datenstationseigenschaften setzen **Scheibe: 10**

Signalmarke: `$AEB1`, `$BF95`, `$0309` $(10 \cdot 2 + 1) \cdot 37 =$`$0309`

Die mit den Befehlen `SD` und `DD` („Set Device parameter“ und „Display device parameter“) zugänglichen Geräteeigenschaften können mit Hilfe dieser Scheibe vorbesetzt werden. Beziehen sich mehrere solcher Scheiben auf das gleiche Gerät, so gelten die Eigenschaften der letzten vom Abtaster (Scanner) erfaßten Scheibe.

Aufbau hinter der Signalmarke:

2 Bytes	Start-`LDN` dieser Scheibe
2 Bytes	aa,bb wie bei `SD`, für Start-`LDN`.
2 Bytes	aa,bb wie bei `SD`, für Start-`LDN`+1.
...	...
`$00`, `$00`	Stop-Marke dieser Scheibe.

aa:	$80	Bitfunktion	Station ist rückspulbar (`REWIND`).
	$40	Bitfunktion	Station kennt `OPEN/CLOSE`.
	$20	Bitfunktion	Nach dem CR erwartet Station LF. (CR=Carriagereturn, LF=Linefeed)
	$10	Bitfunktion	Station ist dialogfähiges Terminal
	$08	Bitfunktion	Station möchte kein Echo (RS232).
	$04	Bitfunktion	Station kennt `RM` bzw. `ERASE`.
	$02	Bitfunktion	Station ist für Ausgabe geeignet.
	$01	Bitfunktion	Station ist für Eingabe geeignet.
bb:	$80	Bitfunktion	Station reagiert auf `DIR+FILES`
	$40	Bitfunktion	Station kennt Formatierbefehl.
	$20	Bitfunktion	Station kennt `CF` (Change)-Befehl
	$10	Bitfunktion	Hierarchische Verwaltung, `MKDIR`
	$08	Bitfunktion	`SEEK, SYNC, SAVEPOS` sind erlaubt.
	$04	Bitfunktion	Report Error ist möglich
	$02	Bitfunktion	Terminal (RS232c) Editor-parameter
	$01	Bitfunktion	Terminal (RS232c) Editor-parameter

Scheibe: 10 **Fortsetzung**

Beispiel:

```
DC    $AEB1,$BF95,(10*2+1)*37     Signalmarke
DC    11                          Start-LDN
DC.B  $2B,0                       LDN 11
DC.B  $33,0                       LDN 12
DC.B  $C7,$E0                     LDN 13
DC.B  0,0                         Stop-Marke
```

Die Station `LDN=11` erwartet ein LF nach jedem CR, arbeitet in beiden Richtungen und möchte kein Echo der eingegebenen Daten. Offensichtlich eine typische Hostschnittstelle.

Die Station `LDN=12` ist offenbar ein Terminal zum Editieren.

Die Station `LDN=13` könnte eine Floppy oder Wechselplatte sein.

`--->` Die Scheibe 10 entspricht einem automatischen `SD`-Befehl.

Neuen Bedienbefehl definieren **Scheibe: 11**

Signalmarke: `$AEB1`, `$BF95`, `$0353` $(11 \cdot 2 + 1) \cdot 37 =$`$0353`

Das Bedieninterface wird für alle Nutzer um einen oder mehrere Befehle erweitert. Dazu wird mit dieser Scheibe je ein „Mnemo“ der neuen Anweisung sowie die relativierte Sprungadresse definiert.

`--->` Die definierte Aktion wird auf der Ebene des Bedieninterface mit sehr hoher Priorität ausgeführt. Es ist darum bei zeitaufwendigen Kommandos unbedingt zu prüfen, ob nicht ein Sohnprozess generiert werden sollte.

Aufbau hinter der Signalmarke:

```
 DC.B  '...'     „String des Befehles“ (ohne Blanks im String!)
(DC.B   0)       Nur wenn String ungerade Anz. Bytes besitzt!
 DC.L   Adr-$    Lang relative Ansprungadresse.
 ...
 DC.B  '...'     „String nächster Befehl“
(DC.B   0)       Nur um String auf gerade Bytezahl zu füllen.
 DC.L   Adr2-$   Lang relativierte Ansprungadresse.
 ...
 $FFFF           Stop-Marke, Ende dieser Scheibe.
```

Beispiel: Neues Kommando zum Inkrementieren der Zelle `$FFA000`.

```
        DC        $AEB1,$BF95,(11*2+1)*37          Signalmarke
        DC.B      'INK',$0          Befehlsname, mit $0 aufgefüllt.
        DC.L      INKSR-$                          Ansprung
        DC        $FFFF                            Stop-marke
        ...       ...
INKSR   ADDQ.B    =1,$FFA000                       Eigentliche Operation
        RTS                                        Rücksprung Bedieninterface.
```

Nach Anschlag von `Ctrl A` oder über die `XC`-Station könnte man nach Einfügen dieser Scheibe in den abgetasteten Bereich den Befehl „INK“ oder „ink“ eingeben. Die „Shell“ von **RTOS–UH** wäre also entsprechend erweitert. Das System nutzt selbst teilweise diese Erweiterungsmöglichkeit, z. B. sind die floppyspezifischen Kommandos durch eine kleine 11-er Scheibe innerhalb des Filehandlers eingebaut.

Scheibe: 11 **Fortsetzung**

Das Bedieninterface stellt für die Shellerweiterung eine Reihe von Hilfsfunktionen zur Verfügung. So gibt es die Möglichkeit, den weiteren Text hinter dem Kommando zu analysieren, den Fehlerausgang anzuspringen oder etwa Sohnprozesse zu generieren und mit Parametern (`SI=xx` etc.) zu versorgen.

---> Die Register `D0-D7`, `A3`, `A5` und `A6` können frei benutzt werden. Das Register `A0` zeigt beim Einsprung auf die Unterprogrammtabelle des Shell-subroutine-packages „`SSRP`“. Man kann mit `JSR 4(A0)` etc. die Dienste des `SSRP` anwählen. `A0` kann bei Bedarf verändert werden. Es läßt sich von der Zelle `CIADR` (`$8B8` bei 68k, `$51A8` bei PowerPC) wieder auf die `SSRP`-Unterprogrammtabelle zurückladen.

---> `A1` zeigt auf das Communication Element des Bedieninterfaces, kann notfalls aus dem Task-WSP (über `A4`) nachgeladen werden mit
`MOVEA.L $8C(A4),A1`.

---> `A2` zeigt auf das nächste Zeichen des Eingabepuffers. Damit sind die Traps `CSA` und `QSA` zur Textanalyse direkt anwendbar. Muß geordnet zurückgegeben werden!

---> `A4` zeigt auf den Task-Workspace des Bedieninterfaces.

---> `A5` zeigt auf den Textpuffer eines fertigen Ausgabe-CE des Bedieninterfaces. Mit `(A5)+` kann man hier Text ablegen und mit `JSR 28(A0)` ausgeben. CR und LF werden dabei automatisch angehängt. Die Zielstation ist durch das `O`-Kommando steuerbar. Nach der Ausgabe wird `A5` wieder reinitialisiert.

---> `A7` zeigt auf den Stack der Bedientask. Dieser liegt innerhalb des Task-Workspace und enthält die Rücksprungadresse in den Bedieninterpreter. Platz für max. 5 Longwords.

---> Alle anderen Register stehen frei zur Verfügung, werden jedoch von den u. a. Unterprogrammen teilweise zerstört.

---> Der Rücksprung in den systemseitigen Interpreter muß mit `RTS` erfolgen, es sei denn, daß einer der über `A0` erreichbaren mit `JMP` anzuspringenden Ausgänge benutzt wird.

Fortsetzung **Scheibe: 11**

Tabelle der über `A0` erreichbaren Funktionen (`JSR, JMP`):

`JSR 0(A0)`	Hier nicht sinnvoll: Aufruf des SSRP Dekoders.
`JSR 4(A0)`	I/O-command processor (Dir,files,mkdir etc.) Hinter dem JSR 4(A0) folgende 4 Bytes sind Parameter: 1. Wort: Die notwendigen Facilitybits („SD“„DD“) 2. Wort: links MODE rechts MODE+1 des Comm.El. „CE“.
`JMP 8(A0)`	Subtask (son process)- generation.
`JSR 12(A0)`	Device/File decoder by (A2) to (A3)+ Ablage füllt ein CE exakt, wenn mit A3 auf STATIO(...) zeigend begonnen wird. Work. dir. etc. wird berücks.
`JMP 16(A0)`	Syntaxfehlerausgang („WRONG COMMAND“).
`--- ------`	Nicht innerhalb der Shell-task.
`--- ------`	Nicht innerhalb der Shell-task
`JSR 28(A0)`	Make output of Text stored by (A5)+ CR und LF werden automatisch angehängt.
`JSR 32(A0)`	Scan on class of character by (A2).
`JSR 36(A0)`	Write address in A3 as 8 hex. digits to (A5)+

Scheibe: 12 — RAM-Bereich definieren

Signalmarke: `$AEB1`, `$BF95`, `$039D` $(12 \cdot *2 + 1) \cdot 37 =$`$039D`

Start und Ende einer beliebigen Zahl von RAM-Sektionen, die unter Verwaltung von **RTOS–UH** stehen sollen, werden mit dieser Scheibe definiert. Wenn der Abtaster mehrere Scheiben des Typs 12 erfaßt, so werden nur die Parameter der letzten abgetasteten Scheibe gültig. Man muß also in jedem Fall mit dieser Scheibe eine vollständige Definition angeben.

Aufbau hinter der Signalmarke:

```
DC.L    RAM1begin,RAM1end     Speichersektion 1
DC.L    RAM2begin,RAM2end     Speichersektion 2
...                           ...
DC.L    RAMxbegin,RAMxend     Speichersektion x
DC.L    0                     Stopmarke dieser Scheibe
```

---> Die Adressen müssen gerade sein und oberhalb der 2. Adresse sollen noch `$20` Zellen RAM sein. (Ende ... `E0`)

---> Der Adressbereich der Sektion 2 muß überlappungsfrei höher liegen als der von Sektion 1, der von Sektion 3 entsprechend höher sein als der von Sektion 2 etc.

---> Die zwischenliegenden Lücken werden vom Nukleus als scheinbare (unlöschbare) Module Namens `#NORAM` angelegt und erscheinen entsprechend beim `S`-Befehl.

---> Die letzte Adresse vor der Stopmarke darf zusätzlich noch mit gesetztem LS-Bit versehen werden. Dann wird von der angegebenen Adresse aus bis zum Bus-error oder bis zum ROM oder bis zum Erreichen des Nukleus in 1 kB großen Schritten getestet und die so ermittelte Obergrenze eingesetzt. Achtung: Ende mit `$...E1` setzen!

Fortsetzung **Scheibe: 12**

Beispiele:

```
DC      $AEB1,$BF95,(12*2+1)*37     12-er  Signalmarke
DC.L    $12000,$0001FFE1            'open end'-spezif.
DC.L    0
```

Das verwaltete RAM erstreckt sich von `$12000` bis zum durch Probieren (Buserror, unveränderlich oder Nukleus erreicht) ermittelten oberen Grenzwert.

```
DC       $AEB1,$BF95,(12*2+1)*37     12-er Signalmarke
DC.L     $800,$3FFE0                 Fixierter Bereich
DC.L     $00080000,$000FFFE0         z. B. VME-System.
DC.L     0
```

Die Grenze von `$800` wird automatisch um die Anzahl Zellen, die **RTOS–UH** dort unten anlegt (ohne evtl. **RTOS–UH**-Code!), nach oben korrigiert. VORSICHT, wenn **RTOS–UH**-Code unten im RAM liegt!

Scheibe: 13 — Modulvariablenblock einrichten

Signalmarke: `$AEB1`, `$BF95`, `$03E7` $(13 \cdot 2 + 1) \cdot 37 =$`$03E7`

Bei ROM-residenten miteinander kommunizierenden Tasks tritt das Problem auf, einen gemeinsamen Variablenblock zu definieren und diesen beim Systemstart zu initialisieren. Da die Tasks wegen der großen Distanz auf diese Objekte mit absolut langer Adressierung zugreifen müssen, muß die Adresse dieser Modulvariablen fest definiert werden können.

Aufbau hinter der Signalmarke:

```
DC.L   Adr.1            Anfangsadresse des Moduls, gerade.*
DC.L   Adr.2   Adresse der (freien)  Folgesektion, gerade.*
DC.B   'Name6b'                   6 Byte langer Modulname.*
DC.L   Data1adresse.  (Adr. erstes von Null versch. Wort).*
DC.L   Blocklaenge.               (Anzahl der Datenworte).*
DC.W   Erstes Datenwort.                                  *
....                        ... Datenblock No. 1          *
DC.W   Letztes Datenwort.                                 *
DC.L   Data2adresse.        (Anfang naechster Datenblock).*
DC.L   Blocklaenge.               (Anzahl der Datenworte).*
DC.W   Erstes Datenwort.                                  *
....                        ... Datenblock No. 2          *
DC.W   Letztes Datenwort                                  *
....                                                      *
                            ... Datenblock No. x          *
....                                                      *
DC.L   0     Statt  Datenadresse:  Stopmarke  der Scheibe.*
```

---> Der angegebene feste Adressraum muß beim Hochlaufen wirklich im verwalteten RAM verfügbar sein, sonst läuft das System nicht an. Alle Worte für die keine Initialdaten angegeben sind, werden vom Nukleus zu Null gesetzt.

Fortsetzung **Scheibe: 13**

Beispiel:

```
DC     $AEB1,$BF95,(13*2+1)*37    Signalmarke.           *
DC.L   $3000                      Anfangsadresse.        *
DC.L   $4000                      Naechste FREE- Sektion.*
DC.B   'Testmd'                   Name des Modules.      *
DC.L   $3100                      Datenblock-adr.        *
DC.L   4                          4 Worte zu initialis.  *
DC     1           Datum                                 *
DC     2            -'-                                  *
DC     3            -'-                                  *
DC     4            -'-                                  *
DC.L   0                          Stopmarke der Scheibe. *
```

PEARL-Module für EPROM-Programme werden erst in das Zielsystem geladen, dann mit dem `PROM`-Befehl (siehe Seite 186) in obenstehende Scheibendaten verwandelt.

Bequemer ist in der Regel jedoch die Verwendung des Linkers, der ebenfalls diesen Scheibentyp erzeugen kann.

Scheibe: 14 — Anschluß von Traps und IR-Vektoren

Signalmarke: `$AEB1`, `$BF95`, `$0431` $(14 \cdot 2 + 1) \cdot 37 =$`$0431`

Für nutzerdefinierte Interruptprozesse muß das System eine Möglichkeit zum Besetzen der entsprechenden Adressvektoren schon während der Hochlaufphase bereitstellen.

Aufbau hinter der Signalmarke:

```
        DC.W    Adr.1         Vektor- oder Pseudovektoradresse. *
        DC.W    Jumpadr1-$    Zieladresse fuer Ansprung relativ.*
        DC.W    Adr.2         Naechste Vektoradresse o.ae.      *
        DC.W    Jumpadr2-$    Relativierte Zieladresse.         *
        ....                                                    *
        DC.W    0               Statt Vektoradresse: Stopmarke. *
```

Beispiel: Der Userinterrupt `$200` soll nach dem Aufsetzen auf die Adresse „XYZ“ zeigen.

```
       DC    $AEB1,$BF95,(14*2+1)*37    Signalmarke            *
       DC    $200,XYZ-$-2               -2 wegen 2 Worte im DC!*
       DC    0                          Stopmarke              *
       .....                                                   *
                                                               *
 XYZ   IRENTC   XYZMAL,$200      Save old  Interrupt-Id etc.   *
       MOVEM.L  D0-D3,-(A7)      Privat benutzte Reg. retten   *
       .....                     Interrupt-code nach  Anwendung*
 EXIT  MOVEM.L  (A7)+,D0-D3      Privat benutzte Reg. rueckladen
       IREXTC                    Exit-Format f"ur Interrupts   *
        .....                                                  *
 * Rueckfallroutine, falls Bus-/Adr-/Opcode-exc. im Interrupt: *
 XYZMAL MOVE ...                z.B. Controller abstellen     *
        BRA      EXIT                                          *
```

Interruptprogramme müssen den vorgeschriebenen Ausgang benutzen, da sonst der Taskumschalter (Dispatcher) ausfallen kann.

Fortsetzung **Scheibe: 14**

Die angegebene Vektoradresse rechnet sich relativ zum „Exception origin“ `EXCORG`, der nur beim 68k den Wert Null hat (`$4000` beim PowerPC). Den jeweils passenden Wert von `EXCORG` erhält man bei Bedarf (s.u.) durch Inkluden des Files `COMEQU` automatisch.

Will man den Mechanismus „mißbrauchen“, um eine andere Speicherstelle (die kein Exceptionlink darstellt) vorzubesetzen, so ist `EXCORG` bei der Angabe der Vektornummer abzuziehen:

```
DC   Helplk-EXCORG,HELP-$-2
```

Nun wird die physikalische Adresse `Helpk` mit der Adresse der Routine `HELP` besetzt.

Damit können prinzipiell alle Speicherzellen im Bereich von `EXCORG` bis `EXCORG+$7FFE` auf diese Weise mit Adresszeigern (4 byte Adr) automatisch versorgt werden, leider kann man damit dann auch u. U. systemeigene Anschlüsse zerstören. Es wird bei Scheiben mit gleichen Vektoradressen immer die letzte vom Abtaster erfaßte Scheibe als gültig genommen.

Man beachte bei dem relativierten zweiten 16-Bit Wort, daß das `$`-Symbol im Assembler immer den Anfang des DC-Blockes bezeichnet, das relativierte Wort aber auf seine eigene Adresse bezogen wird. (Daher die Korrektur mit `$-2` im obigen Beispiel).

Warnung:

Wegen der Einführung von `EXCORG` ist bei der Umstellung von 68k-Assemblertext auf T–Code besondere Vorsicht bei allen 14-er Scheiben geboten. Es laufen nur solche Programme korrekt, bei denen tatsächlich ein Interrupt-, Line-A- oder Traplink angeschlossen wird. In den anderen Fällen ist die Korrektur mit `EXCORG` (wie oben) nötig.

Scheibe: 15 **Warmstart Initialisierungscode**

Signalmarke: `$AEB1`, `$BF95`, `$047B` $(15 \cdot 2 + 1) \cdot 37 =$`$047B`

Bei der Verwendung von Peripherie-Kopplern kommt es oft zur Notwendigkeit, deren Controlregistern beim Systemstart eine gewisse Anfangsinformation mitzugeben. Auch z. B. das Auslesen einer batteriegepufferten Uhr (um damit die Planungsuhr des Systemes zu stellen) sollte bei jedem Warm- oder Kaltstart erfolgen. Um dies dem Systemprogrammierer zu ermöglichen, sucht der Nukleus nach Abschluß aller sonstigen Aufsetzoperationen mit dem Scanner (Abtaster) nach 15-er Scheiben, um deren Code nacheinander zu exekutieren.

Aufbau hinter der Signalmarke:

Maschinencode	fast beliebig, aber kein Tasking, I/O etc.
...	Alle Register bis auf `A7` frei verfügbar.
RTS	Rückkehr in den Nukleus.

Wichtiger Hinweis!

Das System befindet sich noch in einem sogenannten „Kernelmode“ auf der Supervisorebene und kann daher seinen Taskumschalter noch nicht benutzen. Queued I/O etc. ist also keinesfalls möglich. Tritt in dem Scheibencode eine exception (Buserror etc.) auf, so läuft das System unter Umständen mit einem Notstart an. Die im Abtastbereich folgenden 15-er Scheiben werden dann nicht mehr exekutiert, ohne daß dies erkennbar sein muß.

Beispiel: Schreibe das Byte `$3C` in den Coupler `$FF60A2` ein.

```
 DC     $AEB1,$BF95,(15*2+1)*37    Signalmarke
_MOVE.B =$3C,$FF60A2               Initialisierung
 RTS                               Zurueck (Nukleus)
```

Die Bearbeitung der 15-er Scheiben erfolgt in der Reihenfolge, in der der Abtaster sie findet. In jedem Fall ist die Bearbeitung der letzten 15-er Scheibe auch die letzte Aufsetzoperation des Nukleus. Danach startet unmittelbar der Taskumschalter mit der höchstprioritierten Autostarttask. (Meistens `#ERDMN`)

Für Operationen, die nur beim Kaltstart ausgeführt werden sollen, ist eine eigene Scheibe (18) vorgesehen.

Header-Text **Scheibe: 16**

Signalmarke: `$AEB1`, `$BF95`, `$04C5` $(16 \cdot 2 + 1) \cdot 37 =$`$04C5`

Die Scheibe gestattet die Ausgabe von ASCII-Text, der in der **RTOS–UH** Kopfzeile erscheint, sofern das System den standardmäßigen ERROR-Dämonen besitzt. Diese Scheibe wird nämlich nicht vom Nukleus bearbeitet, sondern von `#ERDMN`. Die Textzeichen werden nacheinander ausgegeben, ohne daß automatisch Carriage-Returns etc. eingefügt werden. Solche Control-Zeichen müssen also im Text enthalten sein. Der Text erscheint in der Reihenfolge, in der der Scanner die 16-er Scheiben erfaßt.

Aufbau hinter der Signalmarke:

```
DC.B    'ASCII-Text',$FF        $FF ist Endekennung.
```

Beispiel:

```
DC      $AEB1,$BF95,(16*2+1)*37   Signalmarke
DC.B    $0A,$0D,'Walter Meier's RTOS',$FF  Text
```

Scheibe: 17 — Externsymbol definieren

Signalmarke: `$AEB1`, `$BF95`, `$050F` $(17 \cdot 2 + 1) \cdot 37 =$`$050F`

Es ist mit dieser Scheibe möglich, im EPROM oder **RTOS–UH**-Bootbereich eine globale Marke zu setzen. Der Lader von **RTOS–UH** kann damit automatisch z. B. EPROM-residente Unterprogramme an das Nutzerprogramm anbinden.

Wann immer der Lader eine nach dem Bearbeiten der Liste noch unbefriedigte Referenz findet, sucht er mit Hilfe des Abtasters den RAM/Bootbereich nach dieser Scheibe ab. Die Scheibe hat also für den Nukleus selbst keine Bedeutung, da sie nur vom Lader bei Bedarf gesucht wird.

Aufbau hinter der Signalmarke bei kurzen Namen bis 6 Zeichen:

```
       DC.B   'Name6b'       6 Bytes  langer  globaler  Name .
       DC     Adr-$          Relativierte Adresse des Symbols.
```

Aufbau hinter der Signalmarke bei Namen bis zu 24 Zeichen:

```
       DC.L   name-$         Relativer Zeiger 32 bit
       DC     0              Indikator: ist Zeiger
       DC     Adr-$          Relativierte Adresse des Symbols.
       ...                   Anderer Text oder nichts
       ...
 name  DC.B   'Ein_langer_Name,$20,$FF
```

Mit dem `$FF` als Stop-Marke endet der auf eine gerade Anzahl Zeichen aufgefüllte Text. Man kann auch kurze Namen mit der zweiten Konstruktion global definieren. Der PEARL-Compiler benutzt diesen Weg bei der `/*+G*/`-Option.

Fortsetzung **Scheibe: 17**

Jede 17-er Scheibe kann nur genau ein globales Symbol definieren. Existieren mehrere Scheiben mit dem gleichen Symbol, so verwendet der Lader die erste vom Abtaster erfaßte Scheibe.

Beispiel:

```
          DC       $AEB1,$BF95,(17*2+1)*37     Signalmarke 17.
          DC.B     'RANF  '                    Globaler  Name.
          DC       RANF-$           relative Einsprungadresse.
          .......                              Beliebige Daten.
>RANF     PRODEC   ...              zum Beispiel: PEARL-Unter-
          VARF     .....                          Programmkopf
          .....
```

Das Label „`RANF`“ ist jetzt dem Lader so verfügbar, als stünde es in einem der Input-files des Laders. Das Zeichen „`>`“ an der Einsprungstelle hat nichts mit dieser Scheibe zu tun, sondern ermöglicht gleichzeitig auch noch das normale Linken des resultierenden S-Recordfiles. Es ist natürlich durchaus möglich, dort ein anderes Symbol als `RANF` zu benutzen, sinnvoll ist das meist nicht.

Auch die Einbaufunktionen des PEARL-Compilers werden mit Hilfe von 17-er Scheiben angeschlossen, allerdings im Gegensatz zu den normalen Funktionen mit gegen die offene PEARL-Welt geschützten Namen (z. B. `#SSIN, #SSQRT`)

Scheibe: 18 **Kaltstart Initialisierungscode**

Signalmarke: `$AEB1`, `$BF95`, `$0559` $(18 \cdot 2 + 1) \cdot 37 =$`$0559`

Bei der Verwendung von RAM-Modulen (etwa VME-RAM) kommt es oft zur Notwendigkeit, deren Paritätsbits richtig vorzubesetzen, um einen Bus-error beim späteren Schreib/Lesezugriff zu vermeiden. Auch kann eine „private“ Notiz, daß es sich um einen Kaltstart handelt, nützlich sein. Seltener dagegen wird man Peripherie-Bausteine schon an dieser Stelle initialisieren wollen. Um dieses dem Systemprogrammierer zu ermöglichen, sucht der Nukleus vor Beginn aller sonstigen Aufsetzoperationen mit seinem Scanner (Abtaster) nach 18-er Scheiben, um deren Code nacheinander zu exekutieren.

Aufbau hinter der Signalmarke:

Maschinencode	fast beliebig, aber kein Tasking, I/O etc.
...	Alle Register bis auf `A7` frei verfügbar.
RTS	Rückkehr in den Nukleus.

Wichtiger Hinweis

> Das System befindet sich noch in einem sogenannten „Kernelmode“ auf der Supervisorebene und kann daher seinen Taskumschalter noch nicht benutzen. Queued I/O etc. ist also keinesfalls möglich. Tritt in dem Scheibencode eine Exception (Buserror etc.) auf, so läuft das System nicht an.

Beispiel: Lösche das RAM von `$80000` bis `$FFFFF`.

```
          DC     $AEB1,$BF95,(18*2+1)*37    Signalmarke
          LEA    $80000,A1                  Start-adresse
          LEA    $100000,A2                 End-adr.+1
          CLR    D0              Null vorbereiten, denn CLR
*                                 beginnt mit Read-cycle !!!!
LOOP      MOVE.B D0,(A1)+        loeschen durch schreiben
          CMPA.L A2,A1           Bedingung
          BLT.S  LOOP            Schleifenrueecksprung
          RTS                    Rueckkehr in den Nukleus.
```

Die Bearbeitung der 18-er Scheiben erfolgt in der Reihenfolge, in der der Abtaster sie findet. In jedem Fall ist die Bearbeitung der 18-er Scheiben allen anderen Operationen vorgelagert.

Für Operationen, die nur beim Warmstart ausgeführt werden sollen, ist eine eigene Scheibe (15) vorgesehen.

Filehandler Link **Scheibe: 19**

Signalmarke: `$AEB1`, `$BF95`, `$05A3` $(19 \cdot 2 + 1) \cdot 37 =$`$05A3`

Bekanntlich können in **RTOS–UH** gleichzeitig mehrere unterschiedliche Filesysteme (RTOS, DOS, Mac–OS) benutzt werden. Wenn ein Filehandler nach dem Öffnen des Rootblockes (die ersten 1024 Bytes der Partition) feststellt, daß er mit diesen Daten nichts anfangen kann, so sucht er im System nach weiteren Filehandlern, die bereit sind, diesen File zu bearbeiten. Mit Hilfe dieser 19-er Scheibe können nun solche anderen Filehandler aufgespürt und angesprungen werden.

Aufbau hinter der Signalmarke:

```
DC.L    TESTXY-$        Lang relativer Verweis
```

Der Einstieg `TESTXY` hat folgende Parameterschnittstelle:

```
A0  In:   Speicheradresse des eingelesenen Rootblockes.
D0  Out:  Antwort, bei D0=0 keine Akzeptanz.
          Auch im Statusregister ggf EQ oder NE
A0  Out:  Bei pos. Antwort: Filemanager Entry adr.
```

(Leere Seite vor neuem Kapitel)

Kapitel 10: Netzwerkoperationen

Wenn Ihr Rechner mit Netzwerk-Hardware ausgerüstet ist, kann auf Dateien von ebenfalls an dieses Netzwerk angeschlossenen Rechnern mit dem Netzwerk-Filehandler zugegriffen werden. Dazu geben Sie den Namen des Zielrechners, gefolgt von dem Dateinamen, an. Mit dem Namen

```
/ST8/H0/COMMON/FILE1
```

kann auf die Datei „FILE1“ im Unterverzeichnis „COMMON“ auf der Platte „H0“ des Rechners mit dem Namen „ST8“ zugegriffen werden. Sie können alle Befehle verwenden, die für das entsprechende Gerät auf dem Zielrechner erlaubt sind. Die einzige Ausnahme bilden die Befehle zum wahlfreien Zugriff, die über das Netzwerk (noch) nicht abgewickelt werden können. Die FORMAT-Befehle können aus Sicherheitsgründen ebenfalls nicht über das Netzwerk abgesetzt werden.

Weiter ist es möglich, daß Tasks über das Netzwerk direkt Daten austauschen können. Wenn eine Task, die im Rechner ST5 abgearbeitet wird, Daten an eine Task im Rechner ST8 senden möchte, so wendet sie sich an

/ST8/CH*xxxx*

wobei „CH“ anzeigt, daß ein Datenkanal und keine Datei gemeint ist, und „*xxxx*“ der Name des Datenkanals ist. Die Task im Rechner ST8 muß dann von

/ST5/CH*xxxx*

lesen. Da die Daten zur Übertragung über das Netzwerk zu Blöcken von 254 Byte zusammengefaßt werden, ist ein CRhinter solchen Daten notwendig, die sofort übertragen werden sollen.

Weitere besondere Befehle für das Netzwerk sind „MES“ zur Übertragung von Nachrichten auf ein Bedien-Terminal und „IP“ zum Abbau einer hängengebliebenen Verbindung, wenn dies mit RETURN nicht mehr möglich ist.

Das **RTOS**–**UH**-Netzwerksystem kennt besondere Stationen, die etwa zum Anschluß von Druckern, Plottern etc. dienen. Hier reicht die Angabe des Stationsnamens zum Zugriff. Diese Stationen können aber nur einen Auftrag zur Zeit bearbeiten, ist schon ein Auftrag in Bearbeitung, so erhält man die Fehlermeldung „Station zur Zeit belegt“. Eine solche Station könnte z. B. aus einem EPAC bestehen, der nur die Aufgabe hat, den Drucker netzwerkfähig zu machen.

Obwohl der PEARL-Compiler die Netzwerkstationen nicht kennt, kann man im SYSTEM-Teil eines PEARL-Programmes dennoch deren Namen verwenden. Der Lader besorgt beim Laden des Programmes den entsprechenden Anschluß:

```
NETFILE:/ST8/H0/COMMON/FILE <->;
```

oder

```
NETOUT :/ST8/CHxxxx          ->;
```

oder

```
NETIN  :/ST5/CHxxxx         <- ;
```

Für den Zugriff auf einen Datenkanal muß der Filenamen also mit CH, sonst mit dem Gerätenamen anfangen.

Beispiele:

```
P /ST6/H0/PROG1>/ED/PROG1S LO /ST1
```

Übersetzt aus der Datei PROG1 auf der Platte H0 des Rechners ST6 und schickt die Liste zur Druckerstation ST1.

```
DIR /ST6/H0
```

Gibt das Rootverzeichnis der Platte H0 des Rechners ST6 aus.

`DIR /ST6` Gibt alle Gerätebezeichner aus, die auf der Seite der Gegenstation verfügbar sind.

```
FILES /ST7
```

Gibt alle Verbindungen aus, die auf der Station noch offen sind. Wenn /ST7 die eigene Station ist, so kann man sehen, mit welchen Geräten oder Files auf welchen Rechnern noch Verbindungen aktiv sind. Natürlich sieht man mindestens die eigene Standardout Pathlist auf der Gegenstation offen, weil der Befehl ja gerade dorthin schreibt.

Init a Connection **I P**

Syntax: `IP /stationname/pathlist`
`IP -S /stationname/pathlist`

Die durch *stationname* und *pathlist* angegebene Verbindung wird auf Netzwerkebene abgebrochen. Es werden jedoch keine evtl. noch offenen Dateien geschlossen. Der Befehl ist nur anzuwenden, wenn `RETURN` oder `CLOSE` nicht mehr zu dem gewünschten Ergebnis führen.

Stationname muß ein im ausführenden System bekannter Name einer Netzstation sein.

Pathlist ist die exakte und im Zielsystem gültige Pathlist einer offenen Verbindung.

Beispiel:
```
IP /SN17/PN;
IP -S /ST25/H0/TEX/PROJ.TEX
```

Der Parameter `-S` (Superuser) ist erforderlich, wenn die Verbindung von einem anderen Nutzer aufgebaut wurde.

Das System antwortet mit „`... JOB_ABORTED`“ oder „`... FILE NOT FOUND`“– je nach Sachlage.

L O C K — Lock external access

Syntax: `LOCK` *`stationname`* `[`*`stationname`*`] ...`

Mit diesem Befehl kann man sich gegen (unbefugten) Zugriff anderer Netzstationen schützen, dabei aber gleichzeitig noch einen Zugriff auf bestimmte Geräte oder Directories zulassen. Auch eine Verriegelung gegen alle anderen Netzstationen ist möglich.

stationname: Die Elemente der Liste müssen im System bekannte Stationen sein. Setzt man hier den eigenen Namen ein, so werden **allen** anderen Stationen die Zugriffsrechte entzogen. Den eigenen Stationsnamen kann man mit „`OWNST`“ erfragen. Lesen Sie bitte dazu auf Seite 669 nach.

```
LOCK  /SN6 /SN7
```

Hier schützt man sich gegen Zugriffe der Stationen `/SN6` und `/SN7`.

Das System antwortet mit einer Meldung, die den gesamten Lock-Status für das Netz beschreibt, in dem die jeweils adressierte Station der Liste zu finden ist. Die ausgegebene Liste der Buchstaben „`U`“ und „`L`“ ist nach „`DRIVE`“-Nummern geordnet.

Der komplementäre Befehl zu `LOCK` ist „`UNLOCK`“. Er benutzt die gleiche Syntax und antwortet ebenfalls mit einer Statusmeldung.

Durch Laden eines Assemblermodules „NETLOK“ oder Einbindung eines solchen an geeigneter Stelle im EPROM kann man bestimmte Geräte oder Directories des eigenen Rechners trotz Sperre zugänglich halten. Wenn Sie dies nicht wenigstens für Ihre Console tun, so können Sie von der gesperrten Station auch keine Informationen mehr erhalten, weil diese (z. B. beim `DIR`) ja nicht auf Ihr Terminal schreiben darf.

Fortsetzung **L O C K**

Das Assemblermodul hat einen einfachen Aufbau, denn es enthält nur die textliche Bezeichnung der zugelassenen Zugriffspfade. Beim `DIR` und ähnlichen Befehlen adressiert der ferne Rechner das eigene Terminal über eine „`/LD`-Kodierung“. Aus diesem Grund sind entsprechende Einträge vorzusehen.

```
DC.L    0,0                 Modulkopf
DC      $0010               Typ Modul
DC      'NETLOK'            Name
DC.B    'LD/00',$FF         /A1 für DIR
DC.B    'LD/02',$FF         /A2 für DIR
DC.B    'LD/04',$FF         /A3 für DIR
DC.B    'H0/PUB',$FF        /H0/PUB/...
DC.B    'ED/PUB',$FF        /ED/PUB/...
DC.B    'LD/01.00/PUB',$FF /ED/PUB/...
DC.B    $00                 STOP - Marke
END
```

Wie man sieht, werden die Pfadlisten ohne das eröffnende Zeichen „`/`“ abgelegt und durch das Byte `$FF` beendet.

Ein geladenes Modul dieses Namens übersteuert ein im Boot- oder EPROM-Bereich liegendes.

M E S — Message to other System

Syntax: `MES /`*`stationname`* `[-U` *`unumber`*`]`*`nachricht`*

Die angegebene Nachricht erscheint auf dem Terminal

des optional angegebenen Users, sonst beim User 0. Der adressierte User erhält zusätzlich die Stationsnummer der Quelle der Nachricht mitgeteilt.

Stationsname: Er muß eine im System bekannte Zielstation beschreiben. Verwendet man dabei die eigene Stationsbezeichnung, so erfolgt eine Fehlermeldung.

unumber: Diese Ganzzahl ist die laufende Nummer eines auf dem Zielsystem vorhandenen Nutzerarbeitsplatzes (primäre Shell). Der Nutzerplatz muß vorhanden sein, sonst erfolgt eine Fehlermeldung „`... wrong command`“.

Nachricht: Es ist beliebiger druckbarer Text erlaubt. Als Endekennung wird das Semikolon, das Doppelminus („`--`“) oder das Zeilenende benutzt. Achten Sie auf Metazeichen, wenn Sie mit der Shellsprache arbeiten und auf Shellvariable (`$`-Zeichen!), die den Text verändern können.

Wir studieren dazu ein Beispiel. Wenn auf dem Rechner „ST06“ der Befehl

```
MES /ST8 Kommen Sie mit in die Kantine?
```

eingegeben wird, adressiert man das Terminal des Users 0 (Console) am Rechner „ST8“. Dort erscheint folgende Meldung:

```
>> MESSAGE FROM: /ST06.
Kommen Sie mit in die Kantine?
```

Der Absender auf Station 6 erhält eine Mitteilung („`... message received`“) und sieht so, ob die Nachricht angenommen wurde. Wenn auf dem Terminal der Zielstation eine Eingabe hängt, wird die Nachricht erst nach Beendigung der Eingabe sichtbar.

```
MES /SN24 -U 3 Hallo Nutzer 3
```

adressiert entsprechend den Nutzer mit der laufenden Nummer 3.

Ein Text, der mit „`-U`“ oder „`-u`“ beginnt, kann nu dann transferiert werden, wenn vorher ein echter `-u`-Parameter gegeben wurde:

```
MES /SN24 -u1 -ungeheuer ...
```

Eigene Stations-ID feststellen **OWNST**

Syntax: OWNST *stationsname*;

Bei *stationsname* muß ein im System bekannter Name einer

Netzstation des gewünschten Netzes eingegeben werden. Der Befehl antwortet mit dem Namen der eigenen Station.

Eine Station kann gleichzeitig in verschiedene Netze eingebunden sein und darum auch verschiedene Eigennamen haben. Mit diesem Befehl wird der zur Zielstation passende Eigenname im zuständigen Netz ermittelt.

Bei diesem Befehl stört es nicht, wenn man zufällig den Eigennamen der ausführenden Station bei *stationsname* angibt.

Beispiel: `OWNST /SN1`

antwortet möglicherweise mit:

```
>> OWN DEVICE: /SN11 (NETWORK).
```

U N L O C K — Unlock für Netzstation

Syntax: `UNLOCK` *`stationname`*`[`*`stationname`*`] ...`

Mit diesem Befehl können einer oder mehreren Netzstationen Zugriffsrechte auf den eigenen Rechner eingeräumt werden. Es handelt sich um den Gegenbefehl zu `LOCK`, der auf Seite 666 genau beschrieben ist.

stationname: Die Elemente der Liste müssen im System bekannte Stationen sein. Setzt man hier den eigenen Namen ein, so werden **allen** anderen Stationen die Zugriffsrechte erteilt. Den eigenen Stationsnamen kann man mit `OWNST` erfragen. Lesen Sie bitte dazu auf Seite 669 nach.

```
UNLOCK  /SN6 /SN7
```

Hier erlaubt man den Stationen `/SN6` und `/SN7` den Zugriff.

Das System antwortet mit einer Meldung, die den gesamten Lock-Status für das Netz beschreibt, in dem die jeweils adressierte Station der Liste zu finden ist. Die ausgegebene Liste der Buchstaben „`U`“ und „`L`“ ist nach „`DRIVE`“-Nummern geordnet.

`LOCK` und `UNLOCK` benutzen die gleiche Syntax und antworten beide mit einer Statusmeldung.

Kapitel 11: Glossar

Aktivierung Systemdienst, der eine ↑Task mit dem ↑Taskzustand *schlafend* oder *zur Aktivierung eingeplant* in den Zustand *lauffähig* versetzt. Beinhaltet eine Aufnahme der Task in den ↑Dispatcherring, falls die Task nicht eingeplant war.

Aktivierung, gepufferte Versuchte ↑Aktivierung einer Task, die jedoch einen der Zustände *lauffähig*, *laufend* oder *blockiert* (↑Taskzustand) hat. Hinweis für das Betriebssystem, daß Task bei Beendigung erneut zu starten ist. Bis zu 3 Aktivierungen kann RTOS-UH pro Task puffern.

Aliasname Zweiter Directoryeintrag für eine Datei, über den ebenfalls auf die Datei zugegriffen werden kann. Dadurch stehen zwei ↑Lese-/Schreibzeiger zur Verfügung. Der Bedienbefehl zum Anlegen des Aliasnamens lautet `LINK`.

Alphic-Dation ↑Datenstation in PEARL, bei der die Datenübertragung über den angesprochenen Ein-/Ausgabekanal im Verhältnis zu einem Prozessorbefehl sehr lange dauert. Deshalb Betreuung des Kanales durch eine ↑Betreuungstask.

Ausgabe Schreiben von Daten aus dem Rechner heraus. Prinzipiell über ↑Traps möglich. Bei RTOS-UH über ↑Betreuungstasks gelöst, falls die Ausgabe wesentlich länger als ein Maschinenbefehl dauert. Bei ↑Alphic-Dations Absenden eines ↑CEs an die zugehörige Betreuungstask, verbunden mit dem Schreiben der Daten durch die Betreuungstask.

Ausgabe, asynchrone ↑Ausgabe von Daten, bei der die sendende ↑Task nicht die Ausgabe des letzten zu übertragenden Zeichens abwartet, sondern schon weiterlaufen kann, während die ↑Betreuungstask noch schreibt.

Möglich durch die Verwendung von ↑CEs. Die asynchrone Ausgabe ist gegenüber der synchronen Ausgabe schneller, da sich eine Betreuungstask selbst *blockieren* muß, weil das empfangende Ausgabegerät entweder nicht schnell genug ist oder die Übertragung im ↑DMA-Modus geschieht. Während der Blockierung kann die sendende Task schon weiterarbeiten.

Ist das CE im ↑Return-Mode abgeschickt worden, ist sogar eine asynchrone Ausgabe mit (verzögerter) Quittung möglich.

Ausgabe, synchrone ↑Ausgabe von Daten, bei der die sendende ↑Task solange wartet, bis das letzte Zeichen übertragen ist.

Ausnahmebehandlung Betriebssystemdienst, der Prozessorausnahmen (z. B. Zugriff auf nicht existierende Speicherstellen, ungültiger Code, unerlaubte Adresse) einer CPU entgegennimmt. Führt bei RTOS-UH zur Beendigung einer ↑Interrupt-Service-Routine, falls diese die Ausnahme ausgelöst hat. Die Fehlermeldung gibt in diesem Fall der ↑Error-Dämon aus. Besitzt ein ↑Prozeß zweiter Art keinen ↑Exception-Handler, generiert auch hier der Error-Dämon die Meldung und suspendiert die Task. Im anderen Fall wird der Exception-Handler angesprungen. Der betriebssystemeigene gibt die Meldung aus und beendet den Prozeß.

Ausplanung Systemdienst, der alle ↑Einplanungen und ↑gepufferten Aktivierungen einer Task löscht.

Autoclose Erreicht bei einem Leseauftrag der ↑Lese-/Schreibzeiger das Dateiende, wird bei eingeschaltetem Autoclose die Datei sofort geschlossen. Bei der Datenstation `/ED` erfolgt das Schließen nach jedem Lese-/Schreibzugriff, es sei denn, die Datei wurde exklusiv geöffnet.

Basic-Dation ↑Datenstation in PEARL, bei der die Datenübertragung über die betreute Schnittstelle nur wenige Prozessorbefehle dauert (z.B. A/D- und D/A-Wandler). Oft als ↑Trap oder globales Unterprogramm realisiert.

Bedienbefehl Ursprünglich Kommando, das ein Nutzer über eine ↑primäre Shell oder über eine ↑sekundäre Shell Fall a) an das Betriebssystem absetzen kann. Inzwischen sind auch Bedienfehle in ↑Shellskripten möglich. PEARL-Tasks können über eine Einbaufunktion, die die Task kurzfristig in eine Shell verwandelt, Bedienbefehle absetzen.

Bedienbefehl, nachgeladener Mit Hilfe des Befehls `LOAD` geladene ↑Shellerweiterung. Assembler- oder PEARL-codiert.

Bedienbefehl, permanenter ↑Bedienbefehl, der immer im System vorhanden ist und nicht entladen werden kann. Assembler- oder PEARL-codiert.

Bedienbefehl, transienter Bedienbefehl, der in einer Datei gleichen Namens steht und der dadurch aufgerufen wird, daß der Nutzer den Dateinamen inklusive Pfad eingibt. Handelt es sich um einen ↑relativen Pfad, werden nacheinander alle ↑Execution-Directories durchsucht. Selbstentladung nach Beendigung. S-Records mit transienten Bedienbefehlen lassen sich auch mit dem Bedienbefehl `LOAD` nachladen und werden dann zu einer ↑Shellerweiterung.

Betreuungstask ↑Task, die Ein- und Ausgaben über Ein- und Ausgabekanäle

(z.B. serielle und parallele Schnittstelle, Festplatte, Floppy, Netzwerk) des Rechners betreut.

Bourne-Shell An den UNIX-Bourne-Sprachstandard angelehnte ↑sekundäre Shell, die über die Elemente der Bourne-Sprache verfügt.

Bus-Error-Handler ↑Prozeß erster Art, in den die CPU verzweigt, wenn ein Prozeß auf eine nicht existierende Adresse zugreifen möchte (Bus-Error). Der Bus-Error-Handler wird auch von anderen Prozessen erster Art angesprungen, die ↑Ausnahmebehandlungen annehmen (z.B. bei Wrong-Opcode-Error, Wrong-Address-Error).

CE Abkürzung für ↑Communication-Element.

Code, realer Ausführbarer Prozessor-Code.

Code, virtueller Code, der nicht auf dem Zielprozessor ablaufen kann und daher zu emulieren ist. In RTOS-UH betrifft das vor allem die formatierte ↑Ein- und ↑Ausgabe in PEARL-Programmen. Der entsprechende ↑Emulator heißt ↑Hyperprozessor.

Communication Element (CE) Primär Speicherbereich für ein Datenpaket (inklusive Auftragskodierung), das eine ↑Task an eine ↑Betreuungstask sendet, um eine ↑Ein- oder ↑Ausgabe anzustoßen. Kann aber auch an beliebige Tasks gesendet werden, um Daten auszutauschen. Ein CE beinhaltet Sender, Empfänger, Daten und Kommando, was mit den Daten zu geschehen hat.

Compilezeitfehler Fehler, die der Compiler bei der Übersetzung des Hochsprachtextes ausgibt.

Dämon Prozeß 2. Art, der Betriebssystemdienste (im ↑User-Mode!) übernimmt.

Datenstation Datenstruktur in PEARL zur Ein- und Ausgabe über Ein- und Ausgabekanäle. (↑Alphic- ↑Basic-Dation).

Dation ↑Datenstation.

Device-Mnemo Zeichenkette, mit „/“ beginnend, über die ein ↑Bedienbefehl oder eine ↑Alphic-Dation eine ↑Betreuungstask ansprechen kann (z.B. `/H0` zum Ansprechen der ersten Festplatte im Rechner).

Direct-Memory-Access (DMA) Fähigkeit eines Peripheriebausteins, ohne Zuhilfenahme des Prozessors direkt in den Speicher eines Rechners zu schreiben. Der Baustein teilt sich mit dem Prozessor den Datenbus. Der Prozessor wird dadurch zwar etwas langsamer, insgesamt ergeben sich jedoch viel schnellere Datenübertragungsraten. Besonders vorteilhaft

bei ↑Betreuungstasks in einem ↑Multitaskingsystem: Sie *blockiert* sich nach Initialisierung des Datentransfers und bleibt während des gesamten Transfers *blockiert*. Der Peripheriebaustein sendet nach der Übertragung einen Statusinterrupt (↑Hardware-Interrupt) und die dazugehörige ↑Interrupt-Service-Routine kann die Betreuungstask fortsetzen.

Dispatcher ↑Prozeßumschalter.

Dispatcherring Doppelt verketteter Ring, in den alle ↑Tasks, nach ihrer ↑Priorität geordnet, aufgenommen sind, die nicht den ↑Taskzustand *schlafend* besitzen.

DMA Abkürzung für ↑Direct-Memory-Access.

Echtzeitbetriebssystem Betriebssystem, das auf einen ↑Hardware-Interupt innerhalb einer garantierten Maximalzeit reagiert, egal, in welchem Zustand es sich gerade befindet. Oft wegen der vielen Anwendungen, die bei einer Regelungs- oder Steuerungsaufgabe anfallen, als ↑Multitaskingsystem ausgelegt.

Einbaufunktion ↑permanente Ladebibliothek.

Eingabe Lesen von Daten aus der Rechnerperipherie. Prinzipiell über ↑Traps möglich. Bei RTOS-UH über ↑Betreuungstasks gelöst, falls die Eingabe wesentlich länger als ein Machinenbefehl dauert. Bei ↑Alphic-Dations Absenden eines ↑CEs an die zugehörige Betreuungstask, verbunden mit dem Lesen der Daten durch die Betreuungstask.

Eingabe, asynchrone ↑Eingabe, bei der sich die ↑Task, die das Eingabe-↑CE absendet, weiterhin den ↑Taskzustand *laufend* beibehält und nicht auf das Ende der Eingabe wartet. Sinnvoll bei Betrieb im ↑Return-Mode.

Eingabe, synchrone ↑Eingabe, bei der sich die ↑Task, die das Eingabe-↑CE absendet, *blockiert*. Sobald die ↑Betreuungstask das CE an den Auftraggeber mit Daten oder Fehlermeldung zurücksendet, versetzt RTOS-UH die auftraggebende Task in den ↑Taskzustand *lauffähig*. Standardfall für Eingaben.

Einplanung Systemdienst, der dem ↑Scheduler mitteilt, daß eine ↑Task zu einem bestimmten Zeitpunkt bzw. bei einem bestimmten ↑Hardware-Interrupt fortzusetzen oder zu aktivieren ist. Zeitliche Einplanungen können auch zyklisch erfolgen. Dann sind Zykluszeit und Zyklusende ebenfalls angebbar.

Emulator Programm, das einen Prozessor simuliert, der gar nicht im Rechner enthalten ist. Der ↑PC zeigt während der Emulation auf den Emulator, nicht auf den ↑virtuellen Code.

Environment Eine Datenstruktur, die eine ↑Task benötigt, um als ↑Shell arbeiten zu können. Dazu gehören u.a. die Shellnummer, Standardpfade für Ein- und Ausgaben sowie Fehlermeldungen (stdin, stdout und stderr), ↑Working- und ↑Execution-Directories. ↑Primäre Shells besitzen darüber hinaus noch Zeiger auf einen ↑Environment-Block und eine ↑variable Ladebibliothek, die jedoch von ↑sekundären Shells mit der gleichen ↑User-ID mitgenutzt werden.

Environment-Block Speicherbereich, in dem eine ↑primäre Shell ihre ↑Environment-Variablen ablegt.

Environment-Variable Variable einer ↑primären Shell, die ein Nutzer oder ein Programm mit Hilfe von `ENVSET` definieren, ändern oder löschen kann. Zugriff über „$“: `P $Quelle LO NO` compiliert den Inhalt der Variablen „Quelle“.

Error-Dämon ↑Task, die die RTOS-UH-Startmeldung sowie Fehlermeldungen ausgibt, und die für die Aktivierung einer ↑primären Shell oder einer sekundären Shell Fall a) zuständig ist.

Exception-Handler Unterprogramm, das ↑Ausnahmebehandlungen durchführen kann. Hat ein ↑Prozeß zweiter Art eine Ausnahmebehandlung ausgelöst und hat er einen Exception-Handler montiert, kann dieser an Stelle des ↑Error-Dämons die Fehlermeldung ausgeben und den auslösenden Prozeß ggf. suspendieren oder beenden.

Execution-Directory Ordner auf einem Massenspeicher, in dem RTOS-UH ↑transiente Bedienbefehle und ↑Shellskripte sucht, falls ein transienter Bedienbefehl oder ein Shellskript über einen ↑relativen Pfad aufgerufen wird.

Exklusivöffnung Bei Exklusivöffnung einer Datei kann nur die Task auf die Datei zugreifen, die diese geöffnet hat. Damit die Datei auch geschlossen werden kann, falls die Task nicht mehr existiert, darf die der Task zugeordnete ↑primäre Shell bzw. ↑sekundäre Shell Fall a) mittels `RETURN` die Datei schließen. Ein `RETURN -A` darf jede Shell senden.

FIFO (first-in-first-out) Datenübertragungskonzept bei Datenkanälen, bei dem die zuerst gesendeten Daten auch zuerst den Empfänger erreichen. Auch bei ↑named Pipes benutzt.

File-Handler ↑File-Manager.

File-Manager (FM) Programm, das alle hardwareunabhängigen Dienste zur Ein-/Ausgabe auf/von Massenspeichern oder LANs bereitstellt. Der hardwareabhängige Driver stellt, in Zusammenarbeit mit dem FM, die ↑Betreuungstask zur Verfügung. RTOS-UH kennt 5 Manager:

- ED-FM als eine Art RAM-Disk.
- MS-FM zur Verwaltung von MS-DOS formatierten Massenspeichern.
- UH-FM zur Verwaltung von RTOS-UH formatierten Massenspeichern.
- MAC-FM zur Verwaltung von MAC-OS formatierten Massenspeichern.
- NET-FM zum Lesen und Schreiben von Ein-/Ausgabe-Kanälen anderer, lose gekoppelter Rechner.

Hardware-Interrupt Interrupt eines Peripheriebausteines. Man spricht von einem Dateninterrupt, wenn der Baustein mitteilt, daß neue Daten eingetroffen sind oder entgegengenommen werden können, und von einem Statusinterrupt, wenn der Baustein einen Status meldet (z.B. Kein Papier im Drucker, Schreibfehler).

Hyperprozessor Sammlung von Unterprogrammen, die eine PEARL-kodierte Task zwingend benötigt. Die Unterprogramme sind dem Compiler alle bekannt, so daß Spezifikationen im Quelltext entfallen. Der Hyperprozessor enthält auch einen ↑Emulator, um ↑virtuellen Code zu emulieren. Im Emulatorbetrieb ist jedem „virtuellem Befehl“ ein Unterprogramm zugeordnet.

I/O-Dämon ↑Betreuungstask.

Idle ↑Task niedrigster ↑Priorität, die immer *lauffähig* ist und dann läuft, wenn keine Task höherer Priorität *lauffähig* ist.

Interrupt-Service-Routine Prozeß erster Art, in die der Prozessor bei Auftreten des zugehörigen Hardware-Interrupts verzweigt.

Interrupt-Sperre Kurzzeitige Sperre aller Interrupts, um inkonsistente Datenstrukturen dadurch zu verhindern, daß ein anderer Prozeß den ↑PC erhält. In einem ↑Echtzeitbetriebssystem immer nur kurzzeitig erlaubt (wenige Prozessorbefehle).

Kommandointerface ↑Task, die die Eingaben eines Nutzers interpretiert und ausführt.

Ladebibliothek Bibliothek mit globalen Variablen, Prozeduren und Funktionen, die sowohl RTOS-UH als auch der Nutzer zur Verfügung stellen können.

Ladebibliothek, permanente ↑Ladebibliothek von RTOS-UH. Diese Bibliothek ist Betriebssystembestandteil, nicht entladbar, ↑wiedereintrittsfest und gilt für alle Nutzer.

Ladebibliothek, variable Mit Hilfe des Befehls `LIBSET` nutzerdefinierte ↑Ladebiblitothek. Die Bibliothek gilt nur für die User-ID, deren ↑Shell oder ↑Shellsohnprozeß den Befehl abgesetzt hat.

Lader Programm, das ↑S-Records in den Speicher lädt. Die in den S-Records enthaltenen ↑Bedienbefehle und ↑Tasks können nach der Beendigung des Ladens ausgeführt werden.

Lader, transienter Lader für ↑transiente Bedienbefehle. Das Anlaufen des transienten Laders bleibt i.a. dem Nutzer verborgen.

Laufzeitbibliothek ↑Ladebibliothek.

Laufzeitfehler Fehler, die während der Laufzeit eines Prozesses auftreten. Die dazugehörigen Fehlermeldungen senden der ↑Error-Dämon oder der systemeigene ↑Exception-Handler an ↑stderr.

LDN ↑Logical-Dation-Number.

Lese-/Schreibzeiger Zeiger auf das nächste zu lesende / zu schreibende Zeichen einer Datei.

LIFO (Last-In-First-Out) a) Unterbrechungsmechanismus von Prozessen, bei dem der letzte Unterbrechende auch derjenige ist, der sich als erstes beendet. Bei ↑Multitaskingsystemen nur für ↑Prozesse erster Art verwendbar.

b) Datenablageart in einem Speicherbereich (typischerweise einem Stack). Die zuletzt abgelegten Daten werden als erstes wieder abgeholt.

Logical Dation Number (LDN) Betriebssysteminterne Nummer einer Betreuungstask.

Message-Passing Absenden eines ↑CEs an eine beliebige ↑Task. Die Task wird direkt adressiert, ohne daß eine ↑named Pipe oder eine ↑LDN benutzt wird.

Multitasking-System Betriebssystem, in dem mehrere Anwendungen gleichzeitig laufen können. Die einzelnen Anwendungen heißen ↑Tasks.

PC ↑Program-Counter.

Pfad, absoluter Zeichenkette zum Ansprechen einer Datei mit Angabe des ↑Device-Mnemos. Unterordner und Dateinamen können folgen.

Pfad, relativer Zeichenkette zum Ansprechen einer Datei ohne Angabe eines ↑Device-Mnemos. RTOS-UH sucht beim Aufruf von ↑transienten Bedienbefehlen und ↑Shellskripten in den ↑Execution-Directories. Bei relativen Pfaden, die Bedienbefehlen übergeben werden, stellt RTOS-UH das ↑Working-Directory voran.

Pipe, named Benamter Datenkanal nach der ↑FIFO-Methode innerhalb eines Rechners, um Daten von einer ↑Task an eine andere zu senden, ohne gemeinsame Variablen zu verwenden. Die Sendende muß nicht wissen, wer liest, die Lesende nicht, wer schreibt, da die Anbindung über den Namen der Pipe erfolgt. Diesen Dienst stellen in RTOS-UH die Datenstation `/VI` und `/VO` bereit. Eine lesende Task wird solange `blockiert`, bis Daten vorhanden sind. Dadurch ist eine ↑Prozeßsynchronisation gewährleistet.

Preemption Fähigkeit eines Betriebssystems (auch von RTOS-UH), die Ausführung eines ↑Traps abzubrechen, falls ein ↑Hardware-Interrupt einen Lauf des ↑Dispatchers anstoßen will. Der Trap wird später fortgeführt oder neu begonnen.

Priorität Maß für die Wichtigkeit einer ↑Task. Nutzerdefinierte Tasks können eine Priorität von 1 bis 32767 besitzen, wobei die Task mit der Priorität 1 die höchste Nutzerpriorität hat. Negative Prioritäten (mit noch höherer Priorität) sind ↑Dämonen vorbehalten, wobei bei ↑Betreuungstasks aus Sicht der Echtzeiteigenschaften fast immer ↑variable Prioritäten sinnvoll sind.

Priorität, variable ↑Betreuungstasks können mit veränderlicher Priorität ausgestattet werden, so daß auch die ↑Aus- und Eingabe von Daten prioritätsgerecht erfolgt. Sendet eine Task mittels eines ↑CEs Daten an die Betreuungstask, die vorher keine Aufträge mehr zu bearbeiten hatte, gibt RTOS-UH der Betreuungstask eine gegenüber dem Absender um 1 erhöhte ↑Priorität.

Sendet während der Abarbeitung dieses CEs eine höherpriorisierte Task ein weiteres CE an die Betreuungstask, was zu einem Einketten dieses CEs in die ↑Warteschlange der Betreuungstask führt, hebt RTOS-UH die Priorität der Betreuungstask auf die um eins erhöhte Priorität des zweiten Senders an. Dadurch können zwar die neu eingetroffenen Daten nicht sofort abgearbeitet werden, die Abarbeitung der ersten Daten erfolgt dann jedoch mit erhöhter Priorität.

Procedure-Workspace Speicherbereich, der Variablen einer PEARL-Prozedur oder einer PEARL-Funktion sowie Verwaltungsdaten enthält. Im weiteren Sinne von ↑Tasks angeforderter Speicherbereich zur Ablage von taskinternen Daten.

Program-Counter Register einer CPU, das (je nach Prozessortyp) auf die Adresse des gerade abzuarbeitenden oder des nächsten abzuarbeitenden Prozessorbefehles zeigt.

Prozeß erster Art Systemdienste, die keine eigene Task haben. Prozesse erster Art laufen immer im ↑Supervisor-Mode. Zusammenfassender Begriff für ↑Traps, ↑Interrupt-Service-Routinen, ↑Prozeßumschalter und ↑Scheduler.

Prozeß zweiter Art Selbständig laufendes Programm, in einem ↑Multitaskingsystem auch Task genannt. Laufen im Gegensatz zu ↑Prozessen erster Art im ↑User-Mode. Alle Prozesse 2. Art laufen in einem ↑Echtzeitbetriebssystem prioritätsgerecht, d.h. von allen *lauffähigen* Tasks teilt der ↑Prozeßumschalter der höchstpriorisierten den Prozessor zu. Bei RTOS-UH in ↑Betreuungstasks, ↑Dämonen und nutzerdefinierte Tasks aufgeteilt.

Prozeßsynchronisation Vom Programmierer durch ↑Synchronisationsmittel erzwungenene Reihenfolge bei der Abarbeitung von ↑Prozessen 2. Art, die nicht prioritätsgerecht ist. Bei der Nutzung von gemeinsam, veränderlichen Datenstrukturen zwingend erforderlich.

Prozeßumschalter (PU) ↑Prozeß erster Art, der der höchstpriorisierten *lauffähigen* Task (↑Prozeß zweiter Art) den Prozessor zuteilt. Kann nur anlaufen, wenn a) kein anderer Prozeß erster Art läuft (da der PU von allen Prozessen erster Art die niedrigste Priorität hat) und b) sich keine Task kurzfristig in den ↑Supervisor-Mode begeben hat.

Ausnahme von b): Hat eine Task im Supervisor-Mode eine Ausnahmebehandlung ausgelöst (z.B. BUS-Error, Wrong-opcode-error), kann der ↑Bus-Error-Handler diese suspendieren. Anschließend gibt der Prozeßumschalter einer anderen Task prioritätsgerecht den Prozessor.

Random-Access-Mode In diesem Mode erfolgen Lese- und Schreibzugriffe auf eine Datei eines Massenmediums an beliebig vorgebbarer Position. Im Gegensatz zu normalen Zugriffen zeigt der ↑Lese-/Schreibzeiger nur dann auf das Dateiende, wenn über die originäre Dateigröße hinaus geschrieben wird.

Return-Mode ↑Asynchrone Aus- oder Eingabe, bei der nach erfolgter Bearbeitung durch die ↑Betreuungstask das ↑CE in der ↑Warteschlange des Auftraggebers eingekettet wird. Bei ↑asynchroner Ausgabe ist so ein Betrieb mit (verzögerter) Quittung möglich.

Bei ↑asynchroner Eingabe kann eine ↑Task Eingabe-CEs an verschiedene Betreuungstasks senden und zurückkehrende CEs in der Reihenfolge be-

arbeiten, in der sie zum Auftraggeber zurückkehren. Gleichzeitig lassen sich sogar noch CEs anderer Tasks bearbeiten, die mit Hilfe des ↑Message-Passing gesendet wurden.

Runtime-Library ↑Ladebibliothek.

S-Record Ablageform von Compiler, Assembler und Linker für übersetzte Programme, die jedoch nur die unteren sieben Bits des ASCII-Codes verwendet. Der Befehl `LOAD` kann S-Records verwerten und legt den Code binär im Speicher ab.

Scheduler ↑Prozeß erster Art, der alle zeitlichen und interruptgesteuerten Einplanungen von ↑Tasks verwaltet und diese bei Eintreffen des Ereignisses je nach Auftrag fortsetzt oder aktiviert.

Scheibe Nicht ausführbarer Programmcode, der der ↑Selbstkonfiguration beim RESET, der Definition der ↑permanenten Ladebibliothek oder eines ↑Bedienbefehles dient.

Selbstkonfiguration Fähigkeit von RTOS-UH, sich beim Hochlauf an Hand von ↑Scheiben selbst zu konfigurieren, ohne daß das System gelinkt werden muß. Ermöglicht eine einfache Erweiterung von RTOS-UH für eigene Zwecke durch Hinzufügen neuer Scheiben.

Shell ↑Task, die Bedienbefehle abarbeiten kann. Um als Shell arbeiten zu können, benötigt sie gegenüber einer „gewöhnlichen“ Task ein Shellenvironment, auch einfach nur ↑Environment genannt.

Shell, primäre a) ↑Shell, die von RTOS-UH bereits bei der ↑Selbstkonfiguration des Systems eingerichtet wird und nicht entladen werden kann. Sie dient der Kommunikation des Nutzers mit dem Betriebssystem.

b) Shell, die RTOS-UH beim Einloggen über ein Netzwerk mittels `TELNET` oder `RLG` generiert. Sie hat die gleiche Aufgabe wie eine primäre Shell Fall a), terminiert und entlädt sich jedoch beim Ausloggen.

Shell, sekundäre a) ↑Shell, die vom Nutzer mit Hilfe des Bedienbefehls `SHELL` aufgerufen werden kann, um an Stelle der ↑primären Shell CTRL-A /B /C zu bearbeiten. BREAK aktiviert weiterhin die primäre Shell.

b) Folgt in der Aufruferzeile einem Bedienbefehl, für den ein ↑Shellsohnprozeß generiert wird, ein Kommando mittels „`--`“, verwandelt sich der Sohnprozeß nach der Abarbeitung des eigenen Bedienbefehls in eine sekundäre Shell, um die Folgekommandos abzuarbeiten. Nach Abarbeitung aller Kommandos Selbstbeendigung und Selbstentladung aus dem Speicher.

c) Bei einem exekutierten ↑Shellskript handelt es sich ebenfalls um eine sekundäre Shell. Nach Abarbeitung aller Kommandos Selbstbeendigung und Selbstentladung des generierten ↑Shellsohnprozesses aus dem Speicher.

Shell-Subroutine-Package Sammlung von ↑wiedereintrittsfesten Unterprogrammen, die für Shells sehr nützlich ist. Enthält auch den Code zur Interpretation einer Eingabezeile, so daß der Code einer ↑primären Shell sehr kurz ist.

Shellenvironment ↑Environment.

Shellerweiterung Nachladen von ↑S-Records, in dem ↑Bedienbefehle mit Hilfe von ↑Scheiben definiert sind. Nach fehlerfreiem Laden stehen die Bedienbefehle allen Shells zur Verfügung. Nutzern, die einen über eine Shellerweiterung hinzugefügten Befehl aufrufen, bleibt es verborgen, ob der Bedienbefehl permanent im System vorhanden oder nachgeladen ist.

Shellprozeß ↑Task mit allen Eigenschaften einer ↑Shell.

Shellskript Datei, die RTOS-UH-Bedienbefehle und/oder Sprachelemente der Bourne-Sprache beinhaltet. Läßt sich mit Hilfe des Bourne-Shell-Interpreters abarbeiten. Aufruf über Bedienbefehl `EX`, über Dateinamen, falls Datei im ↑Execution-Directory liegt oder über ↑absoluten Pfad.

Shellsohnprozeß Eine ↑Shell kann Kommandos nur sequentiell abarbeiten. Bei vielen Kommandos, die sehr lange dauern, generiert die Shell eine eigene ↑Task, den sogenannten „Sohnprozeß“, die den Befehl abarbeitet (z.B. bei Compiler, Assembler, Linker, Lader, COPY). Dadurch kann die Shell bereits nach Starten des Sohnprozesses weitere Kommandos abarbeiten. Falls der ↑Vaterprozeß auf die Beendigung des Sohn wartet (Bedienbefehl `WAIT`), teilt der Sohn nach Beendigung seiner Aufgaben dem Vater den Fehlerstatus mit. Auf jeden Fall entlädt er sich nach Beendigung aller Aufgaben selbst.

Nur die bei der Interpretation von ↑Shellskripten generierten Sohnprozesse haben von Anfang an ein eigenes ↑Environment und somit Shelleigenschaften (↑sekundäre Shell, Fall b) und c)).

Sohnprozeß ↑Shellsohnprozeß.

stdin absoluter Standardeingabepfad (↑Environment).

stdout absoluter Standardausgabepfad (↑Environment).

stderr absoluter Standardpfad für Fehlermeldungen (↑Environment).

Supervisor-Call (SVC) ↑Trap.

Supervisor-Mode Privilegierter Mode eines Prozessors. Eine ↑Task, die sich im Supervisor-Mode befindet, unterdrückt einen Lauf des ↑Prozeßumschalters. (Ausnahme: Task, die sich selbst kurz in den Supervisor-Mode begeben hat, ruft diesen selbst, um wieder in ↑User-Mode zu gelangen.)

Supervisor-Stack Gemeinsamer Stack aller ↑Prozesse erster Art. Der gemeinsame Stack ist möglich, da sich Prozesse erster Art nur nach dem ↑LIFO-Prinzip unterbrechen.

Synchronisationsmittel Konstrukte, die bei korrekter Benutzung gewährleisten, daß bei veränderlichen, von mindestens zwei ↑Prozessen 2. Art gemeinsam genutzten Datenstrukturen keine Mischdaten entstehen können. Bei gemeinsam genutztem Speicherbereich (shared Memory) stellt RTOS-UH ↑Traps für Semaphore und Bolt-Variablen zur Verfügung. Anderenfalls bieten sich ↑Message-Passing (falls Adressat bekannt ist) und ↑named Pipes (falls Adressat nicht bekannt ist) beim Austauschen von Datenstrukturen zwischen ↑Tasks an.

Systemtrap ↑Trap.

Task ↑Prozeß zweiter Art.

Taskidentifier (TID) Zeiger auf den Anfang eines Taskkopfes. Die einzige eindeutige (und schnelle) Identifizierungsmöglichkeit, da Tasknamen auch doppelt vergeben sein können. Der TID der gerade laufenden Task liegt in einer betriebssysteminternen Speicherzelle, damit RTOS-UH bei einem Supervisor-Call (↑Trap) immer weiß, welcher ↑Prozeß 2. Art den ↑Trap aufgerufen hat.

Taskkontext Alle Speicherzellen und Registerinhalte, die der ↑Prozeßumschalter retten muß, wenn eine Task den ↑Taskzustand *laufend* verliert bzw. die der Prozeßumschalter besetzt, wenn eine Task den Zustand *laufend* bekommt.

Taskkopf Speichersegment zur Definition einer ↑Task. Enthält wichtige Daten, wie z.B. Namen, ↑Taskzustand, ↑Priorität und Adresse des ersten ausführbaren Codes.

Taskwechsel Vorgang, bei dem der ↑Prozeßumschalter den Prozessor einer ↑Task wegnimmt und einer anderen zuteilt. Geschieht, falls eine höherpriorisierte Task *lauffähig* wird oder die momentan laufende Task sich beendet oder selbst *blockiert*.

Taskworkspace Speichersegment, das Platz für die Taskvariablen, den ↑Taskkontext und verwaltungsinterne Daten, die sich auf die Task beziehen, bietet.

Taskzustand Einer der vier Zustände *laufend*, *lauffähig*, *blockiert* und *schlafend*, den eine ↑Task haben kann.

schlafend (dormant): Task ist geladen, ist jedoch in keinerlei Aktivität verwickelt.

laufend (running): Prozessor ist der Task zugeteilt.

lauffähig (runable): Der Dispatcher teilt dieser Task den Prozessor zu, sobald keine höherpriorisierte Task den Zustand *lauffähig* oder *laufend* hat.

blockiert (blocked): Task darf den Prozessor nicht besitzen und wartet auf ein Ereignis, das diese in den Zustand *lauffähig* versetzt. In RTOS-UH aufgeteilt in:

- Wartend auf Beendigung der Ein- /Ausgabe (`I/O?`[1])
- Wartend auf Freiwerden eines Semaphors / einer Boltvariablen (`SEMA`)
- Wartend auf ↑Procedure-Workspace (`PWS?`)
- Wartend auf freiwerdende ↑CEs (Jede Task hat ein eigenes Kontingent, damit eine Task nicht den gesamten freien Speicher allokieren kann!) (`CWS?`)
- Wartend auf ↑Aktivierung an bestimmtem Zeitpunkt oder bei Interrupt (`SCHD`)
- Wartend auf Fortsetzung an bestimmtem Zeitpunkt oder bei Interrupt (`SUSP`)
- Wartend auf eintreffende CEs (Nur bei ↑Betreuungstasks, `SCHD`)
- Unterbrochen (`SUSP`)

Trap „Unterprogramm“ zur Bereitstellung von Betriebssystemdiensten, das i.a. von einem ↑Prozeß zweiter Art aufgerufen wird. Ein Trap ist ein ↑Prozeß erster Art.

Umlenkung Möglichkeit, Ein- und Ausgabe sowie Fehlermeldungen mit anderen Pfaden als stdin, stdout und stderr (↑Environment) durchzuführen.

Umlenkung, permanente Neudefinition von stdin, stdout oder stderr (↑Environment).

[1] In `teletype` geschriebenes Kürzel gibt an, wie die ↑Bedienbefehle `L` oder `LU` den Taskzustand anzeigen.

User-ID Systeminterne Nummer einer ↑primären Shell. Über diese Nummer kann RTOS-UH den ↑Taskidentifier der Shell durch einen Tabellenzugriff herausfinden. Jeder ↑Task in RTOS-UH ist eine User-ID und somit eine primäre Shell zugeordnet. ↑Betreuungstasks nehmen immer die User-ID des Auftraggebers an. ↑Error-Dämon und ↑Betreuungstasks schreiben ihre Fehlermeldungen nach ↑stderr der zugehörigen primären Shell.

User-Mode Standard Mode eines Prozessors und für ↑Prozesse zweiter Art. In diesem Mode ist ein ↑Taskwechsel erlaubt.

User-Stack Jede ↑Task hat einen eigenen Stack, der auch User-Stack heißt, da eine Task im ↑User-Mode läuft. Er liegt gewöhnlich im ↑Taskworkspace, kann aber auch in einem extra angeforderten Bereich liegen. In einem ↑Multitasking-Betriebssystem benötigt jede Task ihren eigenen Stack, da Prozesse zweiter Art nicht nach dem ↑LIFO-Prinzip abgearbeitet werden können.

Vaterprozeß ↑Shellprozeß, die einen ↑Shellsohnprozeß generiert hat. Kann in RTOS-UH entweder auf die Terminierung des Sohnes warten oder sich prioritätsgerecht den Prozessor teilen.

Verschieblichkeit Fähigkeit von RTOS-UH, an jeder beliebigen Stelle im Speicher laufen zu können, da RTOS-UH nur PC-relative Bezüge enthält.

Warteschlange Sammlung von ↑Communication-Elementen, die an eine (↑Betreuungs-)↑Task gesendet worden sind, aber noch nicht abgearbeitet werden konnten (Jede Betreuungstask hat eine eigene Warteschlange). Die Warteschlange ist prioritätsgeordnet, so daß CEs von Absendern mit höherer ↑Priorität bei Eintreffen in die Warteschlange vor denen von Absendern mit niedrigerer Priorität eingeordnet werden. Besitzt die empfangende Task ↑variable Priorität, ist die Priorität der Task mindestens genauso hoch wie die des höchstpriorisierten CEs in der Warteschlange.

Wiedereintrittsfestigkeit Code ist dann wiedereintrittsfest, wenn er von beliebig vielen Tasks gleichzeitig genutzt werden kann, wobei die Reihenfolge des Austritts aus dem Code völlig unabhängig von der des Eintritts ist. Bedeutet zwangsläufig ein Verbot von Adressierungsarten „↑PC-Relativ“ und „Absolut“ bei schreibenden Zugriffen. Beispiel: Der Code aller ↑Shellsohnprozesse, die RTOS-UH bereitstellt, ist wiedereintrittsfest. Dadurch braucht RTOS-UH bei der Generierung eines Sohnes nur ↑Taskkopf und ↑Taskworkspace zu generieren.

Working-Directory Ordner eines Massenspeichers, der Nutzerdateien enthält. Werden Bedienbefehlen relative Dateinamen übergeben, vervollständigen sie i.a. den Pfad durch das Voranstellen des Working-Directories.

[

Stichwortverzeichnis

]