Dynamische Prioritätsvergabe an Tasks in Prozeßrechensystemen

Bernd F. Eichenauer

Dynamische Prioritätsvergabe an Tasks in Prozeßrechensystemen

Von der Universität Stuttgart
zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Abhandlung

vorgelegt von
BERND F. EICHENAUER
geboren in Karlsruhe

Hauptberichter: Prof. Dr.-Ing. R. Lauber
Mitberichter: Prof. Dr.-Ing. A. Lotze

Tag der Einreichung: 20. Juni 1974
Tag der mündlichen Prüfung: 25. November 1975

Kurzfassung

Die Anforderungen an die Sprachelemente für die Parallelprogrammierung in prozeßorientierten Programmiersprachen werden diskutiert. Es wird gezeigt, daß der Aufbau von Prozeßrechner-Programmsystemen aus unabhängig voneinander erstellten Programmoduln (Modultechnik) nicht möglich ist, wenn die Wichtigkeit von Automationsprogrammen bzw. Tasks über Programm- bzw. Prioritätsnummern statisch festgelegt wird. Ein neues Verfahren für die dynamische Bestimmung der Wichtigkeit von Automationsprogrammen bzw. Tasks wird vorgeschlagen. Anhand von Abtastregelsystemen werden die Vorteile des neuen Verfahrens dargestellt.

Vorwort

Herrn Prof. Dr. R. Lauber danke ich herzlich für zahlreiche Diskussionen und wertvolle Hinweise bei der Abfassung dieser Arbeit. Herrn Stocker vom Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart bin ich für eine Diskussion über Abtastregelsysteme dankbar.

Inhaltsverzeichnis

Literaturverzeichnis

[1] Lauber, R.: Prozeßautomatisierung mit Prozeßrechnern, VDI-Bildungswerk, BW 2362.

[2] BICEPS summary manual/BICEPS supervisory control, GE Proc. Comp. Dept., A GET-3539 (1969).

[3] 1800 process supervisory program (PROSPRO/1800), IBM no. H 20-0473-1 (1968)

[4] Bates, Donald G.: PROSPRO/1800, IEEE transcations on industrial electronics and control instrumentation, Vol. IECI-15, no. 2, 70-75 (1968).

[5] Bendix OPTOL Programming System, The Bendix Corporation, Teterboro N. Y. 1968.

[6] ATLAS Abbreviated Test Language for Avionics Systems, ARINC Specification 416-8, Aeronautical Radio Inc. Annapolis 1973.

[7] Metsker, Gene S.: Checkout test language: An interpretative language designed for aerospace checkout tasks, Fall Joint Comp. Conf. 1329-1336 (1968).

[8] PLACE The compiler for the programming language for automatic checkout equipment, Batelle Memorial Institute, Columbus Laboratories, Technical Report AFAPI-TR-68-27 (1968).

[9] Pike, Herbert E.: Process Control Software, Proceedings of the IEEE, Vol. 58, no. 2, 87-97 (1970).

[10] Official Definition of CORAL66, Inter-Establishment Committee on Computer Applications, London 1970.

[11] Callaway, A. A.: A guide to CORAL programming, Royal Aircraft Establishment, Technical Report 70102 (1970).

[12] Mußtopf, G.: Das Programmiersystem POLYP, Angewandte Informatik, Heft 10, 441-448 (1972).

[13] Gohlke, J.; Hotes H.; Mußtopf G.: Real-Time-Programmiersprachen, ESG Elektronik-System GmbH, Bericht Nr. FII1-002, München 1970, teilweise abgedruckt in: SCS-Schriftenreihe, Band 2, Hamburg 1971.

[14] INDAC8 Industrial Data Acquisition and Control Language, Digital equipment corporation, Maynard Mass.

[15] BBC System DP1000, PAS1 Programmieranleitung, BBC Mannheim 1970.

[16] Kussl, V.: Prozeß-Automations-Sprache PAS1 Sprachbeschreibung, BBC System DP1000, BBC Mannheim 1971.

[17] SYSTEME PROCOL T2000, STERIA, Réf. 1 162 220/00 39 00, Le Chesnay.

[18] Timmesfeld, K. H. et. al.: PEARL - Vorschlag für eine Prozeß- und Experimentautomationssprache, Gesellschaft für Kernforschung mbH, PDV-Bericht KFK-PDV 1, Karlsruhe 1973.

[19] Eichenauer, B.: Die Prozeßprogrammiersprache PEARL -Systembeschreibung und E/A Konzept-, Lecture Notes in Economics and Mathematical Systems, Bd. 78, 552-560, Springer 1972.

[20] Rieder, P.: Das Task- und Timing-Konzept von PEARL, ebenda 561-570.

[21] Eichenauer, B.; Haase, V.; Holleczek, P.; Kreuter, K.; Müller, G.: PEARL, eine prozeß- und experimentorientierte Programmiersprache, Angewandte Informatik, Heft 9, 363-372 (1973).

[22] Lauber, R.: Programmiersprachen für Prozeßrechner, Seminar in höherer Automatik der ETH Zürich (1974).

[23] Richards, M.: The Portability of the BCPL Compiler, Software - Practice and Experience, Vol. 1, 135-146 (1971).

[24] Richards, M.: INTCODE - An interpretative machine code for BCPL, The Computer Laboratory, Cambridge 1972.

[25] Waite, W. M.: The mobile programming system: STAGE2, Comm. ACM, Vol. 13, 415-421 (1970).

[26] Poole, P. C.; Waite, W. M.: Portability and Adaptability, Lecture notes in Economics and Mathematical Systems, Bd. 81, 181-277, Springer 1973.

[27] CIMIC/1 Vorläufige Spezifikation, PDV-Entwicklungsnotizen PDV-E15, Gesellschaft für Kernforschung mbH, Karlsruhe 1974.

[28] Coleman, S. S.; Poole, P. C.; Waite, W. M.: The mobile programming system JANUS, Software - Practice and Experience, Vol. 4, 5-23 (1974).

[29] Richards, M.: The BCPL Reference Manual, Technical Memorandum 69/1, Computer Laboratory, Cambridge 1969.

[30] Richards, M.: BCPL: A tool for compiler writing and system programming, Spring Joint Comp. Conf. 557-566 (1969)

[31] Wirth, N.: The programming language Pascal, Acta Informatica, Bd. 1, 35-63 (1971).

[32] Wirth, N.: The programming language Pascal (revised report), Berichte der Fachgruppe Computer-Wissenschaften, Eidgenössische Technische Hochschule, Zürich 1972.

[33] Wirth, N.: Systematisches Programmieren, Teubner Studienbücher Informatik, Bd. 17, Stuttgart 1972.

[34] Dijkstra, E. W.: Co-operating Sequential Processes, in Programming Languages, edited by F. Genuys, Academic Press, London 1968.

[35] Eichenauer, B.: Einige Anmerkungen über die Beziehungen zwischen Prozeßsprache und Betriebssystem, PDV-Mitteilungen, Bd. 1, Gesellschaft für Kernforschung mbH, Karlsruhe 1971.

[36] PL/1-Handbuch (Teil 1 und Teil 2), IBM Betriebssystem 360, IBM Form G79 998-1 und 79 981-1 (1970).

[37] Conway, R. W.; Maxwell, W. L.; Miller, L. W.: Theory of Scheduling, Addison-Wesley 1970.

[38] Lagally, K.: Aufruf von Systemleistungen in einem schichtenweise gegliederten Betriebssystem, Lecture Notes in Economics and Mathematical Systems, Bd. 78, 208-213, Springer 1972.

[39] Nehmer, J.: Dispatcher-Elementarfunktionen für symetrische Mehrprozessor-DV-Systeme, Karlsruhe 1973.

[40] LTPL-Report: Functional Requirements for Language Features for a Procedural Language for Industrial Computers, Purdue Workshop (1971).

[41] IBM Operating System/360 Concepts and Facilities, in Programming Systems and Languages, edited by Saul Rosen, McGraw-Hill 1967.

[42] Baumann, R.; Eichenauer, B.; Hotes, H.; Hofmann, F.; Nehmer, J.; Rüb, W.: Bericht des Arbeitskreises "Funktionelle Beschreibung von Prozeßrechner-Betriebssystemen" des Unterausschusses "Programmiertechnik" der VDI/VDE Sektion "Technik der Prozeßrechner", erscheint demnächst.

[43] Hotes, H.: Digitalrechner in technischen Prozessen, de Gruyter & Co. 1967.

[44] The ALPHA MIMIC Simulator - Loader - Preparer, Control Data Corporation.

[45] The HCM-230 Simulation System, Hughes Aircraft Corporation.

[46] The FOCUS Simulator, Singer - General Precision Inc., Kearfott Division.

[47] Eichenauer, B.: Entwicklungskonzept für ein Programmsystem zur Erstellung und Erzeugung von System-Software für die Prozeßautomation, PDV-Mitteilungen, Bd. 1, Gesellschaft für Kernforschung mbH, Karlsruhe 1971.

[48] Überarbeitung von PEARL zur Erhöhung der Übereinstimmung mit PL/1, Subset-Arbeitskreis beim Projekt PDV, Gesellschaft für Kernforschung mbH, Karlsruhe 1973.

[49] Eichenauer, B.: Beschreibung des LIPDEK-Betriebssystems für den Rechner DDP-516, BBC Mannheim 1969 (interner Bericht).

[50] Eichenauer, B.: Studie über PAS2 (Prozeß-Automationssprache 2), BBC Mannheim 1970 (interner Bericht).

[51] Lauber, R.: Vorlesungen über Prozeßautomatisierung II, Universität Stuttgart, WS 1973/74.

[52] Haberstock, F.: Über die Synthese von Abtastreglern für Regelkreise beliebiger Ordnung, Regelungstechnik 235-239 und 281-286 (1965).

[53] Föllinger, O.: Synthese von Abtastsystemen im Zeitbereich, Regelungstechnik 269-275 (1965).

[54] Leonhard, W.: Diskrete Regelsysteme, BI Hochschultaschenbücher, Bd. 523/523a, Mannheim 1972.

[55] Kuo, S. S.: Numerical Methods and Computers, Addison-Wesley 1965.

[56] Henn, R.; Lehnhoff, S.: Strategien zur pseudo-kollateralen Verarbeitung von Programmen unter Berücksichtigung vorgegebener Antwortzeiten, Technische Universität München, Abteilung Mathematik, Gruppe Informatik, Bericht Nr. 7307 (1973).

Verzeichnis der Abkürzungen

Δt_{Amax}	Zugelassene Ausführungszeit für einen Ablaufweg im Segment einer Task.
Δt_{Lmax}	Größtmögliche Laufzeit für einen Ablaufweg im Segment einer Task.
Δt_{Rest}	Restzeit, d. i. die zu einem Zeitpunkt t noch verbleibende Zeitspanne bis zum Ablauf der zuletzt vorgegebenen Ablaufzeit.
Δt_w	Wandlungszeit eines Eingabewerks.
$\Delta \tau_i$	Auflösungsdauer für den Zeitbereich i.
P_s	absolute Prioritätsnummer (Systempriorität).
P_r	relative Prioritätsnummer.
s	Sprungantwort der Strecke.
t_{Start}	Startzeitpunkt für eine Task (Zustandswechsel: "bekannt" → "ablaufbereit").
T	Abtastzeit.
T_s	Obere Schranke für die Verschiebung von Abtastzeitpunkten.
TKB	Task-Kontroll-Block.
v	Regelabweichung.
w	Führungsgröße.
x	Ausgangsgröße.
y	diskrete Stellgröße.
$\underline{y}$	treppenförmige Stellgröße.
ZAIZi	Zeiger auf aktuelles Intervall im Zeitbereich i.
ZHBP	Zeiger auf höchste besetzte Priorität.
ZKBR	Zeiger auf kürzeste besetzte Restzeit.
ZTZA	Zeiger auf Task im Zustand "ablaufend".

1. Einführung.

1.1 Prozeßprogrammierung und höhere Programmiersprachen.

In den letzten Jahren haben sich Prozeßrechensysteme zu universell einsetzbaren Rechensystemen entwickelt, die wegen ihrer Anpassbarkeit an unterschiedlichste Automatisierungsfunktionen [1] die Entwicklung spezieller Schaltkreise in zunehmendem Maße überflüssig machen. Bis heute ist es jedoch dem Ingenieur in der Regel nicht möglich, Prozeßrechensysteme ähnlich einfach zu handhaben und einzusetzen, wie er dies bei Verwendung konventioneller Geräte zur Automatisierung technischer Prozesse gewohnt ist.

Die derzeitigen Schwierigkeiten beim Einsatz von Prozeßrechensystemen sind ausschließlich darauf zurückzuführen, daß für die meisten Prozeßrechnertypen bislang keine Hilfsmittel zur problemangemessenen Erstellung von Automationsprogrammen bereitgestellt wurden. Bis heute werden bei der Programmierung von Prozeßrechensystemen die Angaben in einer Automatisierungsfunktion zunächst in eine vom jeweiligen Rechensystem abhängige Form überführt und dann maschinennah codiert.

Diese Vorgehensweise ist mit folgenden Nachteilen verbunden:

1. Der Zeit- und Kostenaufwand für die Erstellung von Prozeßautomationsprogrammen ist, verglichen mit dem Aufwand, der für die Erstellung technisch-wissenschaftlicher und kommerzieller Programme ähnlicher Komplexität erforderlich ist, unangemessen groß.
2. Für die Erstellung von Automationsprogrammen ist die genaue Kenntnis der Hardware und System-Software des jeweils eingesetzten Prozeßrechensystems erforderlich.
3. Der unmittelbare Zusammenhang zwischen Automatisierungsfunktion und Automationsprogramm geht verloren. Maschinennah codierte Automationsprogramme sind dokumentations- und änderungsunfreundlich.
4. Wiederverwendbare Programmpakete (z. B. für die Laborautomation) sind nicht in rechnerunabhängiger Form erstellbar.

Die beiden zuerst aufgeführten Nachteile bedingen den heute üblichen Einsatz hochspezialisierter Ingenieure und Programmierer bei der Erstellung von Automationsprogrammen. Nur wenn einerseits das ingenieurmäßige Verständnis für die Angaben in einer Automatisierungsfunktion vorhanden ist, andererseits durch längeren Umgang mit einem bestimmten Prozeßrechnertyp und dem eingesetzten Prozeßrechner-Gerätesystem die Erfahrung erworben wurde, wie diese Angaben in die rechnernahe Form umzusetzen sind, kann die Erstellung eines Automationsprogramms in erträglicher Zeit ausgeführt werden.

Der unter Punkt 3 angeführte Nachteil macht sich besonders bei der Pflege und Weiterentwicklung von Prozeßautomationsprogrammen bemerkbar. Die Praxis zeigt, daß häufig Änderungen an und Erweiterungen von Automationsprogrammen erforderlich sind. Die Behebung von Fehlern ist beim Einfahren eines Prozeßautomationssystems an der Tagesordnung. In angemessener Zeit können die jeweils erforderlichen Änderungen nur von Spezialisten ausgeführt werden, die mit der Erstellung des ursprünglichen Automationsprogramms befasst waren.

Schließlich verhindert der zuletzt angegebene Nachteil die Weiterverwendung von Erfahrungen und von Programmoduln, wenn verschiedene Typen von Prozeßrechnern und/oder von Prozeßrechner-Gerätesystemen eingesetzt werden.

Schon zu Beginn des vergangenen Jahrzehnts erkannte man, daß eine Verbesserung der Situation nur durch Anhebung des Niveaus erreicht werden kann, auf dem die Programmierung von Prozeßrechensystemen ausgeführt wird. In Analogie zu der Entwicklung auf dem Gerätesektor, wo eine funktionsorientierte Beschreibung verwendet wird und der Anwender über den Aufbau und die Wirkungsweise eines Geräts i. a. nicht Bescheid wissen muß (schwarzer Kasten), wurden funktionsorientierte Programmpakete entwickelt, deren Arbeitsweise mittels weniger Steuerparameter festgelegt werden kann (z. B. [2,3,4]).

Damit waren die oben unter Punkt 1 bis 3 genannten Nachteile, zumindest für einige Teilgebiete der Prozeßautomatisierung, zeitweise aufhebbar.

Es zeigte sich jedoch sehr bald, daß Programmpakete zwar für die Lösung bestimmter Problemkreise recht bequem aber zu wenig flexibel einsetzbar waren. Da sich das Einsatzfeld von Prozeßrechensystemen ständig erweiterte, konnte häufig schon kurze Zeit nach der Fertigstellung eines Pakets ein Teil des Anwendungsgebiets, für das ein Paket entwickelt worden war, nicht mehr abgedeckt werden.

Immerhin haben die mit Programmpaketen gewonnenen Erfahrungen zu der Entwicklung einiger interessanter anwendungsspezifischer Programmiersprachen (z. B. [5,6,7,8,9]) geführt, die für bestimmte Teilgebiete der Prozeßautomatisierung ein bequemes und hinreichendes Hilfsmittel darstellen. So wird z. B. die Prüfsprache STANDARD ATLAS [6] (vergl. auch Abschnitt 1.2) häufig als Hilfsmittel zur Dokumentation von seriell ausführbaren Prüfvorschiften, d. h. von Automatisierungsfunktionen für Folgeprozesse, eingesetzt, obwohl aus weiter unten angegebenen Gründen für die meisten Prozeßrechner noch keine Übersetzer verfügbar sind. Die Übersetzung in den Code einer Zielmaschine wird normalerweise von Programmierern erledigt.

Nachdem man erkannt hatte, daß infolge der Evolution der Prozeßautomatisierung der Umfang anwendungsspezifischer Sprachelemente und der Aufwand für deren Implementierung (Übersetzer und Organisationsprogramme) ständig zunehmen würde, lag es nahe, nach dem Vorbild der technisch-wissenschaftlichen und kommerziellen Programmiersprachen, die ja das jeweilige Anwendungsfeld weitgehend abdecken, ein "mittleres" Sprachniveau anzustreben. Zur Zeit gibt es hinsichtlich des Umfangs der vorzusehenden Sprachmittel zwei unterschiedliche Meinungen, die an die Diskrepanz zwischen ALGOL60 und FORTRAN in den sechziger Jahren erinnern.

Wie bei ALGOL60 wurde beispielsweise bei der Entwicklung von CORAL66 [10,11] , einer ALGOL-Erweiterung, und von PCLYP [12,13] davon ausgegangen, daß wegen des großen Einsatzspektrums einer Prozeßprogrammiersprache und wegen der schon entwickelten Betriebssysteme für vorhandene Prozeßrechner die Programmiersprache selbst keine Realzeit-Sprachelemente enthalten darf. Diese werden entweder vom Benutzer unter Verwendung algorithmischer Sprachelemente realisiert oder durch Aufrufe der jeweils vorhandenen Betriebssystemfunktionen ersetzt.

Es ist unmittelbar einleuchtend, daß auf diese Weise gegenüber den oben erwähnten Programmpaketen nur die größere Flexibilität und bei geeigneter Implementierung die Rechnerunabhängigkeit des Formulierhilfsmittels gewonnen wird. Das Hilfsmittel selbst hat mit dem Gebiet "Prozeßautomatisierung" nicht zu tun, sondern erlaubt dem geübten Programmierer lediglich eine im Vergleich zur Assemblerprogrammierung schnellere Formulierung von Automationsprogrammen, wobei insbesondere die für den Ingenieur relevanten Nachteile 2 bis 4 kaum gemildert werden.

Dagegen wurde bei der Entwicklung der sog. prozeßorientierten Programmiersprachen (z. B. INDAC8 [14], PAS1 [15,16], PROCOL [17], PEARL [18,19,20, 21,22]) versucht, dem Ingenieur ein möglichst bequem handhabbares Hilfsmittel in die Hand zu geben, das in angemessener Zeit erlernbar ist und die zur Formulierung von Prozeßautomationsprogrammen erforderlichen Ausdrucksmittel enthält. Bei der Entwicklung von prozeßorientierten Programmiersprachen geht man davon aus, daß sich die zur Formulierung von Automatisierungsaufgaben erforderlichen Ausdrucksmittel, ähnlich wie dies bei algorithmischen Ausdrucksmitteln möglich ist, auf eine beschränkte Anzahl grundlegenden Operandentypen und Operationen zur Handhabung und Verknüpfung dieser Operandentypen zurückführen lassen. So ist z. B. der Datentyp "Dauer" für die Prozeßprogrammierung fundamental und sollte deshalb in einer Prozeßprogrammiersprache vorgesehen werden. Eine wichtige Grundoperation, für die Dauerangaben erforderlich sind, ist die Operation zur zeitweisen Unterbrechung des Programmablaufs.

Da es aufgrund der geschilderten Erfahrungen mit Programmpaketen und anwendungsspezifischen Programmiersprachen nicht wahrscheinlich ist, daß in absehbarer Zeit eine abgeschlossene anwendungsspezifische Programmiersprache für das Gesamtgebiet "Prozeßautomatisierung" definiert werden kann, stellen die prozeßorientierten Programmiersprachen überall dort ein optimales Hilfsmittel dar, wo die bisherigen anwendungsspezifischen Programmiersprachen nicht hinreichen. In der Tat sind Automatisierungsfunktionen bei Einsatz prozeßorientierter Programmiersprachen ohne Überführung in eine rechnerspezifische Form programmierbar; die Programmiersprachen selbst können zur Darstellung und Dokumentation von Automatisierungsfunktionen eingesetzt werden. Eine ins Detail gehende Kenntnis der Hardware-Eigenschaften und der System-Software von Prozeßrechensystemen ist nicht mehr erforderlich, weil die Einbettung eines Automationsprogramms in ein Rechensystem durch Dienstprogramme automatisch erledigt werden kann. Wegen des erhöhten Dokumentationswerts von Automationsprogrammen kann die Programmpflege auch Ingenieuren übertragen werden, die an der Erstellung des ursprünglichen Automationsprogramms nicht teilgenommen haben. Schließlich wäre nach der Standardisierung einer prozeßorientierten Programmiersprache die weitgehende Portabilität von Automationsprogrammen gesichert.

Obwohl diese Vorteile die Entwicklung und Standardisierung einer prozeßorientierten Programmiersprache wünschenswert machen, wurden solche Programmiersprachen erst in den letzten Jahren definiert oder befinden sich, wie PEARL [18], noch in Entwicklung. Dafür gibt es im wesentlichen die folgenden beiden Gründe:

1. Wahl der Ausdrucksmittel

   Neben den aus technisch-wissenschaftlichen und kommerziellen Programmiersprachen bekannten Sprachelementen zur Beschreibung algorithmischer Zusammenhänge und für die sog. Standard-Ein/Ausgabe werden in einer prozeßorientierten Programmiersprache Ausdrucksmittel benötigt,

   - mit denen die Aufteilung und der Ablauf von Automationsprogrammen an die verschiedenen parallel zueinander und gewöhnlich mit unterschiedlicher Geschwindigkeit ablaufenden Vorgänge in technischen Prozessen angepasst werden kann (Parallelprogrammierung) und
   - die es ermöglichen, die Kommunikation zwischen Prozeßrechensystem und technischen Prozessen trotz der Vielzahl unterschiedlicher Prozeßrechner-Gerätesysteme einheitlich zu beschreiben (Prozeß-Ein/Ausgabe).

   Die Diskussion darüber, welche Sprachmittel in einer prozeßorientierten Programmiersprache erforderlich sind und wie diese Sprachmittel mit den aus konventionellen Programmiersprachen bekannten Ausdrucksmitteln

vereinheitlicht werden können, ist bis heute nicht abgeschlossen. Erst in den letzten drei Jahren scheinen, insbesondere durch die Ergebnisse der PEARL-Definition, Prinzipien für den Aufbau einer allgemeinen prozeßorientierten Programmiersprache gefunden worden zu sein.

Die Beantwortung zahlreicher Fragestellungen steht indessen noch aus. So ist es bisher nicht gelungen, ein einheitliches Konzept für die Ein/Ausgabe zu entwerfen, das sowohl die Standard- wie auch die Prozeß-Ein/Ausgabe umfasst. Die Ausdrucksmittel für die Parallelprogrammierung sind, wie in dieser Arbeit gezeigt wird, teilweise noch nicht auf die Bedürfnisse der Prozeßrechentechnik zugeschnitten.

2. Implementierung von prozeßorientierten Programmiersprachen.

Selbst wenn man voraussetzt, daß die Definition einer allgemeinen prozeßorientierten Programmiersprache gelungen sei, so bleibt die Frage, wie das so gewonnene Hilfsmittel für die zahlreichen Prozeßrechnertypen mit angemessenem Aufwand bereitgestellt werden kann. Da bei Einsatz der z. Z. üblichen Implementierungsverfahren System-Software speziell für jeweils ein Rechnermodell bzw. eine Rechnerfamilie erstellt wird, ist nicht zu erwarten, daß insbesondere die kleineren Hersteller von Prozeßrechnern das Risiko hoher Entwicklungskosten für spezielle System-Software bei der Einführung eines Modells tragen können. Die oben angedeutete Vereinfachung der Prozeßprogrammierung wird daher nur erreichbar sein, wenn es gelingt, die Implementierung von prozeßorientierten Programmiersprachen möglichst weitgehend ohne Bezugnahme auf spezielle Zielrechner durchzuführen.

Die Methoden, die in den vergangenen Jahren bei der Erstellung portabler System-Software eingesetzt wurden, basieren fast ausschließlich auf dem Konzept abstrakter Rechenautomaten [23,24,25,26,27,28]. Man versucht dabei, die Struktur und den Befehlsvorrat eines gedachten Rechners so zu wählen, daß für diesen Rechner erstellte Programme mit einfachen Codegeneratoren in möglichst effizienten Code eines Spektrums realer Rechenautomaten übersetzt werden können. Bei der Erstellung von System-Software kann man sich einer System-Programmiersprache (z. B. CORAL66 [10,11], BCPL [29,30], PASCAL [31,32,33]) und eines Compilers für die Systemsprache, der Code für die abstrakte Maschine absetzt, bedienen. Wird der Compiler in der Systemsprache selbst geschrieben, so bedarf es nur eines einmal zu erstellenden Lotsencompilers für eine reale Maschine, um den Compiler und alle in der Systemsprache erstellten Programme für die realen Rechenautomaten verfügbar zu machen, für die Codegeneratoren erstellt werden (Bootstrapping).

Während die beschriebene Methodik bisher erfolgreich bei der Erstellung portabler Systemprogramme eingesetzt wurde, die nur einfache Ein/Ausgabeanweisungen zur Bedienung von Geräten des Standardperipherie voraussetzen und bei deren Ausführung keine Zeitbedingungen einzuhalten sind, ist ihre Verwendbarkeit bei der Entwicklung portabler Prozeßprogrammiersysteme nicht gesichert. Da der Wechsel des Prozeßrechnertyps in der Regel auch einen Wechsel des Prozeßrechner-Gerätesystems mit sich bringt, muß ein portabler Übersetzer für eine Prozeßprogrammiersprache an ein vorgebbares Gerätesystem und an unterschiedlich ausgelegte Gerätesysteme adaptierbar sein. Weiter sind die Anweisungen für die Ein/Ausgabe und die Parallelprogrammierung nur unter Verwendung organisatorischer Hilfsdienste (Betriebssystemfunktionen) realisierbar. Daher umfasst die Implementierung einer prozeßorientierten oder anwendungsspezifischen Programmiersprache sowohl die Erstellung eines Übersetzers als auch die eines sprachabhängigen Realzeit-Betriebssystems.

Durch Erweiterung der Übersetzungsverfahren und des Befehlsvorrats der bisher eingesetzten abstrakten Maschinen (Zwischensprachen) ist es wahrscheinlich möglich, portable und an das jeweilige Prozeßrechensystem adaptierbare Compiler für Prozeßsprachen zu erstellen (vergl. z. B. [27]). Jedoch ist es bisher nicht gelungen, die Struktur und den Befehlsvorrat eines gedachten Rechners so zu wählen, daß für ein hinreichend großes Spektrum realer Rechnertypen sowohl in Laufzeit als auch in Speicherbedarf ausreichend effiziente Betriebssysteme mit vernünftigem Adaptierungsaufwand generierbar sind. Bis heute ist es deshalb üblich, die jeweils erforderlichen Betriebssystemfunktionen im Assembler des Zielrechners zu erstellen, wozu in der Regel mehrere Mannjahre erforderlich sind.

Eine wichtige Aufgabe der Prozeßdatenverarbeitung wird daher die Entwicklung neuer Methoden sein, die es gestatten, Prozeßprogrammiersysteme möglichst weitgehend portabel zu gestalten. Nur dadurch wird es möglich, die in prozeßorientierten und anwendungsspezifischen Programmiersprachen vorgeschlagenen Sprachmittel anhand eines hinreichend großen Spektrums unterschiedlicher Automatisierungsaufgaben zu erproben. Erst nach der Entwicklung portabler Prozeßprogrammiersysteme werden Prozeßrechensysteme ähnlich einfach handhabbar und einsetzbar sein, wie dies heute schon bei kommerziellen Rechensystemen der Fall ist.

1.2 Zum Niveau prozeßorientierter Programmiersprachen.

Bei der Diskussion der Möglichkeiten zur Vereinfachung der Prozeßprogrammierung wurde in Abschnitt 1.1 festgestellt, daß mit einer mittleren "problemnahen" Sprachebene sowohl die Handhabbarkeit der Sprachmittel durch den Anwender gewährleistet werden kann, als auch die zur Darstellung eines großen Spektrums von Automatisierungsfunktionen erforderlichen Ausdrucksmittel bereitgestellt werden können. Im folgenden sollen die Unterschiede zwischen den verschiedenen Ebenen, auf denen die Formulierung von Prozeßautomationsprogrammen möglich ist, erläutert werden. Da es bislang kein Modell zur Beschreibung des Begriffs "Sprachebene" gibt, werden die Unterschiede anhand eines Beispiels aus dem Gebiet der rechnergeführten Geräteprüfung (sog. GO/NOGO-Test) erläutert.

Bei der Lösung von Prüfaufgaben sind häufig mehrere Signale aufeinanderfolgend an ein Prüfobjekt anzulegen, wobei zwischen dem Anlegen zweier Signale eine vorgegebene Zeitspanne verstreichen muß. Nach dem Anlegen eines Signals soll jedoch sofort untersucht werden, ob das Prüfobjekt die jeweils gewünschte Reaktion aufzeigt. Ist dies der Fall, so wird der Prüfvorgang fortgesetzt, andernfalls abgebrochen.

Um diese Problemstellung einfach darstellen zu können, wird beispielsweise in der anwendungsspezifischen Prüfsprache STANDARD ATLAS [6] eine Anweisung geboten, durch die festgelegt werden kann, nach welcher Mindestdauer ein ATLAS-Programm mit einer bestimmten Anweisung fortgesetzt werden darf.

In Abb. 1.1 ist ein Ausschnitt aus einem ATLAS-Prüfprogramm dargestellt, in dem die beschriebene Problemstellung auftritt. Durch die Anweisung mit

```
                    .
                    .
                    .
100010 APPLY,       AC SIGNAL, VOLTAGE 70 V, FREQ 400 HZ,
                    CNX HI PIN-9   LO PIN-27 §
    20 WAIT FOR,    15 SEC BEFORE STEP 100070 §
    30 VERIFY,      (VOLTAGE), DC SIGNAL, GT 350 MV,
                    CNX HI PIN-12  LO EARTH §
    40 GO TO,       STEP 100070 IF GO §
    50 INDICATE,    MESSAGE, ERROR STEP 100030 §
    60 FINISH §
B§
    70 APPLY,       PULSED AC, ...
```

Abb. 1.1 Beispiel für die Synchronisation des Programmablaufs mit Vorgängen im Prüfobjekt (STANDARD ATLAS).

Anweisungsnummer 100010 wird eine Wechselspannung von 70 Volt mit Frequenz 400 Hz an die Prüflingsklemmen PIN-9 und PIN-27 angelegt. Da das Aufwärmen des Prüflings abgewartet werden muß, darf frühestens nach 15 Sekunden die Anweisung 100070 (die ersten vier Ziffern einer Anweisungsnummer, sog. Testnummer, dürfen weggelassen werden) ausgeführt werden, mit der ein weiteres Signal an das Prüfobjekt angelegt wird.

Um das zu gewährleisten, wird in Testschritt 20 eine WAIT FOR-Anweisung ausgeführt. Danach werden die Testschritte 30, 40 und eventuell 50, 60 ausgeführt, durch die eine Gleichspannung gemessen, überprüft und eventuell nach einer Fehlermeldung an den Operateur der Programmablauf abgebrochen wird. Durch die WAIT FOR-Anweisung wird das Betriebssystem beauftragt, den Ablauf des ATLAS-Programms zu unterbrechen, falls die Anweisung 100070 vor Verstreichen von 15 Sekunden erreicht wird und das Programm nach Ablauf dieser Zeitspanne fortzusetzen.

Die beschriebene Synchronisation des Programmablaufs mit Abläufen in einem Prüfobjekt kann relativ einfach unter Verwendung grundlegender Synchronisationsmechanismen dargestellt werden, die in prozeßorientierten Programmiersprachen vorzusehen sind. Im folgenden werden dazu die von Dijstra [34] eingeführten Semaphoreoperationen verwendet.

Nach Dijkstra kann die Synchronisation von parallelen Abläufen wie folgt beschrieben werden: Vorgegeben sei eine Semaphorevariable, d. h. eine Größe, die positive ganzzahlige Werte anzunehmen vermag. Wird auf die Semaphorevariable eine REQUEST-Anweisung ausgeführt, so soll wie folgt verfahren werden:

a) Falls der Wert der Semaphorevariablen bei der Ausführung der REQUEST-Anweisung größer als Null ist, so wird die Semaphorevariable um 1 erniedrigt.
b) Ist der Wert der Semaphorevariablen bei der Ausführung der REQUEST-Anweisung gleich Null, so wird der Ablauf unterbrochen. Er wird fortgesetzt, sobald die REQUEST-Anweisung gemäß Punkt a) ausführbar wird.

Mit einer RELEASE-Anweisung kann die Fortsetzung von Abläufen, die infolge Nichtdurchführbarkeit von REQUEST-Anweisungen unterbrochen wurden, ausgelöst werden. Durch eine RELEASE-Anweisung auf eine Semaphorevariable wird deren Wert um 1 erhöht.

Unter Verwendung von Semaphoreoperationen kann die anwendungsspezifische WAIT FOR-Anweisung auf "mittlerem" Sprachniveau z. B. wie in Abb. 1.2 dargestellt werden. Darin wurden, der Einfachheit halber, die Ein/Ausgabeanweisungen und die Anweisungen für die Kontrolle der Gleichspannung weggelassen.

```
DECLARE SYNCHRO SEMAPHORE INITIAL(Ø);
                    .
                    .
                    .
AFTER 15 SEC RELEASE SYNCHRO;
                    .
                    .
                    .
REQUEST SYNCHRO;
```

Abb. 1.2 Darstellung der WAIT FOR-Anweisung auf einer mittleren Sprachebene.

Durch die Anweisung "AFTER 15 SEC RELEASE SYNCHRO" wird das Betriebssystem angewiesen, 15 Sekunden nach Erreichen dieser Anweisung eine RELEASE-Anweisung auf die Semaphorevariable SYNCHRO auszuführen. Durch die REQUEST-Anweisung, die an geeigneter Stelle in die Programmniederschrift einzufügen ist, wird der Programmablauf unterbrochen, falls sie vor Ablauf von 15 Sekunden erreicht wird.

Die beiden beschriebenen Ebenen unterscheiden sich von noch tiefer liegenden Programmierebenen dadurch, daß die Programmierung unter Verwendung genormter anlagenunabhängiger Sprachelemente möglich ist. Dagegen werden in den früher zitierten niederen Programmiersprachen oder in Assemblersprachen die zur Organisation von Parallelprogrammen und für die Ein/Ausgabe erforderlichen Maßnahmen unter Verwendung der jeweils vorhandenen Betriebssystemfunktionen, d. h. abhängig von dem jeweils eingesetzten Rechensystem, gelöst. Je nach Komfort des Betriebssystems kann dabei eine größere Anzahl von Anweisungen oder Befehlen zur Darstellung einer Anweisung der anwendungsspezifischen oder mittleren Sprachebene erforderlich sein.

Aus dem Beispiel lässt sich entnehmen, was inhaltlich mit dem Begriff "Sprachebene" gemeint ist. Offenbar hat man eine Programmiersprache in eine um so höhere Ebene einzuordnen, je spezifischer die Sprachmittel auf bestimmte Verwendungszwecke hin zugeschnitten sind oder je weniger Information man über die Umgebung einer Anweisung benötigt, um den Zweck der Anweisung zu verstehen. Strebt man einerseits die problemangemessene Programmierung von Automatisierungsfunktionen an und will andererseits mit vernünftigem Implementierungsaufwand ein großes Spektrum von Automatisierungsaufgaben überdecken, so liegt die Verwendung einer mittleren prozeßorientierten Sprachebene nahe.

1.3 Parallelprogrammierung.

Prozeßrechensysteme werden nur relativ selten zur Automatisierung eines einzigen sequentiell ablaufenden technischen Prozesses (Folgeprozeß) eingesetzt. Da die Automatisierung eines technischen Prozesses gewöhnlich nicht die kontinuierliche Verfolgung und Beeinflussung des Prozeßablaufs erfordert und die Arbeitsgeschwindigkeit moderner Digitalrechensysteme hinreichend groß ist, werden Prozeßrechensysteme weitaus häufiger zur Automatisierung verschiedener parallel zueinander ablaufender technischer Prozesse und/oder von parallel zueinander ablaufenden Vorgängen innerhalb einzelner technischer Prozesse (Teilprozesse) eingesetzt.

Die gemeinsame Verwendung der Betriebsmittel eines Prozessrechensystems (Zentraleinheiten, Analogeingaben, Digitalausgaben, Schreibmaschinen, u. s. w.) führt bei der Erstellung von Automationsprogrammen zu neuartigen Problemstellungen [1] . Während in herkömmlichen Automationssystemen die zur Realisierung von Automatisierungsfunktionen eingesetzten Geräte weitgehend autonom arbeiten, sind in Prozeßrechensystemen Automationsprogramme wegen der gemeinsamen Benutzung der Betriebsmittel voneinander abhängig. Benötigen mehrere gleichzeitig auszuführende Automationsprogramme für ihren Ablauf dasselbe Betriebsmittel, so muß die Reihenfolge, in der dieses Betriebsmittel an die Programme vergeben wird, bei der Programmierung festgelegt werden. Die Automationsprogramme werden in der Regel nicht "parallel" sondern in der Reihenfolge ihrer Wichtigkeit "quasiparallel" zueinander ausgeführt.

Unter "Parallelprogrammierung" wird die Unterteilung von Automationsprogrammen in parallel bzw. quasiparallel zueinander auszuführende Anweisungsfolgen und die Steuerung des Ablaufs dieser Anweisungsfolgen in der Weise verstanden, daß die Überwachung, Steuerung und Regelung technischer Prozesse gemäß den Angaben in den zugeordneten Automatisierungsfunktionen unter bestmöglicher Ausnutzung der Betriebsmittel eines Prozeßrechensystems vollzogen wird.

Bei der Programmierung der frühen Prozeßrechensysteme zu Beginn der sechziger Jahre musste die Vergabe von Betriebsmitteln vom Benutzer selbst erledigt werden. Da diese Organisation bei praktisch jedem Einsatz eines Prozeßrechensystems durchgeführt werden muß und einen Überblick über das gesamte mit dem Rechensystem zu lösende Aufgabenspektrum voraussetzt, ging man jedoch bald dazu über, ablaufsteuernde Funktionen in die Betriebssysteme von Prozeßrechnern einzubauen, die den Benutzer möglichst weitgehend von dieser Arbeit befreien sollten.

Die meisten Betriebssysteme für Prozeßrechner enthalten heute Funktionen

für die Betriebsmittelverwaltung. Aufgrund einer Prioritätsangabe, die häufig in Form einer Programmnummer oder Prioritätsnummer beim Einbau eines Automationsprogramms in ein Programmsystem angegeben wird, kann dem Betriebssystem die Wichtigkeit eines Programms vorgegeben werden. Das Betriebssystem sorgt dann dafür, daß den jeweils wichtigsten lauffähigen Programmen die für ihren Ablauf erforderlichen Betriebsmittel zugeteilt werden.

Für die Steuerung des Programmablaufs stellen Betriebssysteme in der Regel eine Anzahl von Standard-Nahtstellen bereit. So gibt es häufig Betriebssystemeinsprünge bzw. "Makroaufrufe", über die der Anwender die Ausführung eines Programms beim Betriebssystem anmelden oder für eine bestimmte Zeitspanne unterbrechen kann.

Während die Schnittstellen zwischen Anwenderprogramm und Betriebsmittelverwaltung heute von Betriebssystem zu Betriebssystem unterschiedlich gestaltet sind und mehr oder weniger den Geschmack und Erkenntnisstand der Betriebssystemkonstrukteure widerspiegeln, wird bei der Entwicklung von höheren Programmiersprachen für die Prozeßautomatisierung versucht, eine möglichst einheitliche und anwendungsgerechte Schnittstelle zur Betriebsmittelverwaltung festzulegen [35]. Nur hierdurch kann, wie in Abschnitt 1.1 schon dargelegt wurde, die weitgehend hardware-unabhängige Formulierung von Prozeßautomationsprogrammen ermöglicht werden. Prozeßorientierte Programmiersprachen können als natürliche Verallgemeinerung technisch-wissenschaftlicher und kommerzieller Programmiersprachen angesehen werden. Während in letzteren nur die Arbeitsweise von Ein/Ausgabeprogrammen für die Standard-Peripherie (z. B. Drucker, Kartenleser, u. s. w.) festgelegt ist, wird in prozeßorientierten Programmiersprachen auch die der Programme für die Prozeß-Ein/Ausgaben und zur Realisierung der Parallelprogrammierung vorgegeben.

Darüberhinaus ist in einer höheren Programmiersprache auch die Aufteilung eines Automationsprogramms in parallel zueinander auszuführende Anweisungsfolgen dokumentationsfreundlich darstellbar. Die in einer Automatisierungsfunktion spezifizierten parallel auszuführenden Anweisungsfolgen zur Bedienung von Teilprozessen brauchen nicht mehr als Einzelprogramme erstellt zu werden, die vom Benutzer unabhängig voneinander in ein Rechensystem eingebaut werden, sondern können gemeinsam innerhalb eines Quellprogramms dargestellt werden.

1.4 Ziel der Arbeit.

In Abschnitt 1.1 wurde schon angedeutet, daß die Sprachmittel für die Parallelprogrammierung in prozeßorientierten Programmiersprachen bisher nicht durchwegs in zweckdienlicher Weise festgelegt wurden. Dies betrifft insbesondere die Sprachelemente zur Steuerung der Betriebsmittelvergabe, deren Wahl bisher in Anlehnung an die Eigenschaften heute verfügbarer Realzeit-Betriebssysteme getroffen wurde. Die auf derzeitige Implementationen bezogene Auswahl von Ausdrucksmitteln bedingt, daß ein Teil der in Abschnitt 1.1 erläuterten Nachteile auch bei Einsatz der bisher vorgeschlagenen prozeßorientierten Programmiersprachen weiter besteht.

Eine der wichtigsten Forderungen, die bei der Entwicklung einer prozeßorientierten Programmiersprache erfüllt werden sollte, ist die Forderung nach unabhängiger Erstellbarkeit von Automationsprogrammen für voneinander unabhängige technische Prozesse. Wie bei der Erstellung technisch-wissenschaftlicher und kommerzieller Programme sollte es möglich sein, ein Automationsprogramm ohne Bezugnahme auf weitere mit dem Rechensystem auszuführende Programme zu erstellen. Ist dies gewährleistet, so braucht sich der Anwender bei der Formulierung, Pflege und Weiterentwicklung eines Automationsprogramms nur mit der ihm zugewiesenen Aufgabe zu beschäftigen. Die übrigen, parallel bzw. quasiparallel auszuführenden Automationsprogramme brauchen dabei nicht betrachtet zu werden. Weiter ist es nur unter dieser Voraussetzung möglich, wiederverwendbare Programmpakete zu erstellen.

Zur Erfüllung der angegebenen Forderung wurde in PEARL ([18], S.5, 1. Absatz) die Erstellung von Programmsystemen in sog. PEARL-Programmoduln ermöglicht. Jedoch lässt sich schon anhand einfacher Beispiele zeigen, daß die in PEARL vorgesehene Betriebsmittelvergabe über Prioritätsnummern nur bei schwacher Auslastung eines Prozeßrechensystems die unabhängige Programmierung von Automatisierungsfunktionen in Form von Programmoduln zulässt. In der vorliegenden Arbeit wird ein Verfahren zur Steuerung der Betriebsmittelvergabe vorgeschlagen, daß den modularen Aufbau von Prozeßrechner-Programmsystemen gestattet.

2. Grundzüge einer Theorie der Aufgabendurchführung.

2.1 Der Taskbegriff in PL/1.

Die Parallelprogrammierung in einer höheren Programmiersprache wurde erstmalig in PL/1 (siehe z. B. [36]) geboten, um dem Benutzer der IBM-Rechnerserie 360 die Organisation des Parallelbetriebs von Zentraleinheiten und von Geräten der Standard-Peripherie zu erleichtern. Um gleichzeitig auszuführende Anweisungsfolgen handhaben zu können, werden in PL/1 die Programmgrößen "Task", "Event" und "Priority" eingeführt.

Ein PL/1-Programm besteht aus einer Menge von Prozeduren, die jeweils aus einer Folge von PL/1-Anweisungen bestehen. Unter den Prozeduren ist eine, die sog. Hauptprozedur, durch deren Aufruf ein Programm gestartet werden kann, ausgezeichnet. Jede Prozedur kann weitere Prozedurvereinbarungen und Anweisungen zum Aufruf von Prozeduren enthalten.

In einer Anweisung zum Aufruf einer Prozedur kann festgelegt werden, wie die aufzurufende Prozedur ausgeführt werden soll. Dabei werden in PL/1 zwei Aufrufarten unterschieden (vergl. Abb. 2.1):

1. Serielle Ausführung (Unterprogramm).
   Die aufrufende Prozedur soll nach Ausführung der Aufrufanweisung unterbrochen werden bis die aufgerufene Prozedur ausgeführt ist. Nach Beendigung der aufgerufenen Prozedur wird die aufrufende Prozedur fortgesetzt.
2. Parallele (asynchrone) Ausführung.
   Die aufrufende Prozedur soll nach Ausführung der Aufrufanweisung fortgesetzt werden. Die aufgerufene Prozedur soll gleichzeitig ausgeführt werden.

Die hier interessierende zweite Aufrufart unterscheidet sich von der zuerst genannten dadurch, daß die aufgerufene Prozedur nicht als Unterprogramm, sondern unter Kontrolle des Betriebssystems (Operating System 360) ausgeführt werden soll. Um mit asynchron ablaufenden Prozeduren auf Quellsprachenebene arbeiten zu können, wird in PL/1 die Programmgröße "Task" eingeführt:

> Eine PL/1-Task ist eine identifizierbare Ausführung einer Prozedur und aller von ihr seriell aufgerufenen Prozeduren. Eine PL/1-Task ist dynamisch; sie existiert nur während der Ausführung eines Programms oder von Teilen eines Programms.

Aus dem oben genannten Grund kann man stattdessen auch von folgender **Definition ausgehen:**

Eine PL/1-Task ist die dynamische Ausführung einer Prozedur und aller von ihr seriell aufgerufenen Prozeduren unter Regie eines Betriebssystems.

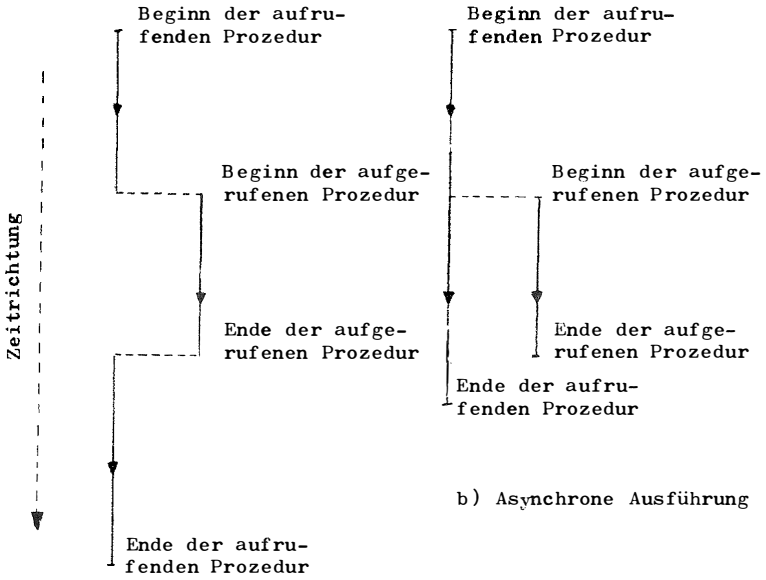


Abb. 2.1 Serielle und asynchrone Ausführung von Prozeduren in PL/1.

Eine PL/1-Task existiert danach nur innerhalb der Zeitspanne, während der eine Prozedur asynchron ausgeführt wird. Durch den Zusatz "unter Regie des Betriebssystems" wird zum Ausdruck gebracht, daß

- die Ausführung einer Prozedur als Task durch ein Betriebssystem gesteuert wird und daß
- eine Task auch während der Zeitspannen identifizierbar ist, in denen sie unterbrochen wurde.

Um eine PL/1-Task zu erzeugen, muß in einer Anweisung zum Aufruf einer Prozedur mindestens eine der folgenden Angaben auftreten:

1. Task-Option.
   Jeder Task außer einer Haupttask[1] kann bei ihrer Erzeugung ein Name

1) Eine Haupttask wird durch Aufruf einer Hauptprozedur über Bedienungseingabe (Job Control) erzeugt.

(Bezeichner) zugeordnet werden. Über den Tasknamen kann im weiteren Verlauf eines Programms eine bestimmte Ausführung einer Prozedur identifiziert werden.

2. Event-Option.
Bei den meisten Anwendungen muß der Ablauf von Tasks synchronisiert werden. Um ein einfaches Beispiel vor Augen zu haben, sei angenommen, daß eine Anzahl von Werten berechnet und auf einem Drucker ausgegeben werden soll. Zerlegt man das Programm in zwei Prozeduren, von denen die eine zur Berechnung von Werten eingesetzt wird, während die andere zum Drucken schon berechneter Werte dient, so kann der Druckvorgang parallel zur Berechnung der Werte erfolgen.

Die asynchrone Ausführung der beiden Prozeduren muß gelegentlich synchronisiert werden. So muß z. B. die Task, die Werte druckt, unterbrochen werden, wenn die Task, die Werte bereitstellt, noch keine neuen Rechenergebnisse geliefert hat. Die Ausgabetask kann fortgesetzt werden, sobald Rechenergebnisse vorliegen.

Die Synchronisation von Tasks wird in PL/1 unter Verwendung sog. Ereignisvariablen durchgeführt, welche die Bitwerte 'Ø'B und '1'B annehmen können. Der Name der Ereignisvariablen wird beim Aufruf einer Prozedur in der sog. Event-Option angegeben.

Bei der Erzeugung einer Task, der eine Ereignisvariable zugeordnet wurde, nimmt diese den Bitwert 'Ø'B an. Der Wert '1'B kann der Ereignisvariablen während des Ablaufs einer Task durch eine COMPLETION-Anweisung zugewiesen werden und wird automatisch bei normaler Beendigung der Task angenommen.

Vermittels einer WAIT-Anweisung kann die Fortsetzung von Tasks vom Wert einer Ereignisvariablen abhängig gemacht werden. Besitzt die Ereignisvariable bei der Ausführung der WAIT-Anweisung den Wert 'Ø'B, so wird die Task, während der die WAIT-Anweisung ausgeführt wird, solange unterbrochen, bis die Ereignisvariable den Wert '1'B annimmt.

3. Priority-Option.
Häufig benötigen mehrere Tasks, die vom Betriebssystem gleichzeitig zum Ablauf gebracht werden sollen, dasselbe Betriebsmittel. Sind die Tasks verschieden wichtig, so kann die Reihenfolge, in der sie ablaufen sollen, vom Programmierer durch Angabe einer Prioritätsnummer festgelegt werden.

In PL/1 geht aus der Prioritätsnummer die relative Wichtigkeit einer Task bezüglich der Task (Muttertask) hervor, durch die sie erzeugt wurde. Eine Task kann in PL/1 nie wichtiger als ihre Muttertask sein. Die Priorität einer Haupttask wird beim Start eines Programms festgelegt.

2.2 Erweiterungen des Taskbegriffs von PL/1 für die Zwecke der Prozeßautomatisierung.

Schon anhand einfacher Beispiele kann man erkennen, daß der Taskbegriff von PL/1, der für kommerzielle Anwendungen ausreicht, für die Zwecke der Prozeßautomatisierung zu eng gefasst ist. In der Tat gehen die meisten der bisher veröffentlichten prozeßorientierten Programmiersprachen von abweichenden Definitionen aus.

Um zu einer geeigneten Definition des Taskbegriffs zu gelangen, sollen zunächst zwei bei der Lösung von Prozeßautomatisierungsaufgaben häufig vorkommende Problemstellungen diskutiert werden. Dabei wird von der in Abschnitt 2.1 angegebenen Taskdefinition ausgegangen und es werden bestimmte Systemdienste vorausgesetzt, die von Realzeit-Betriebssystemen häufig geboten werden.

Beispiel 1:

Eine Prozedur (etwa zur Ausgabe eines Protokolls) soll bei Erreichen einer vorgegebenen Uhrzeit als Task ausgeführt werden. Das Betriebssystem bietet einen Systemdienst, der nach Beauftragung eine Task zu einer vorgebbaren Uhrzeit selbstständig startet.

Um den vorausgesetzten Systemdienst einsetzen zu können, müsste eine Task schon zum Zeitpunkt der Beauftragung des Systemdienst identifizierbar sein. Da die Task aber erst existiert, wenn die Anweisung zum Aufruf der Prozedur ausgeführt wird, kann sie dem Systemdienst nicht bekannt gemacht werden. Der Systemdienst wäre also im Widerspruch zur Voraussetzung nicht realisierbar.

Beispiel 2:

Eine Prozedur (z. B. zur Meßwerterfassung) soll in vorgegebenen Zeitabständen wiederholt werden. Das Betriebssystem bietet einen Systemdienst, der nach Beauftragung den zyklischen Start einer Task selbstständig durchführt.

Da jede asynchrone Ausführung der Prozedur eine neue Task darstellt, müsste man bei der Beauftragung des Systemdienstes eine abzählbar unendliche Anzahl von Tasks identifizieren können. Da diese Tasks aber zum Zeitpunkt der Beauftragung nicht existieren, ist der Systemdienst nicht realisierbar.

Wie die Beispiele zeigen, wäre die Verwendung des Taskbegriffs von PL/1 in prozeßorientierten Programmiersprachen nicht problemgerecht und für den Anwender unbequem. Häufig einsetzbare Systemdienste, die normalerweise von Prozeßrechner-Betriebssystemen geboten werden, müssten vom Anwender

selbst durch geeignete Gestaltung des Automationsprogramms realisiert werden.

Die Unzweckmäßigkeit der PL/1-Taskdefinition für die Prozeßautomatisierung rührt ersichtlich daher, daß in PL/1 eine Task nur dynamisch existiert; Deklaration und Start einer Task fallen zusammen. Die Ausführung einer Prozedur unter Regie eines Betriebssystems ist in PL/1 keine Programmgröße im üblichen Sinn, die unabhängig von ihrer Verwendung definiert werden kann. Erweitert man den Taskbegriff dahingehend, so gelangt man zu einer für die Zwecke der Prozeßautomatisierung hinreichenden Begriffsbildung.[1)]

Als besonderer Vorteil des so erweiterten Taskbegriffs für die Prozeßautomatisierung ist die einfache Übertragbarkeit von Angaben in einer Automatisierungsfunktion in ein Automationsprogramm hervorzuheben. Wurde in einer Automatisierungsfunktion spezifiziert, welche Teile des Automationsprogramms asynchron zueinander auszuführen sind und wann bzw. wie diese Programmteile ausgeführt werden sollen, so kann jedes Teilprogramm durch eine Folge von von Anweisungen dargestellt werden, deren Ausführung im Automationsprogramm als Task vereinbart wird. Die Ausführungsbedingungen, aus denen hervorgeht, wann Tasks zu starten, zu unterbrechen, fortzusetzen und zu beenden sind, werden unter Verwendung von Anweisungen zur Handhabung von Tasks dargestellt.

Zur Erläuterung dieser Vorgehensweise sollen die in den Beispielen zu Beginn dieses Abschnitts angegebenen Problemstellungen unter Verwendung der in PEARL [18,47] vorgeschlagenen Syntax dargestellt werden. Die gemäß:

```
PROTOKOLL:    TASK;
                 .
                 .
                 .
              Anweisungsfolge zur Ausgabe
              eines Protokolls
                 .
                 .
                 .
              END;
KONTROLLE:    TASK;
                 .
                 .
                 .
              Anweisungsfolge für die Übernahme
```

1) In Anlehnung an PL/1 wird in dieser Arbeit weiterhin die Bezeichnung "Task" verwendet. Nach Dijkstra [34] wird neuerdings auch die Bezeichnung "Prozeß" benutzt.

```
                und Kontrolle von Meßwerten
                        .
                        .
                        .
                    END;
```

vereinbarten Tasks PROTOKOLL und KONTROLLE werden nach Ausführung der Anweisungen:

```
AT 12:Ø:Ø ACTIVATE PROTOKOLL;
ALL 5 SEC ACTIVATE KONTROLLE;
```

täglich um 12 Uhr bzw. alle 5 Sekunden zur Ausführung freigegeben.

Um asynchron ablaufende Vorgänge in Prozeßautomatisierungssystemen einheitlich beschreiben zu können, ist es zweckmäßig, von einer Verallgemeinerung der früher verwendeten Begriffe "Anweisungsfolge" und "Betriebssystem" auszugehen. Ersetzt man diese Begriffe durch "Lösung einer Aufgabe" und durch "Organisator", so kann eine Task wie folgt definiert werden:

> Eine Task ist die selbstständige oder von Organisatoren geleitete Durchführung der Lösung einer Aufgabe.

Eine Aufgabe wird in der Automatisierungsfunktion spezifiziert (z. B. Ausgeben eines Protokolls um 12 Uhr). Die Lösung einer Aufgabe kann in einem Automationsprogramm oder in einem Gerät (Zähler, Regelkreis, u. s. w.) bestehen. Die Begriffe "Aufgabe" und "Lösung einer Aufgabe" brauchen jedoch nicht präzisiert zu werden, da sich die folgenden Abschnitt nur damit beschäftigen, wie Lösungen von Aufgaben durchgeführt werden, und nicht damit, worin "Aufgaben" und "Lösungen von Aufgaben" bestehen können.[1] Zur Vereinfachung der Darstellung wird im folgenden zwischen den Begriffen "Aufgabe" und "Lösung einer Aufgabe" nicht unterschieden.

Der hier eingeführte Taskbegriff besitzt bei Anwendung auf die Prozeßautomatisierung den Vorteil, daß asynchrone Abläufe in Prozeßautomatisierungssystemen unabhängig von ihrer Realisierung spezifiziert werden können. Sämtliche in einer Automatisierungsfunktion angegebenen Abläufe im Prozeßrechensystem und im technischen Prozeß werden als Tasks angesehen. Im Automationsprogramm werden die mit einem Prozeßrechensystem durchzuführenden Aufgaben

---

1) Eine analoge Situation findet man beispielsweise in der Theorie der Warteschlangen. Der dort eingeführte Begriff "Operation" braucht selbst nicht erklärt zu werden, da sich die Theorie nur mit der Reihung von Operationen und nicht mit den Operationen selbst beschäftigt (siehe z. B. [37], S. 4).

beschrieben. Wie sich in Abschnitt 2.3 zeigen wird, ordnet sich wegen der einheitlichen Behandlung aller Parallelabläufe die zeitabhängige und stochastische Steuerung von Tasks natürlich unter dem Begriff der Synchronisation ein.

2.3 Organisation von Tasks; Ereignisse.

Wie früher schon erwähnt wurde, tritt häufig der Fall ein, daß der Ablauf einer Task durch eine andere Task gesteuert werden muß oder daß der Ablauf mehrerer Tasks miteinander zu synchronisieren ist. Die zur Koordination gekoppelter Tasks erforderlichen Maßnahmen werden gewöhnlich nicht von den Tasks selbst gelöst, sondern durch Manager bzw. Organisatoren wahrgenommen. Durch die zweckmäßige Aufteilung in sog. fachbezogene und in organisatorische Tätigkeiten wird, insbesondere bei einer großen Anzahl asynchron durchzuführender Aufgaben, eine übersichtliche und i. a. auch effiziente Aufgabendurchführung gewährleistet.

Um zwischen der Durchführung von Aufgaben und ihrer Koordination unterscheiden zu können, wurde in Abschnitt 2.2 der Begriff des Organisators eingeführt. Darunter soll ein Gebilde verstanden werden,

- dem Tasks bekannt und unbekannt gemacht werden können und
- daß auf Anweisung hin den Ablauf von Tasks steuert und synchronisiert.

Danach kann sowohl ein Operator, der ein Schaltpult bedient, als auch ein Betriebssystem (Organisationsprogramm), das die Ausführung von Anweisungsfolgen steuert, als Organisator angesehen werden.

Es ist bemerkenswert, daß die vorangegangenen Definitionen eine hierarchisch geordnete Struktur von Organisatoren zulassen. Die Durchführung von Organisationsaufgaben kann ja nach der Definition der Task in Abschnitt 2.2 selbst wieder als Task angesehen werden. Um die Hierarchie nach oben abschließbar zu machen, wurde zwischen der selbstständigen und organisierten Aufgabendurchführung unterschieden.

Damit ist man auf natürliche Weise zu einem schichtenweise geordneten Organisationsschema gelangt, das im täglichen Leben (z. B. bei der Abwicklung von Projekten) und neuerdings auch bei der Entwicklung von Betriebssystemen (z. B. [38,39]) erfolgreich eingesetzt wird. Die Anzahl der Organisationsebenen (Schichten) und die Funktionen der Organisatoren in den einzelnen Ebenen (Aufgabe der Schichten) müssen für das jeweils zu lösende Aufgabenspektrum in zweckmäßiger Weise festgelegt werden.

Dabei sollten die folgenden beiden Forderungen so gut wie möglich erfüllt werden:

1. Durch das Organisationsschema sollte gewährleistet werden, daß von einer Organisationsebene nur die jeweils benachbarten Ebenen angesprochen werden können.

2. Die Beauftragung eines Organisators sollte so erfolgen, daß Aussagen über die Weise, in der Leistungen vom Organisator erbracht werden sollen, soweit wie möglich vermieden werden.

Da man sich bei der Bestimmung der Ausdrucksmittel für die Parallelprogrammierung in der äußersten Schicht (Anwenderebene) befindet, ist die erste Forderung, durch deren Erfüllung Systemverklemmungen (deadlocks) vermeidbar sind [38], hier von untergeordneter Bedeutung. Die zweckmäßige Aufteilung eines Betriebssystems in Schichten ist Aufgabe des Implementators.

Dagegen sollte die zweite Forderung bei der Entwicklung prozeßorientierter Programmiersprachen möglichst gut erfüllt werden. Nur wenn die Sprachelemente anwenderfreundlich und nicht implementationsbezogen gewählt werden, sind die in Abschnitt 1.1 beschriebenen Vorteile höherer Programmiersprachen erreichbar. Inwieweit die bisher vorgeschlagenen Ausdrucksmittel für die Parallelprogrammierung dieser Forderung genügen, wird in Abschnitt 2.7 und in Kapitel 3 untersucht.

Der Mechanismus, der es ermöglicht, einem Organisator die Synchronisation von Tasks zu übertragen, lässt sich bequem unter Verwendung des Ereignisbegriffs darstellen. Was unter einem Ereignis verstanden werden soll, ist schon oft diskutiert worden, ohne daß m. W. bisher eine hinreichende Definition bekannt geworden ist. Der Grund hierfür liegt wohl darin, daß in den bisher diskutierten Modellen zur Beschreibung der Aufgabendurchführung in der Regel Funktionen verschiedener Organisationsebenen gleichzeitig verwendet werden. Ein Beispiel, bei dem eine solche Vermischung vorliegt, wird in Abschnitt 2.4 diskutiert.

Geht man davon aus, daß die Synchronisation von Tasks durch einen Organisator bei Erfüllung vorgegebener Synchronisierbedingungen (z. B. Eintritt einer Uhrzeit, Ablauf einer Zeitspanne, Eintreffen eines Unterbrechungssignals, Setzen einer Ereignisvariablen) durchgeführt wird, so liegt folgende Definition nahe:

> Ein Ereignis ist der Eintritt des Erfülltseins einer Synchronisierbedingung.

Die Synchronisation von Tasks kann damit wie folgt beschrieben werden: Einem Organisator werden Maßnahmen (z. B. Starten von Tasks, Unterbrechen von Tasks, Fortsetzen von Tasks) zusammen mit Synchronisierbedingungen vorgegeben. Erkennt der Organisator ein Ereignis, so führt er die zugeordneten Maßnahmen aus. Durch Mitteilungen (z. B. Triggern eines Unterbrechungssignals) können einem Organisator Ereignisse zur Kenntnis gebracht werden.

Daß der beschriebene Mechanismus zur Darstellung von Synchronisationsvorgängen ausreicht, soll im folgenden anhand der Synchronisation mit Semaphorevariablen (vergl. Abschnitt 1.2) erläutert werden. Durch eine REQUEST-Anweisung wird der Organisator beauftragt, bei einem Semaphoreereignis die Steueranweisung "Fortsetzen einer Task" auszuführen. Ist ein solches Ereignis schon eingetreten, so kann die Steueranweisung sofort durchgeführt werden. Durch Erniedrigen der Semaphorevariablen sorgt der Organisator dafür, daß nur so viele Fortsetzungsanweisungen ausgeführt werden, wie Ereignisse eingetreten sind. Durch RELEASE-Anweisungen werden dem Organisator Semaphoreereignisse zur Kenntnis gebracht.

Die Anweisungen zur Ansprache von Organisatoren können daher in die folgenden drei Klassen eingeteilt werden:

1. Definitorische Anweisungen.
   Durch definitorische Anweisungen (Deklarationen) werden Organisatoren Tasks bekannt und unbekannt gemacht.

2. Steueranweisungen.
   Mittels Steueranweisungen kann der Ablauf von Tasks unmittelbar oder mittelbar, d. h. in Abhängigkeit von Ereignissen, beeinflusst werden.

3. Synchronisieranweisungen.
   Über Synchronisieranweisungen können Synchronisierbedingungen unmittelbar oder mittelbar erfüllt, aufgehoben und wieder wirksam gemacht werden.

2.4 Zustände von Tasks.

Bei der Klassifizierung der Anweisungen zur Beauftragung von Organisatoren war bisher implizite vorausgesetzt worden, daß man bei der Organisation von Tasks verschiedene diskrete Stadien oder Zustände unterscheiden kann, die durch Ausführung definitorischer Anweisungen angenommen bzw. verlassen und durch Ausführung von Steueranweisungen ineinander übergeführt werden.

Aus der früher angegebenen Forderung, daß man die Ausdrucksmittel zur Beschreibung der Aufgabendurchführung möglichst ohne Bezugnahme auf spezielle Organisatoren festlegen sollte, folgt für die Unterteilung von Taskzuständen:

1. Bei Zustandsdefinitionen sollen Aussagen darüber vermieden werden, wie Tasks von einem Organisator verwaltet werden.

2. Von speziellen Betriebsmitteln abhängige Zustände sind nicht zugelassen.

Diesen Forderungen kann man mit folgender Zustandseinteilung genügen:

Zustand "bekannt":

Die Task ist bei einem Organisator bekannt. Welche Aufgabe durchgeführt werden soll, braucht noch nicht festgelegt worden zu sein (Taskvariable, vergl. Abschnitt 2.7).

Zustand "ablaufbereit":

Die Task wartet auf für ihren Ablauf erforderliche Betriebsmittel.

Zustand "ablaufend":

Die Task besitzt alle für ihren Ablauf erforderlichen Betriebsmittel.

Zustand "zurückgestellt":

Die Task wurde unterbrochen.

Die Unterscheidung zwischen dem Zustand "bekannt" und dem Zustand "ablaufbereit" wurde in Abschnitt 2.2 begründet. Dort wurde gezeigt, daß die Taskorganisation von PL/1, die keine Taskkonstanten zulässt, für die Prozeßautomatisierung nicht zweckmäßig ist. Der in anderen Zustandmodellen (siehe z. B. [40]) eingeführte Zustand "unbekannt" wird hier nicht benötigt. Es ist nicht sinnvoll, eine bestimmte Task als "unbekannt" zu bezeichnen. Um nämlich diese Aussage machen zu können, muß die Task identifizierbar sein, was wiederum nur möglich wäre, wenn die Task bekannt ist.

Die bisher publizierten Zustandmodelle unterschieden sich von der hier vorgenommenen Einteilung im wesentlichen durch weitere Untergliederung der angegebenen vier Elementarzustände. Dabei werden in der Regel Voraussetzungen über die Gestaltung der Task- und Betriebsmittelverwaltung (d. h. über die Realisierung des Organisators) getroffen. Viele der Schwierigkeiten, die bei den Versuchen aufgetreten sind, zu einer für die Konstruktion von prozeßorientierten Programmiersprachen ausreichenden Beschreibung der Aufgabendurchführung zu gelangen, sind auf Zustandeinteilungen zurückzuführen, bei denen die Funktionsweise spezieller Organisatoren berücksichtigt wurde.

Als Beispiel sei hier die vom LTPL-Komitee vorgeschlagene Einteilung betrachtet. In dem LTPL-Report [40] wird zwischen folgenden sieben Zuständen unterschieden:

0. Zustand "nonexistent".
Die Task[1)] ist dem System unbekannt, d. h. es wurde kein Speicherplatz

---

1) In dem Report wird unter einer Task ein Rechenprozeß, d. h. die Ausführung einer Anweisungsfolge durch einen Prozessor, verstanden.

für den Zustandsvektor[1] der Task vergeben.

1. Zustand "dormant".
   Die Task ist dem System bekannt, d. h. Speicherplatz für den Zustandsvektor wurde vergeben. Jedoch wurde weder das Segment, d. h. die auszuführende Anweisungsfolge, zugeordnet, noch wurde Speicherplatz für die Daten vergeben.

2. Zustand "idle".
   Die Task existiert. Ein Segment und Speicherplatz für die Daten wurden zugeordnet. Wann die Task ablaufen soll, ist noch nicht festgelegt worden.

3. Zustand "runnable".
   Die Task ist ablaufbereit. Alle angeforderten Betriebsmittel mit Ausnahme einer Zentraleinheit wurden ihr zugeordnet.

4. Zustand "suspended".
   Die Task wartet auf den Eintritt eines Ereignisses[2], beispielsweise darauf, daß ein Betriebsmittel verfügbar wird oder auf ein Signal zur Fortsetzung von einer anderen Task.

5. Zustand "scheduled".
   Die Ausführung des Segments einer Task ist eingeplant. Die Task soll zu einem späteren Zeitpunkt oder bei Eintritt eines Ereignisses ausgeführt werden. Wenn dieser Zeitpunkt oder das Ereignis eintritt, wird die Task automatisch in den Zustand "runnable" überführt.

6. Zustand "running".
   Das Segment einer Task wird durch eine Zentraleinheit ausgeführt.

Hinsichtlich des Zustands "nonexistent" räumen die LTPL-Autoren ein, daß dieser Zustand eigentlich kein wirklicher Zustand sei, daß aber bei seiner Verwendung die Darstellung klarer wird. Inwieweit das bei Einsatz als unangemessen erkannter Beschreibungsmittel möglich ist, sei dahingestellt.

---

1) Der Zustandsvektor enthält die für die Ausführung der Task erforderlichen Angaben wie Taskzustand, Prioritätszahl, Einplanbedingungen, u. s. w. Er entspricht dem sog. Task-Kontroll-Block, der in der Betriebssystemtheorie verwendet wird (vergl. z. B. [41], S. 632 und Abschnitt 3.5 der vorliegenden Arbeit).

2) In dem Report wird der Ereignisbegriff wie folgt eingeführt: "Ein Ereignis ist das Auftreten einer programmierten Wirkung innerhalb eines Rechenprozesses".

Vergleicht man die Zustandseinteilung von LTPL (Zustand 1 bis 6) mit der zu Beginn des Abschnitts vorgeschlagenen Einteilung, so ergibt sich folgende Zuordnung:

- Die von LTPL vorgeschlagenen Zustände "dormant", "idle" und "scheduled" wurden unter dem Zustand "bekannt" zusammengefasst.

  Die von LTPL vorgenommene Unterscheidung ist vielleicht bei der Konstruktion eines Realzeit-Betriebssystems sinnvoll, interessiert aber den Anwender nicht. Für die Anwendung ist es hinreichend, wenn eine Aufgabe entweder bei der Bekanntmachung einer Task oder in einer Steueranweisung spezifiziert werden kann, mittels der die Task vom Zustand "bekannt" in den Zustand "ablaufbereit" gebracht wird. Eine Task, die bei einem Ereignis in den Zustand "ablaufbereit" gebracht werden soll, befindet sich bis zu dem Ereignis im Zustand "bekannt".

- Der Zustand "runnable" und der Zustand "suspended", soweit er wegen Nichtverfügbarkeit von Betriebsmitteln angenommen werden soll, sind unter dem Zustand "ablaufbereit" zusammengefasst worden.

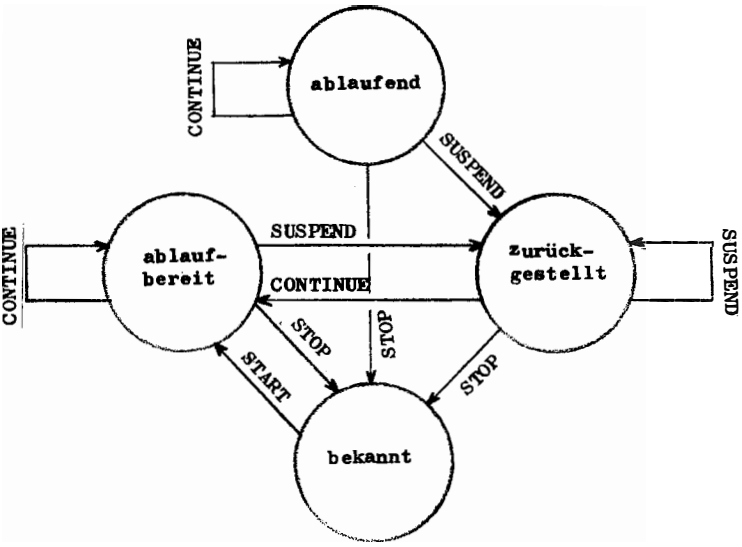
  Diese Zusammenfassung folgt aus der eingangs angegebenen Forderung, daß von speziellen Betriebsmitteln abhängige Zustände nicht zulässig sind. Die Einführung des Zustand "runnable" in LTPL kann nur historisch verstanden werden und geht offenbar davon aus, daß die Nichtverfügbarkeit einer Zentraleinheit einen Engpaß bei der Ausführung von Prozeßautomationsprogrammen darstellt. Da es häufig vorkommt, daß mehrere Tasks auf die Zuteilung eines langsamen oder nur exklusiv verwendbaren E/A-Geräts warten (z. B. langsame Analogeingabe, Drucker), während die Zentraleinheit unbeschäftigt ist, müsste man auch von solchen Betriebsmitteln abhängige Zustände vorsehen. Wie man auch ohne betriebsmittelabhängige Taskzustände eine Optimierung der Betriebsmittelvergabe auf Sprachniveau erreichen kann, wird in Abschnitt 2.6 diskutiert.

- Der Zustand "suspended", soweit er wegen Wartens auf ein (LTPL-) Ereignis zur Fortsetzung einer Task oder aufgrund einer Steueranweisung angenommen wird, entspricht dem Zustand "zurückgestellt".

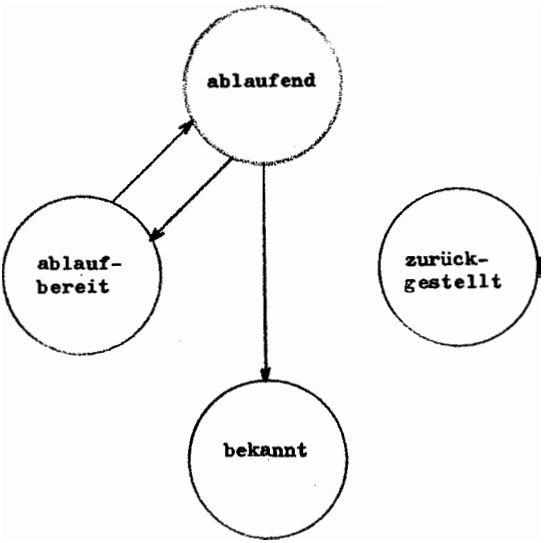
- Die Zustände "running" und "ablaufend" entsprechen sich.

2.5 Übergänge zwischen den Zuständen von Tasks.

Zwischen den vier Zuständen einer Task "bekannt", "ablaufbereit", "zurückgestellt" und "ablaufend" sind insgesamt 16 Übergänge denkbar. Davon sind jedoch nur die in Abb. 2.2 dargestellten 12 Übergänge sinnvoll. In der Abbildung wurden Übergänge, die von einem Organisator selbstständig voll-



a) Übergänge aufgrund von Steueranweisungen.



b) Organisierte Übergänge

Abb. 2.2 Zustände und Übergänge zwischen den Zuständen von Tasks.

zogen werden, und Übergänge aufgrund von Steueranweisungen getrennt dargestellt.

a) Ausgangszustand "bekannt".

Vom Zustand "bekannt" kann eine Task nur durch eine Start-Anweisung in den Zustand "ablaufbereit" gelangen.

Der Übergang "bekannt" → "zurückgestellt" ist nicht sinnvoll, da eine Task vor ihrem Start nicht unterbrochen werden kann. Auch der Übergang in den Zustand "ablaufend" ist nicht möglich, da die Betriebsmittelzuteilung nur unter Berücksichtigung der Wichtigkeit aller gleichzeitig ablaufbereiten Tasks, d. h. vom Organisator, vollzogen wird.

b) Ausgangszustand "ablaufbereit".

Vom Zustand "ablaufbereit" kann eine Task alle drei weiteren Zustände erreichen.

Der Übergang in den Zustand "ablaufend" wird, wie unter a) beschrieben, durch einen Organisator selbstständig vollzogen. In die Zustände "zurückgestellt" bzw. "bekannt" wird eine Task durch eine Steueranweisung zur Unterbechung (SUSPEND) oder zur Beendigung (STOP) überführt. Schließlich kann durch eine Fortsetzungsanweisung die Wichtigkeit einer Task verändert werden. Die Task verbleibt in diesem Fall im Zustand "ablaufbereit".

c) Ausgangszustand "zurückgestellt".

Erreichbar sind nur die Zustände "ablaufbereit" und "bekannt", wenn eine Steueranweisung zur Fortsetzung (CONTINUE) oder eine STOP-Anweisung ausgeführt wird. Der Zustand "ablaufend" ist aus gleichen Gründen wie unter a) nicht erreichbar. Eine SUSPEND-Anweisung auf eine schon zurückgestellte Task wirkt wie eine Leeranweisung.

d) Ausgangszustand "ablaufend".

Übergänge in alle weiteren Zustände sind möglich.

Der Zustand "zurückgestellt" wird durch Ausführen einer Steueranweisung zur Unterbrechung erreicht. In den Zustand "bekannt" gelangt eine Task, wenn entweder eine Anweisung zur Beendigung ausgeführt wird oder wenn das Segment der Task ausgeführt ist. Im zuletzt genannten Fall wird der Übergang automatisch vom Organisator vollzogen. Der Zustand "ablaufbereit" wird angenommen, wenn einer Task vom Organisator Betriebsmittel entzogen werden, die eine andere wichtigere Task für ihren Ablauf benötigt.[1)]

1) Ein dem hier vorgeschlagenen ähnliches Zustandsmodell mit 5 Task- bzw. Prozeßzuständen wird in [42] vorgestellt.

2.6 Steuerung der Betriebsmittelzuteilung.

Die Einführung des Zustand "ablaufbereit" war erforderlich, weil in Prozeßrechensystemen häufig diesselben Betriebsmittel für den Ablauf mehrerer Tasks benötigt werden. Damit die dasselbe Betriebsmittel anfordernden Task gemäß ihrer Wichtigkeit zum Ablauf gebracht werden können, müssen dem Organisator Entscheidungshilfen vorgegeben werden, aus denen hervorgeht, in welcher Reihenfolge die Betriebsmittel an ablaufbereite Tasks vergeben werden sollen.

Bei der Wahl der die Wichtigkeit von Tasks beschreibenden Parameter sollten folgende Forderungen erfüllt werden:

1. Die Form der Prioritätsangabe soll es ermöglichen, die in einer Automatisierungsfunktion niedergelegten Ausführungsbestimmungen (z. B. Reaktionszeiten) einfach in ein Automationsprogramm zu übertragen.
2. Die Prioritätsvergabe an Tasks soll ohne Bezugnahme auf die Prioritäten anderer Tasks möglich sein.

Es sei vorab bemerkt, daß beide Forderungen in allen bisher publizierten prozeßorientierten Programmiersprachen nicht erfüllt wurden. Bei der Entwicklung von Prozeßprogrammiersprachen wurde bisher durchwegs die für kommerzielle Anwendungen hinreichende Form der Prioritätsangabe übernommen (vergl. Abschnitt 2.1), nach der die Wichtigkeit einer Task durch eine ganze Zahl zum Ausdruck gebracht wird.

Auf den ersten Blick könnte man zu der Annahme gelangen, daß die zweite Forderung, bei deren Erfüllung Programmsysteme zur Automatisierung technischer Prozesse modular aufgebaut werden können, einen Widerspruch in sich darstellt. Durch die Prioritätsangabe soll ja gerade die relative Wichtigkeit von Tasks zum Ausdruck gebracht werden. In der Tat ist die Forderung unerfüllbar, wenn die Reihenfolge, in der ablaufbereite Tasks ausgeführt werden sollen, statisch über Prioritätsnummern festgelegt wird. Geht man jedoch davon aus, daß die Wichtigkeiten von Tasks in Prozeßrechensystemen durch technische Prozesse bestimmt werden, daß also durch Prioritätsangaben die Synchronisierung der Ablaufgeschwindigkeit von Tasks in Prozeßrechensystemen mit der von Tasks in anderen Teilen von Prozeßautomatisierungssystemen herbeigeführt werden soll, so lässt sich eine Form der Prioritätsangabe ableiten, aus der die Wichtigkeit von Tasks dynamisch abgeleitet werden kann (vergl. Abschnitt 3.2).

Die nach den bisherigen Betrachtungen vorzusehenden Sprachelemente für die Parallelprogrammierung sind zwar zur adäquaten Darstellung der Angaben in einer Automatisierungsfunktion hinsichtlich der Zerlegung eines Automationsprogramms in parallel zueinander auszuführende Programmteile (Tasks)

und für deren Steuerung hinreichend, gestatten aber nur bedingt die optimale Nutzung aller Geräte von Prozeßrechensystemen. Während über Prioritätsangaben dem Organisator vorgegeben wird, in welcher Reihenfolge ablaufbereite Tasks ausgeführt werden sollen, kann damit, ohne Bezugnahme auf das jeweils eingesetzte Prozeßrechensystem, nur umständlich und ineffizient eine gute Ausnutzung langsamer Ein/Ausgabe-Geräte (z. B. Analogeingabegeräte mit Wandlungszeiten im Millisekundenbereich) erreicht werden.

Die angedeutete Problemstellung geht unmittelbar aus folgendem Beispiel hervor: Vorgelegt sei eine Automatisierungsfunktion zur Überwachung eines Dauerprüfstands. Darin sei angegeben, daß eine Anzahl von Meßwerten übernommen und mit vorgegebenen Grenzen verglichen werden soll. Die Reihenfolge der Übernahmen und Kontrollen sei dabei beliebig.

Ohne Kenntnis der Anschlüsse der Meßwertgeber an das Prozeßrechner-Gerätesystem kann mit den bisher beschriebenen organisatorischen Maßnahmen i. a. das Gerätesystem nicht optimal genutzt werden. Folgende Arten der Programmierung sind denkbar:

a) Sequentielle Ausführung der Übernahmen und Kontrollen:
Werden die Meßwertgeber an mehrere langsame Analogeingabewerke angeschlossen, so sind während der Messung eines Werts alle weiteren Werke nicht nutzbar. Vernachlässigt man die im Vergleich zur Wandlungszeit Δt_w eines Eingabewerks normalerweise kurze Zeitspanne für die Ausführung eines Kontrollalgorithmus, so ergibt sich bei N Meßwerten, von denen je N_i; i = 1, ..., M über eines von M Eingabewerken übernommen werden, statt der optimalen Laufzeit der Kontrolltask $\Delta t_w . \max\{N_i;\ i = 1, \ldots, M\}$ eine Laufzeit von $\Delta t_w . \sum_{i=1}^{M} N_i$.

Beispielsweise würde bei sequentieller Übernahme von je 10 Werten über drei langsame Eingabewerke mit einer Wandlungszeit von je 30 msec die Laufzeit der Kontrolltask von 300 msec bei paralleler Benutzung der Eingabewerke auf beinahe 1 sec vergrößert. Soll ein Dauerprüfstand alle 5 sec kontrolliert werden, so wären an das Rechensystem statt maximal 16 nur höchstens 5 solcher Prüfstände anschließbar.

b) Parallele Ausführung der Übernahmen und Kontrollen:
Die parallele und damit hinsichtlich der Prozeß-Peripherie optimale Nutzung der Betriebsmittel eines Prozeßrechensystems ist gewährleistet, wenn für jede Übernahme (einschl. der zugehörigen Kontrolle) eine eigene Task eingeführt wird. Diese Form der Organisation verursacht jedoch neben einer beträchtlichen Vergrößerung des Anwenderprogramms auch eine starke Belastung des Betriebssystems hinsichtlich des Speicherbedarfs zur Ver-

waltung von Tasks (Task-Kontroll-Blöcke) und hinsichtlich der Organisationszeit zur Verwaltung der zahlreichen Tasks.

Für das oben unter a) angegebene Beispiel wären beispielsweise pro Prüfstand statt 3 Task-Kontroll-Blöcken 30 solcher Verwaltungsblöcke erforderlich.

Wie daraus hervorgeht, lässt sich der Parallelbetrieb von langsamen Geräten der Prozeß-Peripherie einerseits und die Verwaltung von Tasks andererseits nur optimieren, wenn die Konfiguration des und der Anschluß der Meßwertgeber und Stellglieder an das Prozeßrechner-Gerätesystem bekannt sind. Diese Arbeit braucht jedoch nicht dem Benutzer angelastet zu werden, sondern kann vom Übersetzer für die prozeßorientierte Programmiersprache, dem das jeweilige Prozeßrechner-Gerätesystem in einem eigenen Programmteil bekannt gemacht werden kann (EQUIPMENT-Teil in INDAC8 und PAS1, SYSTEM-Teil in PEARL) ausgeführt werden. Erforderlich sind dazu neben Sprachmitteln zur Definition und Steuerung von "parallel auszuführenden" Anweisungsfolgen (Tasks), Ausdrucksmittel zur Definition von "parallel ausführbaren" Anweisungsfolgen.

Unter einer "parallel ausführbaren" Anweisungsfolge wird eine Anweisungsfolge verstanden, die parallel zu weiteren Anweisungsfolgen ausgeführt werden darf, aber nicht ausgeführt werden muß. Welche der "parallel ausführbaren" Anweisungsfolgen in "parallel auszuführende" Anweisungsfolgen übersetzt werden, entscheidet der Übersetzer unter Berücksichtigung der Konfiguration des jeweils eingesetzten Prozeßrechensystems und aufgrund der Eigenschaften des verwendeten Organisationsprogramms.

Würde man beispielsweise jede Übernahme und Kontrolle in dem obigen Beispiel als "parallel ausführbare" Anweisungsfolge darstellen, so könnte bei der Übersetzung wie folgt verfahren werden: alle Meßwertübernahmen (einschließlich der zugeordneten Kontrollalgorithmen), in denen jeweils ein bestimmtes Analogeingabegerät angesprochen wird, werden vom Übersetzer in eine sequentiell auszuführende Anweisungsfolge gebracht. In Abhängigkeit von dem jeweils eingesetzten Organisationsprogramm kann der Übersetzer dabei weitere Optimierungen vorbereiten, z. B. die Kontrolle des n-ten Meßwerts durch die Zentraleinheit parallel zur Digitalisierung des (n+1)-ten Meßwerts durch das Analogeingabegerät. Für jede der Anweisungsfolgen bereitet der Übersetzer eine Task (Systemtask) und deren Synchronisation mit der Benutzertask vor.

Ersichtlich kann auf diese Weise die gleiche Organisationsform geschaffen werden, die der Benutzer bei ausschließlicher Verwendung des Taskkonzepts und bei Kenntnis der Konfiguration eines Prozeßrechensystems einsetzen würde. Die Definierbarkeit von "parallel ausführbaren" Anweisungsfolgen

ermöglicht also die von der Konfiguration eines Prozeßrechensystems unabhängige Darstellung von Automatisierungsfunktionen unter bestmöglicher Ausnutzung der Betriebsmittel eines Prozeßrechensystems und trägt damit zur Aufhebung der in Abschnitt 1.1 unter Punkt 2. und Punkt 4. genannten Nachteile bei.

2.7 Folgerungen hinsichtlich der Ausdrucksmittel für die Parallelprogrammierung.

Aus den Betrachtungen der vorangegangenen Abschnitte lassen sich die Anforderungen an die Ausdrucksmittel für die Parallelprogrammierung ableiten. Gemäß der in Abschnitt 2.3 vorgenommenen Einteilung kann dabei zwischen definitorischen Anweisungen, Steueranweisungen und Synchronisieranweisungen unterschieden werden.

a) Definitorische Anweisungen.

Zur Beschreibung des Ablaufs einer Task benötigt man, wie in Abschnitt 2.4 erläutert wurde, nur Aussagen über die Task während der Zeitspanne, in der sie durch ein Betriebssystem verwaltet werden soll. Eine Task wird beim Betriebssystem durch eine Anweisung zum Erzeugen bzw. Vernichten bekannt bzw. unbekannt gemacht.

Wird eine Anweisung zur Erzeugung einer Task ausgeführt, so richtet das Betriebssystem für die Task einen Task-Kontroll-Block ein, über den der weitere Ablauf der Task verfolgt und beeinflusst werden kann. Durch eine Anweisung zum Vernichten einer Task wird der Task-Kontroll-Block wieder freigegeben und kann z. B. bei der Erzeugung einer weiteren Task erneut verwendet werden.

Wie bei anderen Programmgrößen kann man auch bei Tasks zwischen Konstanten und Variablen unterscheiden. Während Taskkonstanten bei ihrer Erzeugung die auszuführende Anweisungsfolge, das Segment der Task, fest zugeordnet wird, kann eine Taskvariable während des Programmablaufs zur Ausführung verschiedener Segmente eingesetzt werden, die ihr beispielsweise bei der Ausführung von Start-Anweisungen zugewiesen werden.

Der Grund für die Einführung von Taskvariablen ist jedoch von dem, der zur Einführung arithmetischer Programmvariablen geführt hat, verschieden. Während beispielsweise die Programmierung eines Iterationsverfahrens zur Berechnung von Näherungswerten bei vorgegebener Ergebnisgenauigkeit nur unter Verwendung von Programmvariablen möglich ist, die während des Programmablaufs die bei einem Iterationsschritt berechneten Werte aufnehmen, dienen Taskvariable zur ökonomischen Verwertung von Rechenzeit und Speicher-

bedarf bei der Erstellung großer Automationsprogramme. Ist nämlich bei der Erstellung eines Automationsprogramms bekannt, daß bestimmte Anweisungsfolgen C_1, ..., C_n nie zeitlich parallel zueinander, aber parallel zu anderen Anweisungsfolgen auszuführen sind, so kann der Speicherplatz für Task-Kontroll-Blöcke klein gehalten werden, wenn für die Ausführung von C_1, ..., C_n eine Taskvariable vorgesehen wird. Die Rechenzeit wird u. a. verkürzt, weil das bei der Einrichtung eines Task-Kontroll-Blocks auszuführende Speicherplatz-Zuteilungsprogramm nur einmal durchlaufen werden muß.

Die Angabe von Parametern zur Organisation von Tasks (Priorität der Task, Startbedingungen, u. s. w.) ist bei der Erzeugung von Taskvariablen nicht sinnvoll, da diese ja für die Ausführung verschiedener Segmente herangezogen werden. Dagegen ist es möglich, solche Parameter bei der Erzeugung von Taskkonstanten anzugeben. In den einfachen prozeßorientierten Programmiersprachen wie INDAC8 und PAS1, die nur Taskkonstanten vorsehen, werden die meisten Organisationsparameter einer Task bei ihrer Erzeugung zugeordnet. In Programmiersprachen, die wie PEARL sowohl Taskkonstanten als auch Taskvariable zulassen, ist es zwecks möglichst einheitlicher Gestaltbarkeit der auf Tasks wirkenden Steueranweisungen zweckmäßig, organisatorische Parameter in die Steueranweisungen aufzunehmen.

Außer definitorischen Anweisungen für die Erzeugung und Vernichtung von Tasks sollte zur Optimierung der Betriebsmittelvergabe auch die Definition von "parallel ausführbaren" Anweisungen ermöglicht werden. Obwohl erst hierdurch einer der wesentlichen Vorteile der Aufteilung eines Automationsprogramms in Equipment- bzw. Systemteil und in Problemteil genutzt werden kann, fehlen solche Anweisungen in den meisten prozeßorientierten Programmiersprachen (z. B. auch in PEARL). Die einzige Programmiersprache, in der die Betriebsmittelvergabe in dieser Weise optimiert werden kann, ist m. W. PAS1.

b) Anweisungen zur Steuerung von Tasks.

Nach der Erzeugung einer Task befindet sich diese im Zustand "bekannt" und kann vermittels der in Abschnitt 2.5 schon beschriebenen Steueranweisungen in die Zustände "ablaufbereit" und "zurückgestellt" überführt werden. Nach jeder Ausführung des Segments oder bei Ausführung einer STOP-Anweisung kehrt die Task in den Zustand "bekannt" zurück.

Für die Synchronisation von Tasks, d. h. für die ereignisabhängige Ausführung von Steueranweisungen und Synchronisieranweisungen, werden bei der Erstellung von Automationsprogrammen folgende Synchronisierbedingungen benötigt:

- Zeitbedingungen
  Zeitbedingungen werden dazu verwandt, um den Start oder die Ablaufgeschwindigkeit von Tasks im Prozeßrechensystem an den Ablauf von Tasks außerhalb des Rechensystems anzupassen und setzen die Kenntnis des zeitlichen Verhaltens der Abläufe in technischen Prozessen bzw. von kommerziellen Vorgängen voraus.
  Typische Beispiele sind:

  - Zyklischer Start einer Task zur Überwachung eines technischen Prozesses. Die Zeitspanne zwischen zwei aufeinander folgenden Abläufen der Überwachungstask wird abhängig von der maximalen Änderungsgeschwindigkeit der charakteristischen Prozeßgrößen so gewählt, daß Fehlabläufe im technischen Prozeß rechtzeitig erkannt werden.
  - Unterbrechung des Ablaufs einer Task für eine bestimmte Dauer oder bis zum Erreichen einer Uhrzeit, um Einstellvorgänge in einem technischen Prozeß oder hinreichend große Änderungen von Meßwerten (z. B. Zählerstände für Verbrauchsberechnungen) abzuwarten.
  - Ausführen des Segments oder von Segmentteilen einer Task innerhalb vorgegebener Ausführungszeiten, um die Kopplung eines Prozeßrechensystems an einen technischen Prozeß in bestimmten Prozeßzuständen zu erzwingen. Aus der jeweils einzuhaltenden Ausführungszeit kann die Wichtigkeit einer Task abgeleitet werden (vergl. Kapitel 3).

- Synchronisierung über Botschaften.
  Synchronisierbedingungen, deren Erfüllung vom Betriebssystem nicht selbstständig erkannt werden kann, sind erforderlich, um Tasks, deren Ablauf von stochastischen Ereignissen abhängt oder deren Schachtelung sich dynamisch einstellt, aufeinander abzustimmen.

  Für die Synchronisation von Tasks innerhalb eines Rechensystems werden Synchronisationsvariable wie Zustandsvariable [15,16], Semaphorevariable (vergl. Abschnitt 1.2), Ereignisvariable (siehe Abschnitt 2.1) und BOLT-Variable [18] eingesetzt. Die Synchronisation von Tasks im Prozeßrechensystem mit dazu externen Tasks wird über Flags (Statusangaben) und Unterbrechungssignale ausgeführt.

Zur Definition und logischen Verknüpfung von Ereignissen werden in prozeßorientierten Programmiersprachen zahlreiche Sprachformen geboten. In der Regel sind in Synchronisierbedingungen Uhrzeitereignisse, Dauerereignisse, Unterbrechungsereignisse, Semaphoreereignisse und sinnvolle Kombinationen solcher Ereignisse spezifizierbar. Die sinnvollen Kombinationen sind in den verschiedenen Sprachbeschreibungen ausführlich dargelegt (vergl. insbesondere [16], Kap. 3, Abschnitt 4 und [18], Abschnitt 4.1.3).

Außer den in Abschnitt 2.5 beschriebenen Steueranweisungen sind zur bequemen Programmierung der zeitweisen Unterbrechung einer Task Steueranweisungen einsetzbar, die sich durch Spezialisierung der in Abb. 1.2 dargestellten Synchronisationsmethode ergeben. Wegen der Häufigkeit, mit der Anweisungen zur Unterbrechung von Tasks bei der Erstellung von Automationsprogrammen eingesetzt werden müssen, wurden entsprechende Ausdrucksmittel schon vor der Entwicklung von prozeßorientierten Programmiersprachen in Makroassemblern für die Prozeßprogrammierung vorgesehen(vergl. z. B. [43], S.238). Fehlt in Abb. 1.2 die Anweisungsfolge zwischen der RELEASE- und der REQUEST-Anweisung, so kann der Sachverhalt auch durch "AFTER 15 SEC RESUME;" oder durch "DELAY 15 SEC;" dargestellt werden, d. h. der Ablauf der Task wird unterbrochen und nach 15 Sekunden fortgesetzt.

Zu beachten ist, daß durch die Anweisungsfolge:

```
AFTER 15 SEC CONTINUE;
SUSPEND;
```

nicht der gleiche Sachverhalt dargestellt wird. Wird nämlich im Mehrprogrammbetrieb die Task, die diese Anweisungsfolge enthält, nach Ausführung der bedingten Fortsetzungsanweisung durch das Betriebssystem für mehr als 15 Sekunden in den Zustand "ablaufbereit" gebracht, so entsteht eine Systemverklemmung (deadlock).

c) Synchronisieranweisungen.

Synchronisieranweisungen werden benötigt, um dem Betriebssystem Ereignisse zur Kenntnis zu bringen (Botschaften) oder um Synchronisierbedingungen aufzuheben und wieder wirksam zu machen. Typische Beispiele für Botschaften sind:

- o RELEASE-Anweisung für Semaphorevariable,
- o FREE- und LEAVE-Anweisungen für BOLT-Variable und
- o Anweisungen zum Triggern von Unterbrechungssignalen.

Synchronisieranweisungen, mit denen Synchronisierbedingungen aufgehoben und wieder wirksam gemacht werden können, sind z. B. Anweisungen zum Sperren und Entsperren von Unterbrechungseingängen.

Bemerkenswert ist, daß in den bekannten prozeßorientierten Programmiersprachen ereignisabhängige Synchronisieranweisungen nicht zugelassen sind. So muß beispielsweise die in Abb. 1.2 verwendete Botschaft: "AFTER 15 SEC RELEASE SYNCHRO;" häufig unter Verwendung einer eigens für diesen Zweck einzuführenden Task realisiert werden. Ebenso fehlen die zur Untersuchung der Programmstabilität wichtigen bedingten Triggeranweisungen, durch die z. B. Folgen von Unterbrechungsereignissen vorgebbar wären.

3. Ein neues Verfahren zur Steuerung der Betriebsmittelvergabe in Prozeßrechensystemen.

3.1 Das Problem der Prioritätsvergabe in den bisher publizierten prozeßorientierten Programmiersprachen.

Ein wesentlicher Vorteil von Prozeßrechensystemen gegenüber konventionellen Automatisierungssystemen besteht in der Unabhängigkeit der Geräte-Komponenten von den Automatisierungsfunktionen. Da die Geräte in der Regel nur elementare Funktionen (z. B. Wandeln von Analogwerten in Digitalwerte) ausführen, lässt sich ein Prozeßrechner-Gerätesystem i. a. kosteneffektiv aus einer relativ geringen Anzahl von Geräten aufbauen, die bei der Realisierung aller Automatisierungsfunktionen eingesetzt werden.

Obwohl auch im kommerziellen Mehrprogrammbetrieb die Betriebsmittel (Zentraleinheiten, Drucker, u. s. w.) von allen Tasks gemeinsam benutzt werden, können Anweisungsfolgen zur Lösung kommerzieller oder technisch-wissenschaftlicher Aufgaben unabhängig voneinander erstellt und als Tasks mit beliebig vorgebbaren Prioritäten parallel bzw. quasiparallel zueinander ausgeführt werden. Die durch die Tasks zu erbringenden Ergebnisse (Leistungen der Tasks) sind dabei von den Prioritäten unabhängig, die den Tasks zugeordnet werden.

Die Unabhängigkeit von Leistung und Priorität resultiert daraus, daß im konventionellen Mehrprogrammbetrieb die von Tasks zu erbringenden Ergebnisse i. a. nicht davon abhängen, wann Tasks ausgeführt werden. Eine Task, die beispielsweise zur Lösung einer Differentialgleichung dient, wird die gleichen Ergebnisse liefern, gleichviel ob sie morgends oder abends ausgeführt wird. Die Leistungen von Tasks zur Lösung kommerzieller und technisch-wissenschaftlicher Aufgaben sind allein durch die Rechenalgorithmen bestimmt, deren Ausführung sie darstellen.

Ein Prozeßrechensystem ist ein "offenes" Rechensystem, das erst durch Einbezug des technischen Prozesses, d. h. durch Übergang zum Prozeßautomatisierungssystem, abgeschlossen werden kann. Die Leistung einer Task, die zur Automatisierung eines technischen Prozesses dient, ist durch die Automatisierungsfunktion [1] bestimmt, die sowohl den Rechenalgorithmus als auch die Ausführungsbestimmungen für diesen vorgibt. Durch die Ausführungsbestimmungen wird der Ablauf von Tasks im Prozeßrechensystem mit dem Ablauf von Tasks in anderen Teilen des Prozeßautomatisierungssystems synchronisiert.

Zur Programmierung der Ausführungsbestimmungen können die in Kapitel 2 beschriebenen Anweisungen zur Definition und Handhabung von Tasks eingesetzt werden. Dabei kann nur festgelegt werden, wann eine Task einen der Zustände

"bekannt", "ablaufbereit" und "zurückgestellt" annehmen soll. Übergänge zwischen diesen Zuständen und dem Zustand "ablaufend" werden vom Betriebssystem (Organisator) vollzogen, das im Konfliktfall, d. h. wenn mehrere Tasks dasselbe Betriebsmittel anfordern, dieses Betriebsmittel an die wichtigste ablaufbereite Task vergibt.

Anhand von Gegenbeispielen soll im folgenden gezeigt werden, daß die bislang verwendete Form der Prioritätsangabe, die für kommerzielle Anwendungen aus den oben genannten Gründen hinreicht, nicht mit der in Kapitel 1 begründeten Forderung nach unabhängiger Programmierbarkeit von Automatisierungsfunktionen verträglich ist (Modultechnik). Die Beispiele zeigen, daß weder absolute noch relative Prioritätsnummern für die unabhängige Erstellung von Automationsprogrammen hinreichen.

Beispiel 1:

Vorgelegt seien zwei Automatisierungsfunktionen A_1 und A_2 mit den in Tabelle 3.1 und Tabelle 3.2 angegebenen Ausführungsbestimmungen. A_1 und A_2 sollen unabhängig voneinander programmiert werden und gleichzeitig bzw. quasigleichzeitig in einem Prozeßrechensystem mit einer Zentraleinheit ausgeführt werden. In den Tabellen bedeuten Δt_{Amax} die maximal

Taskname	Δt_{Amax}	t_{Start}	Hierarchie
R_1	1 sec	alle 1.5 sec	Haupttask
SR_1	0.3 sec	sofort nach Start von R_1	Subtask von R_1

Tabelle 3.1 Ausführungsbestimmungen von A_1

Taskname	Δt_{Amax}	t_{Start}	Hierarchie
R_2	1.2 sec	alle 1.5 sec	Haupttask
SR_2	0.9 sec	sofort nach Start von R_2	Subtask von R_2

Tabelle 3.2 Ausführungsbestimmungen von A_2

zugelassene Ausführungszeit -d. i. die Zeitspanne, in der eine Task ausgeführt sein soll- und t_{Start} gibt an, wann eine Task gestartet

werden soll. Aus Δt_{Amax} geht hervor, wie schnell das Segment einer Task ausgeführt werden muß, damit das Automationsprogramm mit Vorgängen außerhalb des Prozeßrechensystems synchron läuft.

a) Nichtverwendbarkeit fester Systemprioritäten:
Bei unabhängiger Programmierung der Automatisierungsfunktionen A_1 und A_2 lassen sich die in Tabelle 3.1 und Tabelle 3.2 angegebenen Ausführungsbestimmungen realisieren, wenn bei der Wahl der Prioritätszahlen P_s die folgenden beiden Ungleichungen eingehalten werden:

$$P_s(R_i) > P_s(SR_i); \quad i = 1, 2.$$

Damit alle vier Tasks unter Einhaltung der Ausführungsbestimmungen parallel ausführbar sind, muß zusätzlich gelten:

$$P_s(R_2) > P_s(R_1) > P_s(SR_2) > P_s(SR_1) .$$

Dieser Ungleichung kann bei der Programmierung von A_1 bzw. A_2 nur Rechnung getragen werden, wenn dabei die Ausführungsbestimmungen von A_2 bzw. A_1 schon bekannt sind.

b) Nichtverwendbarkeit relativer Prioritätszahlen:
Auch die Verwendung relativer Prioritätszahlen führt bei dem vorliegenden Beispiel nicht zum Ziel.

Um die Ausführungsbestimmungen von A_1 und A_2 zu realisieren, müssen die relativen Prioritätszahlen P_r der Subtasks den Ungleichungen:

$$P_r(SR_i) < 0; \quad i = 1, 2$$

genügen, d. h. die Prioritäten der Subtasks sind größer als die der Haupttasks.

Es gibt aber keine Wahl der Prioritäten $P_r(R_i)$; i = 1, 2, die zu der gewünschten Ausführungsreihenfolge: SR_1, SR_2, R_1, R_2 führen würde. Ist nämlich $P_r(R_1) < P_r(R_2)$, so werden die Tasks in der Reihenfolge SR_1, R_1, SR_2, R_2 ausgeführt. Falls $P_r(R_1) > P_r(R_2)$ gilt, ergibt sich die Reihenfolge: SR_2, R_2, SR_1, R_1. Im Fall $P_r(R_1) \equiv P_r(R_2)$ ist die Reihenfolge undefiniert.

Die unabhängige Programmierung von A_1 und A_2 wäre danach nur unter Verwendung variabler Systemprioritäten möglich. Man könnte dabei, ähnlich wie z. B. in PROCOL [17] vorgesehen, so vorgehen, daß bei der Programmierung die Prioritätsangaben in Form von Ganzzahlvariablen angegeben werden, die beim Systemstart per Bedienung initialisiert werden. Daß auch die Verwendung von Prioritätsvariablen i. a. nicht zum Ziel führt, zeigt das nächste Beispiel.

Beispiel 2:

Gegeben seien zwei Automatisierungsfunktionen A_1 und A_2, die unabhängig voneinander programmiert werden sollen. Nach ihrer Erstellung sollen die Automationsprogramme quasiparallel in einem Prozeßrechensystem mit einer Zentraleinheit ausgeführt werden. In Tabelle 3.3 sind die Ausführungsbestimmungen für A_1 und A_2 aufgeführt.

Taskname	Δt_{Amax}	t_{Start}
Q_1	120 msec	alle 120 msec
Q_2	80 msec	alle 80 msec

Tabelle 3.3 Ausführungsbestimmungen von A_1 und A_2

Bezeichnet man mit Δt_{Lmax} die größte Laufzeit einer Task -d. i. die Zeitspanne, die eine Task im ungünstigsten Fall für ihren Ablauf benötigt- und nimmt zur Vereinfachung der Diskussion an, daß jede der Tasks Q_i; i = 1, 2 während ihres Ablaufs ständig die Zentraleinheit belegt, so lassen sich drei Fälle unterscheiden:

Fall a): $$\Delta t_{Lmax}(Q_1) + \Delta t_{Lmax}(Q_2) \leq 80 \text{ msec}$$

Die Ausführungsbestimmungen für Q_1 und Q_2 werden eingehalten, wenn die Prioritätszahlen P_s beim Urstart gemäß:

$$P_s(Q_1) > P_s(Q_2)$$

gewählt werden.

In Abb. 3.1 ist die Schachtelung von Q_1 und Q_2 für $\Delta t_{Lmax}(Q_1)$ = 30 msec und $\Delta t_{Lmax}(Q_2)$ = 20 msec dargestellt. Innerhalb der

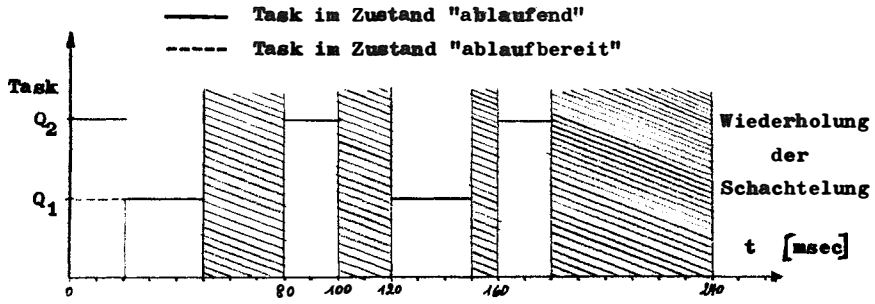


Abb. 3.1 Schachtelung von Q_1 und Q_2 im Fall a).

schraffierten Gebiete befinden sich beide Tasks im Zustand "bekannt"; die Zentraleinheit ist unbeschäftigt.

Fall b): $80 \text{ msec} < \Delta t_{Lmax}(Q_1) + \Delta t_{Lmax}(Q_2)$

$$2.\Delta t_{Lmax}(Q_1) + 3.\Delta t_{Lmax}(Q_2) \leq 240 \text{ msec}$$

Die Ausführungsbestimmungen lassen sich nicht durch Zuweisung von Prioritätszahlen beim Urstart realisieren. In Abb. 3.2 ist die Ausführungssequenz für $\Delta t_{Lmax}(Q_1) = 50$ msec und $\Delta t_{Lmax}(Q_2) = 40$ msec dargestellt.

Zum Zeitpunkt 0 sind beide Tasks im Zustand "ablaufbereit". Da Q_2 früher als Q_1 beendet sein soll, wird zunächst Q_2 ausgeführt, d. h. $P_s(Q_1) > P_s(Q_2)$. Danach wird Q_1 zum Ablauf gebracht.

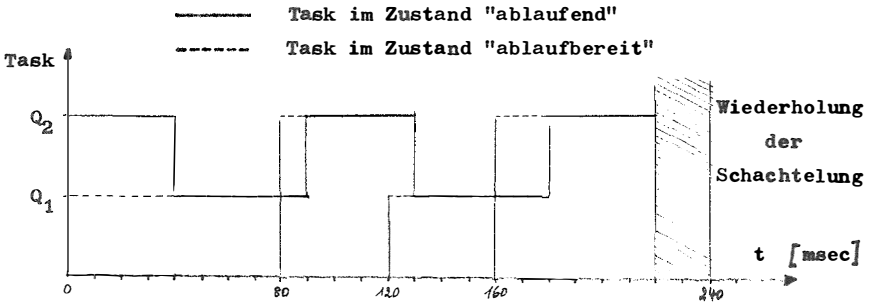


Abb. 3.2 Schachtelung von Q_1 und Q_2 im Fall b).

Nach 80 msec wird Q_2 wieder in den Zustand "ablaufbereit" überführt. Da Q_1 noch nicht beendet ist und spätestens nach 120 msec ausgeführt sein soll, muß Q_2 warten bis Q_1 ausgeführt ist, d. h. $P_s(Q_1) < P_s(Q_2)$.

Fall c): $2.\Delta t_{Lmax}(Q_1) + 3.\Delta t_{Lmax}(Q_2) > 240 \text{ msec}$

Die Ausführungsbestimmungen sind nicht realisierbar (Systemüberlastung).

In Abb. 3.3 sind die Bereiche, in denen die Ablaufsteuerung unterschiedlich zu handhaben ist, dargestellt. Unterhalb der Geraden $\Delta t_{Lmax}(Q_1) + \Delta t_{Lmax}(Q_2) = 80$ msec (einfach schraffierter Bereich) ist die Prioritätssteuerung gemäß Fall a) möglich. Im Bereich zwischen den Geraden $\Delta t_{Lmax}(Q_1) + \Delta t_{Lmax}(Q_2) = 80$ msec und $2.\Delta t_{Lmax}(Q_1) + 3.\Delta t_{Lmax}(Q_2) = 240$ msec (doppelt schraffierter Bereich) kann der zeitgerechte Ablauf der beiden Tasks nur durch Änderung der Automationsprogramme (Einfügen von Steueranweisungen in die Tasksegmente) herbei-

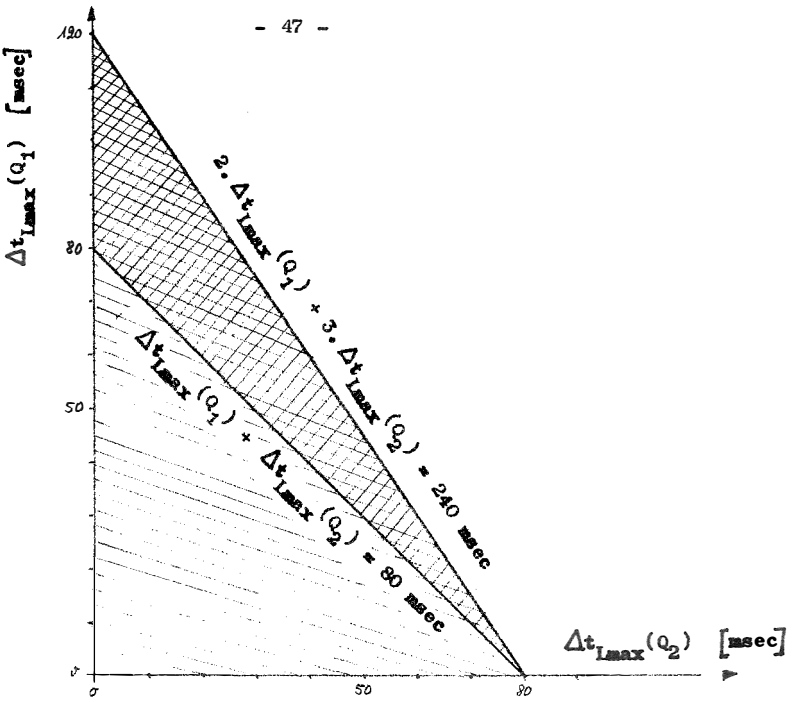


Abb. 3.3 Zur Abhängigkeit der Ablaufsteuerung von der Laufzeit bei Steuerung über Prioritätsnummern.

geführt werden. Außerhalb der schraffierten Bereiche ist das Rechensystem überlastet.

Es ist überraschend, daß schon bei derart einfachen Problemstellungen die Automationsprogramme für unabhängige Automatisierungsfunktionen nicht unabhängig voneinander erstellt werden können. Um beispielsweise zu gewährleisten, daß die Ausführungsbestimmungen in Beispiel 2, Fall b) eingehalten werden, muß ein Prioritätswechsel oder ein gegenseitiges Aussperren über Synchronisationsvariable zwischen Q_1 und Q_2 erfolgen. Das kann nur durch Steueranweisungen realisiert werden, deren Art, Anzahl und Ort im Automationsprogramm erst bekannt ist, wenn außer den Ausführungsbestimmungen auch die Laufzeiten aller gleichzeitig auszuführenden Tasks bekannt sind.[1)]

---

1) Bei der Diskussion der vorangegangenen Beispiele wurde angenommen, daß durch das Organisationsprogramm die im Zustand "ablaufend" befindliche Task in den Zustand "ablaufbereit" versetzt wird, sobald eine höherpriore Task ablaufbereit wird. Die in Abb. 4.2 dargestellte Schachtelung lässt sich auch erreichen, wenn eine Task unabhängig von der Aktivierung höherpriorer Tasks solange im Zustand "ablaufend" verbleibt, bis sie ausgeführt ist. Durch geringfügige Erweiterung des Beispiels lässt sich zeigen, daß auch diese Organisationsstrategie nicht hinreichend ist (vergl. Beispiel 2, Abschnitt 4.2).

Man erkennt daraus, daß mittels der bisher vorgeschlagenen Prozeßprogrammiersprachen die modulare Programmerstellung nur bei schwacher Auslastung eines Rechensystems möglich ist. Ist diese Voraussetzung nicht gegeben oder wird durch Erweiterung schon erstellter Programmsysteme aufgehoben, so entstehen auch bei Verwendung der bisher vorgeschlagenen höheren Prozeßprogrammiersprachen änderungsunfreundliche monolithische Programmsysteme, deren Aufbau und Wirkungsweise, ähnlich wie bei der Programmierung in Assemblersprachen, installationsabhängig von einem Spezialistenteam konzipiert und nur von diesem verstanden wird.

3.2 Anforderungen an das Vokabular zur Steuerung der Betriebsmittelvergabe aus der Sicht des Anwenders.

Die Betrachtungen in Kapitel 2 und die Ergebnisse des vorangegangenen Abschnitts legen die Vermutung nahe, daß bei der Entwicklung von prozeßorientierten Programmiersprachen die Funktionen des Organisators nicht durchwegs in zweckdienlicher Weise festgelegt wurden.

Wie im folgenden gezeigt wird, liegt der Grund für die erläuterten Schwierigkeiten in der nicht problemgerechten Form, in der die Wichtigkeit von Tasks bisher zum Ausdruck gebracht wird. Für die Prioritätsangabe werden nicht die bei der Aufstellung von Automatisierungsfunktionen verwendeten problemspezifischen Parameter herangezogen, sondern es wird, um eine naheliegende Implementierung der Ablauforganisation zu ermöglichen, eine mehr oder minder willkürliche Abbildung dieser Parameter auf die Menge der ganzen Zahlen ausgeführt. Da diese Abbildung bei der Programmerstellung erfolgen muß, ist sie i. a. erst möglich, wenn die Ausführungsbestimmungen aller Automatisierungsfunktionen und weitere installationsabhängige Parameter (z. B. Laufzeiten von Tasks, Bereich für Prioritätsnummern) bekannt sind.

Welche Angaben für die Wichtigkeit von Tasks relevant sind, soll im folgenden anhand eines einfachen Steuerungsproblems erläutert werden. Dazu wird das in Abb. 3.4 dargestellte Prozeßrechensystem betrachtet. Nach Start des

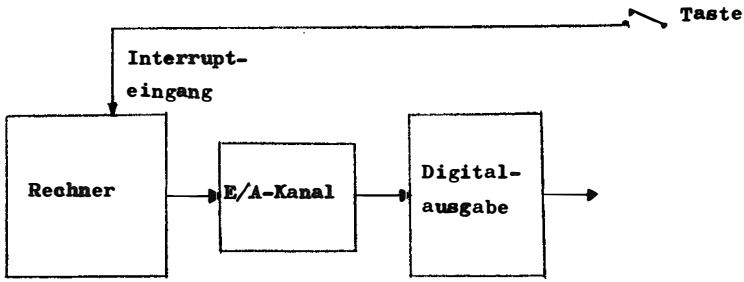
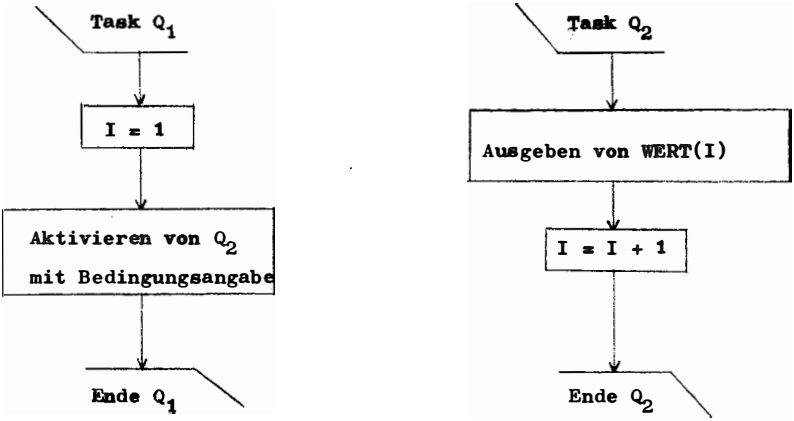


Abb. 3.4 Aufbau eines einfachen Prozeßrechensystems.



a) Automationsalgorithmen

Taskname	Δt_{Amax}	t_{Start}
Q_1	10 sec	durch Interrupt
Q_2	0.05 sec	durch Q_1 mit Bedingung: ALL 0.5 SEC DURING 8 MIN 19.6 SEC

b) Ausführungsbestimmungen

Abb. 3.5 1. Lösung des Steuerungsproblems

des Automationsprogramms durch Bedienen der Taste sollen die in einer Liste WERT gespeicherten 1000 Stellwerte in Abständen von 0.5 sec über die Digitalausgabe ausgegeben werden.

Die Güte der Steuerung wird i. a. davon abhängen, wie genau die zeitlichen Abstände zwischen den einzelnen Steuerausgaben eingehalten werden. Damit die Steuerung mit einer bestimmten Mindestgenauigkeit ausgeführt wird, muß die Automatisierungsfunktion Angaben enthalten, aus denen hervorgeht, welche zeitlichen Toleranzen bei den Stellwertausgaben zugelassen sind. Für das Beispiel sei angenommen, daß ein Stellwert spätestens 0.05 sec nach Eintritt eines Ausgabezeitpunkts ausgegeben sein soll.

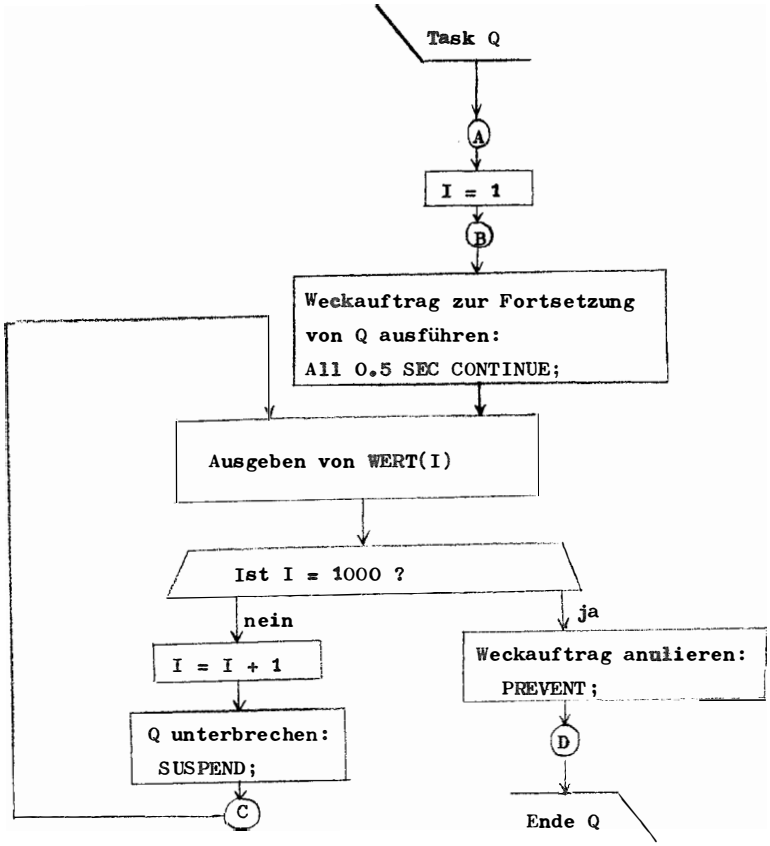
In Abb. 3.5 und Abb. 3.6 sind zwei Lösungen des Steuerungsproblems dargestellt.

Bei der in Abb. 3.5 angegebenen Lösung werden zwei Tasks Q_1 und Q_2 verwendet. Q_1 wird bei Drücken der Taste durch Interrupt gestartet und führt eine Startanweisung für Q_2 mit Bedingungsangabe: ALL 0.5 SEC DURING 8 MIN 19.6 SEC aus. Durch Q_2 werden danach die 1000 Stellwerte in Abständen von 0.5 sec ausgegeben. Während $\Delta t_{Amax}(Q_1)$ = 10 sec als Reaktionszeit auf das Drücken der Taste häufig willkürlich festgelegt werden kann, bestimmt $\Delta t_{Amax}(Q_2)$ = 0.05 sec die Güte der Steuerung. Durch Einhalten der maximal zugelassenen Ausführungszeit für Q_2 wird dafür gesorgt, daß die Stellwerte "rechtzeitig" ausgegeben werden.

Offenbar geht in dem Beispiel aus $\Delta t_{Amax}(Q_1)$ und $\Delta t_{Amax}(Q_2)$ die Wichtigkeit der Tasks Q_1 und Q_2 hervor. Das Beispiel zeigt nochmals sehr deutlich, daß sich bei Verwendung von Prioritätsnummern ohne Kenntnis der weiteren parallel auszuführenden Automationsprogramme keine Prioritätsaussage machen lässt, durch welche die Güte der Steuerung gewährleistet werden könnte. Dagegen wäre bei Verwendung von Δt_{Amax} als problemgerechter Form der Prioritätsangabe die unabhängige Programmierung von Automatisierungsfunktionen möglich.

Wie die in Abb. 3.6 dargestellte zweite Lösung des Steuerungsproblems zeigt, wäre die maximal zulässige Ausführungszeit einer Task nur bei einfachen linearen Automationsalgorithmen hinreichend. Ist dies, wie in Abb. 3.6a nicht der Fall, so müssen die Ausführungsbestimmungen für den Automationsalgorithmus detaillierter beschrieben werden.

Wie aus Abb. 3.6a ersichtlich ist, sind für die zeitgerechte Ausführung der Task Q vier Wege im Ablaufdiagramm wesentlich. Anfang und/oder Ende dieser Wege sind durch die in das Diagramm eingefügten Identifikatoren Ⓐ, Ⓑ, Ⓒ, Ⓓ kenntlich gemacht. Zur Vereinfachung wurde in Abb. 3.6b die Schreibweise: (A,B) verwendet, die als "Ablaufweg von Ⓐ nach Ⓑ " zu lesen ist.



a) Automationsalgorithmus

Taskname	t_{Start}	Ablaufweg	Δt_{Amax}
Q	durch externen Interrupt	(A,B)	10 sec
		(B,C)	0.05 sec
		(C,C)	0.05 sec
		(C,D)	0.05 sec

b) Ausführungsbestimmungen

Abb. 3.6 2. Lösung des Steuerungsproblems

Die maximal zugelassene Ausführungszeit für (A,B): $\Delta t_{Amax}(A,B)$ ist wie bei Lösung 1 durch die Reaktionszeit von 10 sec gegeben. Nach Ausführung der Weckbeauftragung muß die erste Stellwertausgabe spätestens nach 0.05 sec erfolgt sein, damit der zeitliche Abstand zwischen der ersten und der zweiten Ausgabe eingehalten wird ($\Delta t_{Amax}(B,C) = 0.05$ sec).

Die Schleife (C,C) wird bei jedem Weckereignis durchlaufen bis alle Stellwerte ausgegeben sind. Um die geforderte Genauigkeit einzuhalten, muß $\Delta t_{Amax}(C,C) = 0.05$ sec gesetzt werden.

Schließlich muß für (C,D) ebenfalls $\Delta t_{Amax} = 0.05$ sec angenommen werden, damit auch der letzte Stellwert zeitgerecht ausgegeben wird.

Wie aus dem Beispiel ersichtlich ist, wird von der Anwendung her gesehen die Wichtigkeit einer Task am zweckmäßigsten durch Zeitangaben dargestellt, aus denen hervorgeht, wie "schnell" das Segment einer Task oder Teile eines Segments ausgeführt werden müssen. Durch die Angabe von Ausführungszeiten werden die i. a. mit unterschiedlicher Geschwindigkeit ablaufenden Task innerhalb und außerhalb eines Prozeßrechensystems miteinander synchronisiert.

3.3 Dead line scheduling; dynamische Prioritätsvergabe.

Das in Abschnitt 3.2 diskutierte Beispiel legt für die Prioritätsbestimmung von Tasks folgendes Prinzip nahe (Prinzip der kürzeren Restzeit):

> Zu einem Zeitpunkt t seien n ablaufbereite Tasks Q_i; i = 1, 2, ..., n; n = 2, 3, ... vorgelegt. Weiter sei $\Delta t_{Rest}(Q_i)$ der zum Zeitpunkt t noch verbleibende Rest der vorgegebenen Ausführungszeit für den aktuellen Ablaufweg in Q_i.
>
> a) Gilt für die Restzeiten zweier Tasks zum Zeitpunkt t:
>
> $$\Delta t_{Rest}(Q_k) < \Delta t_{Rest}(Q_l); \quad k,l \in \{1, \ldots, n\},$$
>
> so ist die Ausführung des aktuellen Ablaufwegs von Q_k wichtiger als die Ausführung des aktuellen Ablaufwegs von Q_l, d. h. Q_k ist wichtiger als Q_l.
>
> b) Falls zum Zeitpunkt t:
>
> $$\Delta t_{Rest}(Q_k) \cong \Delta t_{Rest}(Q_l); \quad k,l \in \{1, \ldots, n\}$$
>
> gilt, so sind Q_k und Q_l gleichwichtig. Die aktuellen Ablaufwege von Q_k und Q_l dürfen in beliebiger Reihenfolge ausgeführt werden.

Man überzeugt sich leicht davon, daß aus dem angegebenen Prinzip die Schachtelung der Tasks Q_1 und Q_2 in Beispiel 2, Abschnitt 3.1 sowohl im Fall a)

als auch im Fall b) richtig abgeleitet werden kann und zwar

- ohne, daß bei der Programmierung von A_1 bzw. A_2 die Ausführungsbestimmungen von A_2 bzw. A_1 bekannt sein müssen und
- ohne, daß vor dem Urstart eine alle Tasks betreffende Prioritätsstaffelung oder gar Modifizierung von Tasksegmenten erforderlich ist.

Bei Einsatz eines aus dem angegebenen Prinzip folgenden Verfahrens für die Betriebsmittelvergabe ist auch bei der Erstellung von Automationsprogrammen eine Programmiertechnik möglich, die mit den im kommerziellen Anwendungsbereich entwickelten Techniken vergleichbar ist. Wie dort ist es bei Verwendung des angegebenen Prinzips auch bei der Erstellung von Prozeßautomationsprogrammen möglich,

- Automatisierungsfunktionen für voneinander unabhängige technische Prozesse unabhängig voneinander zu programmieren und
- zu schon ablaufenden Programmsystemen ohne deren Änderung weitere Automationsprogramme hinzuzubringen.

Bevor auf die Konsequenzen des Prinzips für die Programmierung eingegangen wird, soll noch ein Einwand entkräftet werden, der gegen das Prinzip vorgebracht werden könnte. Da in einem auf dem Prinzip aufbauenden Verfahren die Wichtigkeit von Tasks stets dynamisch, d. h. zur Laufzeit, ermittelt werden muß, ist zu erwarten, daß im Vergleich zur nummerngesteuerten Betriebsmittelzuteilung mehr Zeit für die Organisation von Tasks benötigt wird.

Gegen diesen Einwand ist vorzubringen, daß mehr Organisationszeit gerade dann verbraucht wird, wenn die Betriebsmittelzuteilung durch feste Taskprioritäten gesteuert werden könnte. Wie Fall a) von Beispiel 2 in Abschnitt 3.1 zeigt, ist das genau dann der Fall, wenn ohnedies genügend Rechenzeit für die Organisation verfügbar ist.

Bei starker Auslastung eines Rechensystems wird auch bei Steuerung über Prioritätsnummern mehr Organisationszeit verbraucht, die durch die vom Anwender in die Tasksegmente einzufügenden zusätzlichen Steueranweisungen hervorgerufen wird (vergl. Fall b), Beispiel 2, Abschnitt 3.1). Die Steuerung über Restzeiten dürfte in diesem Fall zu vergleichbaren Organisationszeiten führen.

Für die Anwendungsprogrammierung ergibt sich aus dem Prinzip eine Erhöhung der Sicherheit bei der Programmerstellung. Die Sicherheit wird einerseits dadurch erhöht, daß die bei der Prozeßanalyse gefundenen und/oder bei der Konzipierung von Prozeßautomatisierungssystemen festgelegten Ausführungsbestimmungen einfach in ein Automationsprogramm übertragbar sind. Andererseits entfällt die erst bei der Integration aller gleichzeitig auszuführen-

den Automationsprogramme durchführbare Prioritätszuteilung und gegebenenfalls Modifikation schon erstellter Programme; d. h. voneinander unabhängige Automationsprogramme sind auch unabhängig voneinander testbar.

Das bedeutet nicht, daß vor dem gemeinsamen Lauf aller parallel auszuführenden Automationsprogramme (Test des Gesamtsystems) eine Aussage darüber gewonnen werden kann, ob das Rechensystem überlastet wird, d. h. ob überhaupt genügend Rechenzeit zur Ausführung aller Automationsprogramme bereitsteht. Durch auf dem Prinzip der kürzeren Restzeit basierende Verfahren werden jedoch der Gesamttest und die für diesen bereitzustellenden Emulatorsysteme (siehe z. B. [44,45,46,47]) vereinfacht, weil

- o die für die Betriebsmittelzuteilung relevanten Parameter im Programm selbst dokumentiert sind und
- o weil nur noch überprüft werden muß, ob das Gesamtsystem im sog. "worst case" lauffähig ist.[1)]

Man darf daher erwarten, daß die aus dem angegebenen Prinzip ableitbaren Verfahren hinsichtlich der Betriebsmittelvergabe in Prozeßrechensystemen zu gleichguten Ergebnissen wie die bisher eingesetzten Verfahren führen werden, wobei die beschriebenen Nachteile der nummerngesteuerten Betriebsmittelzuteilung entfallen.

3.4 Sprachelemente für die dynamische Prioritätszuteilung.

Beim Entwurf von Sprachelementen für den Einsatz des neuen Verfahrens kann man verschiedene Wege einschlagen. Es ist einerseits möglich, wie in INDAC8 und PAS1 vorgesehen, die für die Betriebsmittelzuteilung relevanten Angaben zusammengefasst in einem eigenen Programmteil formulierbar zu machen.

---

1) Die Frage, ob eine vorgegebene Menge von Automationsprogrammen in einem Rechensystem gemeinsam lauffähig ist, kann bisher nur durch gemeinsames Ausführen der Automationsprogramme in der realen Umgebung oder durch Simulation bzw. Emulation beantwortet werden. Um hier eine Verbesserung zu erreichen, müssten einerseits die für die Entscheidungsfindung relevanten Informationen (z. B. die maximale Durchlaufzahl für alle Programmschleifen, Interrupthäufigkeiten, u. s. w.) vollständig im Quellprogramm dokumentierbar sein. Andererseits müssten neuartige Übersetzungsprogramme und Programmkomposer entwickelt werden, die aufgrund dieser Informationen und von Parametern der Rechensysteme (z. B. Befehlsausführungszeiten) ermitteln könnten, ob genügend Rechenzeit für die zeitgerechte Ausführung aller Programme verfügbar ist. Bis heute wurde die Entwicklung solcher Programmiersprachen, Übersetzer und Komposer nicht begonnen.

Andererseits kann man, wie z. B. von PROCOL und PEARL vorgezeichnet, die Steuerangaben an den Stellen in der Programmniederschrift ermöglichen, an denen sie ausgeführt werden sollen.

Welche Methode zur Beschreibung der Ausführungsbestimmungen von der Anwendung her gesehen, die zweckmäßigere ist, kann nur die Erfahrung zeigen. Der Verfasser ist der Meinung, daß die zuletzt angeführte Darstellungsweise dem gegenwärtigen Stand der Prozeßautomatisierungstechnik am besten angepasst ist.[1)]

Im folgenden wird eine Erweiterung der Syntax und Semantik von PEARL beschrieben, mit der das Prinzip der kürzeren Restzeit eingesetzt werden kann. Dabei wird die im PEARL-Report [18] verwendete Syntaxnotation und es werden, soweit möglich, die dort eingeführten metasprachlichen Bezeichnungen verwendet.

Bei der Erweiterung von PEARL wird davon ausgegangen, daß auch in Prozeßrechensystemen häufig Programme ausgeführt werden, deren Ablauf keinen oder nur sehr schwachen zeitlichen Einschränkungen unterliegt. Beispielsweise braucht für Programme, mit denen täglich einmal Zählerstände gelesen, Verbrauchsberechnungen durchgeführt und Rechnungen gedruckt werden (sog. Anwenderprogramme) i. a. keine Ausführungszeit vorgegeben werden. Moderne Prozeßrechensysteme können darüberhinaus gleichzeitig für die Prozeßlenkung und für kommerzielle Anwendungen eingesetzt werden.

Es ist deshalb sinnvoll, zwischen Vordergrund- und Hintergrundorganisation zu unterscheiden. Eine Task wird durch die Vordergrund- bzw. Hintergrundorganisation verwaltet, wenn die Wichtigkeit für den aktuellen Ablaufweg durch eine Ausführungszeit bzw. eine Prioritätsnummer festgelegt worden ist. Als Ablaufweg wird eine Folge von Anweisungen einer Task bezeichnet, deren Beginn die erste Anweisung der Task oder die auf eine Unterbrechungsanweisung mit Prioritätsangabe folgende Anweisung ist und deren Ende

---

1) Die Darstellung eines Automationsprogramms in verschiedenen Programmteilen wäre zweckmäßig, wenn für die Durchführung aller Automatisierungsvorhaben eine definierte Aufgabenteilung angegeben werden könnte. Wenn die Durchführung von Automatisierungsvorhaben in dieser Weise systematisiert werden könnte, wäre es sinnvoll, die Ergebnisse der mit den verschiedenen Teilaufgaben befassten Teams in jeweils einem eigenen Teil des Automationsprogramms darstellbar zu machen. Ansätze in dieser Richtung machen INDAC8, PAS1 und PEARL. Bei Einsatz dieser Programmiersprachen kann das mit der Konzipierung des Gerätesystems bzw. mit der Aufstellung der Automatisierungsfunktionen beschäftigte Team die Ergebnisse in jeweils einem eigenen Programmteil (Systemteil und Problemteil von PEARL) darstellen.

eine Anweisung zur Unterbrechung des Programmablaufs mit Prioritätsangabe (Ausführungszeit oder Prioritätsnummer) oder eine Anweisung zur Beendigung der Task ist.

Da der zur Darstellung der Ausführungszeit erforderliche Datentyp "Dauer" schon in PEARL vorgesehen ist, brauchen bei der Anpassung von PEARL an die neue Methode nur wenige Anweisungen zur Handhabung von Tasks modifiziert werden. Ersetzt man die Prioritätsangabe ([18], S. 113, Regel 4.4.1/1) durch:

priority-option ::= [EXECTIME duration-expression | PRIORITY integer-expression-seven] ,

und fordert, daß bei jeder Anweisung, die zum Start oder zur Fortsetzung einer Task führt oder führen kann, eine neue Priorität angegeben werden darf, so ergeben sich folgende Änderungen:[1)]

1. ON-Statement:

Regel 4.1.2/1 aus [18] wird ersetzt durch:

on-statement ::=
ON {,.. signal-designation ... } priority-option :
{unlabeled-statement | SYSTEM;}

Die in einer ON-Anweisung angegebene Anweisung gehört zu dem oder enthält den Ablaufweg, der bei Eintreffen eines durch signal-designation bezeichneten Signals durchlaufen werden soll. Um diesen Ablaufweg zeitgerecht ausführen zu können, wurde in der ON-Anweisung die Angabe einer Priorität zugelassen.

2. RESUME-Anweisung:

Regel 4.2.3/5 wird ersetzt durch:

resume-statement ::= schedule RESUME priority-option ;

Durch die Prioritätsangabe kann die Ausführungszeit eines auf die RESUME-Anweisung folgenden Ablaufwegs festgelegt werden.

3. REQUEST-Anweisung:

Die REQUEST-Anweisung ([18], Regel 4.3.2/4) wird ersetzt durch:

---

1) Dabei wurden die im Subset-Arbeitskreis beim Projekt "Prozeßlenkung mit DV-Anlagen" bei der GFK Karlsruhe beschlossenen Änderungen von PEARL [48] schon berücksichtigt.

request-operation ::=

REQUEST {,.. semaphore-designation ...} priority-option ;

Entsprechend lassen sich auch die BOLT-Operationen ([18] , Regel 4.3.3/1) RESERVE und ENTER erweitern.

3.5 Modell einer Ablauforganisation mit restzeitgesteuerter Vordergrund- und nummerngesteuerter Hintergrundverwaltung.

Die Implementierung der Sprachelemente für die Parallelprogrammierung in PEARL erfordert die Erstellung eines umfangreichen Organisationsprogramms. In dem vorliegenden Abschnitt soll ein Verfahren beschrieben werden, mit dem sich das Prinzip der kürzeren Restzeit -auch für Einprozessoranlagen- realisieren lässt.

Für die Implementierung der in Abschnitt 3.4 vorgeschlagenen PEARL-Erweiterungen ist eine Warteschlange für ablaufbereite Hintergrundtasks (Taskwarteschlange) und mindestens eine Warteschlange für die Zeitverwaltung erforderlich. In diese Warteschlangen werden die Task-Kontroll-Blöcke (im folgenden mit TKB abgekürzt) der ablaufbereiten und eingeplanten Tasks eingeordnet.

Obwohl es vom Prinzip her möglich wäre, sowohl die ablaufbereiten Vordergrundtasks als auch die Dauereinplanungen über eine einzige Zeitwarteschlange zu verwalten, ist es, um die Suche nach der wichtigsten ablaufbereiten Vordergrundtask zu beschleunigen, zweckmäßig, dazu eine eigene Zeitwarteschlange zu verwenden. Würde man Dauerereignisse und Restzeiten über eine einzige Warteschlange verwalten, so wären bei der Suche nach der wichtigsten ablaufbereiten Vordergrundtask die Zustandsangaben aller in die Zeitwarteschlange eingeordneten TKBs auszuwerten.

Alle Informationen, die für die Organisation von Tasks benötigt werden, sind in den zugeordneten TKBs notiert. U. a. enthält dieser Speicherplatz für folgende Angaben:

A1. Zustand der Task.

A2. Hinweise auf zwei TKBs, die ablaufbereiten Tasks gleicher Wichtigkeit zugeordnet sind.

A3. Hinweise auf zwei TKBs, die zum gleichen Startzeitpunkt eingeplanten Tasks zugeordnet sind.

A4. Priorität (Ausführungszeit oder Prioritätsnummer).

A5. Dauer zwischen zwei Dauerereignissen bei zyklischer Beauftragung.

A6. Segmentbeginn (Startadresse).

A7. Registerstände.

Der Taskzustand (vergl. Abschnitt 2.4) wird benötigt, um die Ausführbarkeit von Steueranweisungen zu überprüfen. Beispielsweise darf bei der Ausführung einer SUSPEND-Anweisung auf eine schon zurückgestellte Task kein Umsetzen der unter Punkt A2 angegebenen Zeiger erfolgen. Letzteres ist jedoch notwendig, wenn sich die Task im Zustand "ablaufbereit" oder "ablaufend" befindet. Über den Taskzustand kann außerdem festgestellt werden, ob und wann die in Abschnitt 3.2 an einem Beispiel erläuerten Programmodifikationen erforderlich sind bzw. wann das System überlastet ist.

Die unter Punkt A2 aufgeführten Hinweise werden benötigt, um den TKB einer im Zustand "ablaufbereit" befindlichen Task, die sich gemeinsam mit anderen ablaufbereiten Tasks um Betriebsmittel (in dem hier beschriebenen Fall die Zentraleinheit) bewirbt, gemäß ihrer Priorität in die Taskwarteschlange bzw. die Ablaufschlange für Vordergrundtasks einzuordnen.

Bei Beschränkung auf Systemprioritäten besitzt die Taskwarteschlange den in Abb. 3.7 dargestellten einfachen Aufbau. Bei der Erstellung der Abbildung

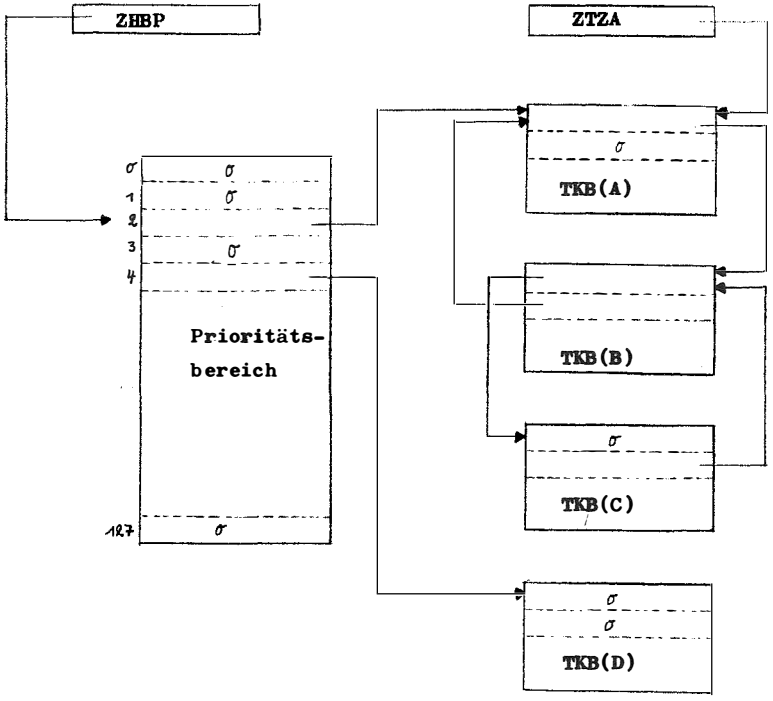


Abb. 3.7 Aufbau einer Taskwarteschlange bei nummerngesteuerter Ablauforganisation (Systemprioritäten).

würde angenommen, daß sich zu dem betrachteten Zeitpunkt die Tasks B, C, D mit Task-Kontroll-Blöcken TKB(B), TKB(C), TKB(D) im Zustand "ablaufbereit" befinden und daß die über TKB(A) verwaltete Task A gerade abläuft. IN ZTZA (Zeiger auf Task im Zustand "ablaufend"), wird die Adresse des TKB der im Zustand "ablaufend" befindlichen Task notiert.

Auf den TKB einer ablaufbereiten Task mit Priorität n wird im n-ten Element des Prioritätsbereichs hingewiesen. Sind mehrere Tasks gleicher Priorität ablaufbereit, so sind die zugeordneten TKBs verkettet. Die Vorwärts/Rückwärtsverzeigerung ist nötig, damit beim Umordnen eines TKBs in der Taskwarteschlange (Prioritätsänderung) oder beim Ausordnen eines TKBs aus der Warteschlange (Zurückstellen, Beenden, Umordnen in die Ablaufschlange für Vordergrundtasks) die Verzeigerung aktualisiert werden kann. Die Suche nach der wichtigsten ablaufbereiten Task wird über den Zeiger ZHBP (Zeiger auf höchste besetzte Priorität) beschleunigt, der auf das Element mit niedrigstem Index (Prioritätsnummer) verweist, in dem die Adresse eines TKBs eingetragen wurde (in Abb. 3.7 das Element 2).

Die unter Punkt A 3 angegebenen Hinweise dienen zur Verkettung von TKBs, die in die Zeitwarteschlange eingeordnet worden sind. Auch hier ist eine Vorwärts/Rückwärtsverzeigerung erforderlich, um z. B. bei Aufhebung einer zyklischen Beauftragung (PREVENT-Anweisung) die Einplanungsverzeigerung auf den neuen Stand bringen zu können.

Die Zeitwarteschlange besteht aus mehreren Zeitbereichen mit jeweils 64 Elementen, von denen in Abb. 3.8 zwei Bereiche dargestellt wurden. Jedem Element von Zeitbereich 1 entspricht eine Zeitspanne $\Delta\tau_1$, deren Länge z. B. durch Weckaufträge an eine Realzeituhr eingestellt werden kann. Die den Elementen der höheren Zeitbereiche zugeordneten Zeitspannen $\Delta\tau_i$ ergeben sich aus $\Delta\tau_1$ gemäß: $\Delta\tau_i = 32.\Delta\tau_{i-1}$. Bei einer Auflösungsdauer von $\Delta\tau_1$ = 50 msec lassen sich mit 3 Zeitbereichen zyklische Beauftragungen bis zu etwa 54 min bearbeiten.

Jedem Zeitbereich ist ein Zähler ZAIZi (Zeiger auf aktuelles Intervall im Zeitbereich i) zugeordnet, der die Zahlen von 0 bis 63 zyklisch durchläuft und nach Ablauf von jeweils einer Intervallzeit $\Delta\tau_i$ weitergezählt wird.

Eine Zeitspanne Δt wird bei der Einplanung eines TKBs in die Zeitwarteschlange in Teilzeiten $\Delta t_i = n_i.\Delta\tau_i$; $0 < n_i \leq 63$ aufgespalten, aus denen hervorgeht, wie lange der TKB über den Zeitbereich i verwaltet werden soll. Die Einordnung des TKBs erfolgt zunächst in den Zeitbereich i mit größtem Δt_i im Abstand n_i modulo 63 zum Zählerstand ZAIZi.

Durch die Zähler ZAIZi werden Bearbeitungsintervalle in den Zeitbereichen definiert. TKBs, die in das Bearbeitungsintervall eines der Zeitbereiche i;

$i > 1$ eingeordnet sind, werden in den Zeitbereich i-1 umgeordnet.[1] Die im Bearbeitungselement von Zeitbereich 1 angegebenen TKBs, werden gemäß der Prioritätsangabe (Angabe A4) in die Taskwarteschlange oder die Ablaufschlange für Vordergrundtasks eingetragen (Zustandswechsel: zurückgestellt bzw. bekannt → ablaufbereit).

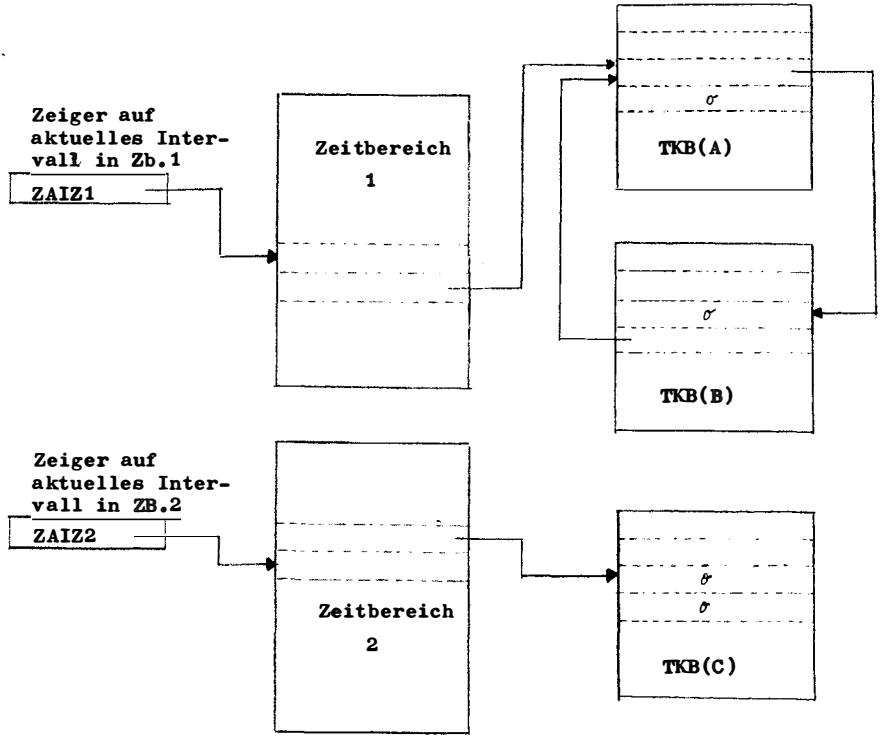


Abb. 3.8 Zum Aufbau der Zeitwarteschlange

Damit nicht durch den Zeitbedarf für die Umordnung von TKBs ablaufbereite hochpriore Benutzertasks verzögert werden, sind für die Umordnung (System-) Tasks vorgesehen, die sich gemeinsam mit den ablaufbereiten Vordergrundtasks um die Zentraleinheit bewerben.

Um diese Verteilung der Organisationszeit zu ermöglichen, werden jeweils die Hälfte aller Zeitbereiche außer dem größten als Pufferbereiche verwendet. Die Umordnung eines TKBs, der zum Zeitpunkt t vom Zeitbereich i nach Zeitbereich i-1 umzuordnen ist, kann daher schon im Intervall $t - \Delta\tau_i$ bis t

1) Bei mehr als drei Zeitbereichen kann der TKB nach Durchlaufen von maximal drei Zeitbereichen aus der Zeitwarteschlange ausgeordnet werden, da die Zeitangabe dann schon mit etwa 1 ‰ Genauigkeit eingehalten wird.

ausgeführt werden.

Auch für die Verwaltung der Vordergrundtasks wird eine Zeitwarteschlange der soeben beschriebenen Art verwendet (Ablaufschlange). Die wichtigste Task wird wie folgt bestimmt: ausgehend von dem Zeiger auf das aktuelle Intervall im Zeitbereich 1 der Ablaufschlange wird der Zeitbereich 1 durchsucht. Wird dabei ein Element der Wartschlange gefunden, in dem die Adresse eines TKBs eingetragen wurde, so wird die Task, die über diesen TKB verwaltet wird, in den Zustand "ablaufend" gebracht. Ist kein TKB im Zeitbereich 1 der Ablaufschlange eingeordnet, so wird die Suche in gleicher Weise im Zeitbereich 2 fortgesetzt, u. s. w. Ähnlich wie bei der nummerngesteuerten Ablauforganisation kann das Auffinden der wichtigsten Task durch einen Zeiger auf das Element der Ablaufschlange, in dem TKBs mit der kürzesten Restzeit eingeordnet wurden, beschleunigt werden.

In Abb. 3.9 ist ein Momentausschnitt von Zeitbereich 1 der Ablaufschlange dargestellt. Da der Zeiger auf das aktuelle Bearbeitungsintervall auf Element 60 hinweist, sind die Tasks A und B, auf deren TKBs im Element 62

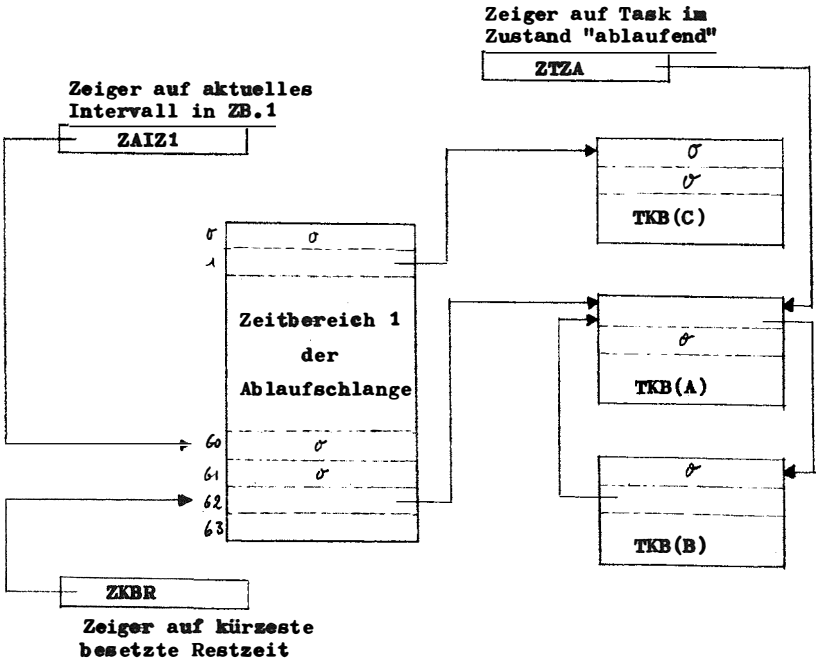


Abb. 3.9 Zur Ablaufsteuerung über Restzeiten.

verwiesen wird, gleichwichtig und wichtiger als die Task C mit Task-Kontroll Block TKB(C), dessen Adresse in Element 1 notiert ist. Nimmt man eine

Auflösungszeit von 50 msec an, so müssten die Tasks A und B nach 100 msec zuzüglich der Zeitspanne, die bis zum Weiterschalten des Zeigers ZAIZ1 noch verbleibt, ausgeführt worden sein.

Während die Überschreitung vorgegebener Ausführungszeiten bei ausschließlicher Verwendung einer nummerngesteuerten Ablauforganisation nur durch zusätzliche Überwachungsprogramme, innerhalb derer die Abbildung der Ausführungszeiten auf Prioritätsnummern wieder rückgängig gemacht wird, oder durch Fehlverhalten eines technischen Prozesses feststellbar ist, ergibt sie sich bei Steuerung über Restzeiten von selbst. Dieser Umstand dürfte erheblich zur Vereinfachung des Gesamttests (vergl. Abschnitt 3.3) beitragen. Wird nämlich beim Weiterschalten des Zeigers ZAIZ1 festgestellt, daß in dem bis dahin aktuellen Intervall des Zeitbereichs 1 der Ablaufschlange noch auf TKBs verwiesen wird, so konnten vorgegebene Ausführungszeiten nicht eingehalten werden (Systemüberlastung). Weiter ist es möglich, das Verfahren auf negative Restzeiten auszudehnen, um kurzzeitige Systemüberlastungen aufzufangen.

Die Verwendung der Angaben A5 bis A7 im TKB ist naheliegend. Die Dauerangabe A5 wird zur erneuten Einplanung eines Dauerereignisses ausgewertet, wenn ein TKB die Zeitwarteschlange verlässt. Aus A6 wird bei der Ausführung einer ACTIVATE-Anweisung die Startadresse für das Segment einer Task entnommen. Diese wird dabei in das Element des TKBs übertragen (Angaben A7), in dem auch die Fortsetzungsadresse bei einer Unterbrechung der Task notiert wird. Schließlich wird der unter A7 angegebene Bereich eines TKBs zur Aufnahme von Arbeitsregistern bei Taskunterbrechungen eingesetzt.[1)]

---

1) Das vorausgehend dargestellte Implementierungsverfahren ist eine Erweiterung der PAS1-Ablauforganisation [49,50].

4. Anwendung des Verfahrens auf diskrete Regelsysteme.

4.1 Regelsysteme mit endlicher Einschwingdauer (dead beat response).

Ausführungszeiten für Tasks zur Automatisierung technischer Prozesse können entweder durch Messung des zeitlichen Verhaltens der technischen Prozesse bestimmt oder aus Prozeßmodellen abgeleitet werden. Ein Anwendungsfall, in dem sich die Ausführungszeiten von Tasks modellmäßig bestimmen lassen, sind Abtast-Regelsysteme, die nach einem Steuerungsprinzip entworfen werden. Bei Kenntnis des Verlaufs der Führungsgröße und/oder der Störungen des Regelsystems kann der Abtastalgorithmus so gewählt werden, daß die Ausgangsgröße bei Änderungen der Führungsgröße nach einer vorgegebenen Zeitspanne den neuen Endwert erreicht [51,52,53,54].

In Abb. 4.1 ist das Blockschaltbild eines einfachen Abtast-Regelsystems dargestellt, das aus einem diskreten Regler (Automationsprogramm), einem Halteglied (z. B. Analogausgabe) und einer Regelstrecke 2. Ordnung besteht. Zu den Abtastzeitpunkten t_i; i = 0, 1, ..., die normalerweise als Vielfache einer geeigneten Abtastzeit T gewählt werden, wird die Regelabweichung v(t), die sich aus der Differenz der Führungsgröße w(t) und der Ausgangsgröße x(t) ergibt, von dem diskreten Regler abgetastet (Analogeingabe). Der diskrete Regler liefert daraufhin die Stellgrößen $y(t_i)$, die durch das Halteglied in die treppenförmige Stellgröße $\overline{y}(t)$ für die Regelstrecke 2. Ordnung umgesetzt werden.

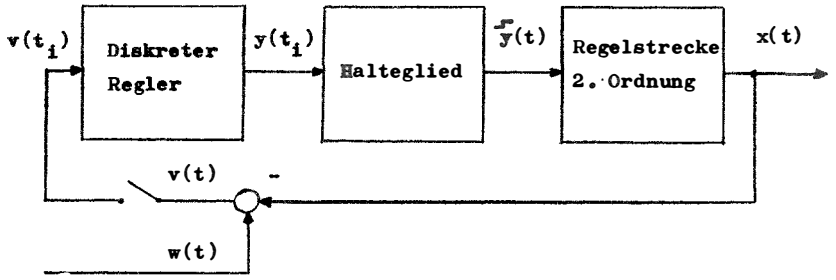


Abb. 4.1 Blockschaltbild eines Abtast-Regelsystems mit Strecke 2. Ordnung.

Geht man bei der Synthese des diskreten Reglers von sprungförmigen Änderungen der Führungsgröße w(t) = w.1(t) bei stationärem Ausgangszustand des Regelsystems:

$$x(0) = x_o; \quad \dot{x}(0) = 0$$

aus, so kann die Ausgangsgröße x(t) wegen der Linearität der Strecke als Summe von Sprunganworten s der Strecke zu den Zeitpunkten i.T; i = 0, 1, ... dargestellt werden:

$$(4.1) \qquad x(t) = x_o + \sum_{i=0}^{\infty} y(i.T).s(t-i.T) \; .$$

Durch geeignete Wahl der diskreten Stellwerte y(i.T) kann erreicht werden, daß sich ein Regelsystem der in Abb. 4.1 betrachteten Art mit Strecke n-ter Ordnung nach genau n Abtastzeiten wieder in einem stationären Zustand befindet (dead beat response). Für ein System 2. Ordnung muß dazu gelten:

$$x(2.T) = w; \quad \dot{x}(2.T) = 0.$$

Mit Gl. (4.1) und mit $s(0) = \dot{s}(0) = 0$ erhält man daraus:

$$y(0) = (w - x_o) \frac{\dot{s}(T)}{s(2.T).\dot{s}(T) - s(T).\dot{s}(2.T)} = (w - x_o).y_o$$

$$y(T) = -(w - x_o) \frac{s(2.T)}{s(2.T).\dot{s}(T) - s(T).\dot{s}(2.T)} = -(w - x_o).y_1$$

Der Wert von y(2.T) ergibt sich aus der Forderung, daß der neue Führungswert für $t \geq 2.T$ an der Strecke anliegen soll:

$$y(2.T) = w - x_o - y(0) - y(T) = (w - x_o).y_2 \; .$$

Aus der z-Transformierten des diskreten Reglers:

$$F_R(z) = \frac{y(z)}{v(z)} = \frac{y(2.T) + y(T).z + y(0).z^2}{v(2.T) + v(T).z + v(0).z^2}$$

erhält man durch Rücktransformation mit:

$$v(0) = w - x_o$$

$$v(T) = w - x(T) = w - x_o - y(0).s(T) = (w - x_o).v_1$$

$$v(2.T) = 0$$

die Rekursionsformel (i.T → i):

$$(4.2)\quad \begin{cases} y(i) = y_2.v(i-2) + y_1.v(i-1) + y_0.v(i) - v_1.y(i-1) \\ v(-2) = v(-1) = y(-1) = 0. \end{cases}$$

In Abb. 4.2 ist der ideale Verlauf der diskreten Regelung für ein Übertragungsglied 2. Ordnung mit Sprungantwort:

$$(4.3)\quad s(t) = \begin{cases} 1 - \dfrac{1}{T_1 - T_2}\left(T_1.e^{-t/T_1} - T_2.e^{-t/T_2} \right) & \text{für } t \geq 0 \\ 0 \text{ für } t \leq 0 \end{cases}$$

dargestellt.

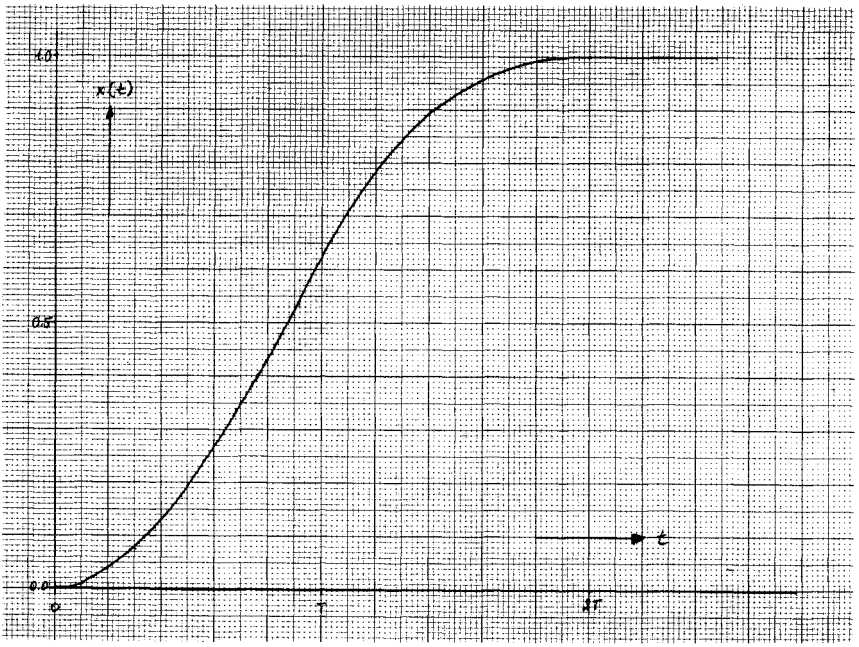


Abb. 4.2 Idealer Verlauf einer diskreten Regelung (dead beat response) für eine PT_2-Strecke ($T_1 = 2.T$; $T_2 = T$).

4.2 Vergleich der nummerngesteuerten mit der restzeitgesteuerten Ablauforganisation anhand von Regelungsverläufen.

Der in Abschnitt 4.1 konstruierte diskrete Regler gewährleistet das schnelle Einschwingen des Regelsystems nur dann, wenn die Abtastung der Regelabweichung und die Ausgabe der Stellwerte zu den bei der Synthese des Reglers

vorausgesetzten Abtastzeitpunkten i.T; i = 0, 1, ... erfolgt. Werden die Abtastzeitpunkte im Mehrprogrammbetrieb nicht eingehalten und/oder kann die Laufzeit des Automationsprogramms, durch das der diskrete Regler dargestellt wird, gegenüber der Abtastzeit nicht vernachlässigt werden, so ergeben sich Abweichungen von dem in Abb. 4.2 dargestellten idealen Regelverlauf, die bei restzeitgesteuerter Ablauforganisation durch Vorgabe von Ausführungszeiten für den Regelalgorithmus (4.2) einschränkbar sind.

Die Vorteile der zeitgesteuerten Ablauforganisation sollen im folgenden anhand des Regelverlaufs von PT_2-Strecken dargestellt werden. In den Beispielen werden, der Übersichtlichkeit halber, nur jeweils zwei Regelsysteme der in Abschnitt 4.1 beschriebenen Art betrachtet und der Regelverlauf bei Einsatz nummerngesteuerter Ablauforganisationen mit dem bei restzeitgesteuerter Ablauforganisation verglichen. Die diskreten Regler seien durch Automationsprogramme realisiert, die auf einer Rechenanlage mit einer Zentraleinheit ausgeführt werden.

Beispiel 1: Vergleich mit einer nummerngesteuerten Ablauforganisation, die den Ablauf einer Task unterbricht, sobald eine Task höherer Priorität ablaufbereit wird.

Betrachtet werden zwei Übertragungsglieder 2. Ordnung mit folgenden Zeitkonstanten:

Strecke 1: $T_1^{(1)} = 2 \cdot T^{(1)}$; $T_2^{(1)} = T^{(1)}$;

Strecke 2: $T_1^{(2)} = 2 \cdot T^{(2)}$; $T_2^{(2)} = T^{(2)}$.

Darin ist $T^{(1)}$ bzw. $T^{(2)}$ die Abtastzeit für Strecke 1 bzw. Strecke 2. Weiter sei:

$$T^{(2)} = \frac{2}{3} T^{(1)} \quad .$$

Unter diesen Voraussetzungen ergeben sich hinsichtlich der Ausführung der Regelalgorithmen (4.2) für die beiden Strecken die in Abschnitt 3.1, Beispiel 2 diskutierten Verhältnisse. Da nur im Fall b) des in Abschnitt 3.1 diskutierten Beispiels ein Unterschied zwischen der nummerngesteuerten und der restzeitgesteuerten Ablauforganisation vorliegt, muß die Laufzeit Δt_{Lmax} des Regelalgorithmus für das Beispiel dem Intervall:

$$0.5 \cdot T^{(2)} < \Delta t_{Lmax} \leq 0.6 \cdot T^{(2)}$$

entnommen werden. Den späteren Abbildungen liegt der Wert: $\Delta t_{Lmax} = 0.55 \cdot T^{(2)}$ zugrunde. Weiter wurden die Tasks zur Regelung der Strecke 1 bzw. Strecke 2 wie früher mit Q_1 bzw. Q_2 bezeichnet.

Die Priorität von Q_1 kann größer, gleich oder kleiner als die von Q_2 gewählt werden (d. h. die Prioritätsnummer von Q_1 ist kleiner, gleich oder größer als die von Q_2). Während sich bei unterschiedlichen Prioritätsnummern der zu erwartende Regelverlauf eindeutig vorhersagen lässt, hängt er bei gleichen Prioritätsnummern vom Aufbau der die Taskwarteschlange bearbeitenden Betriebssystemfunktion (Scheduler) ab. Der Fall gleicher Prioritätsnummern wird deshalb im folgenden nicht betrachtet.

In Abb. 4.3 bzw. Abb. 4.4 ist die Schachtelung von Q_1 und Q_2 unter der Voraussetzung dargestellt, daß die Priorität von Q_1 größer bzw. kleiner als die von Q_2 ist. Dabei wurde in Anlehnung an den PEARL-Report ([18], S. 102, letzter Satz) angenommen, daß die Aktivierung einer Task, die sich nicht im Zustand "bekannt" befindet, nach Beendigung der Task durchgeführt wird.

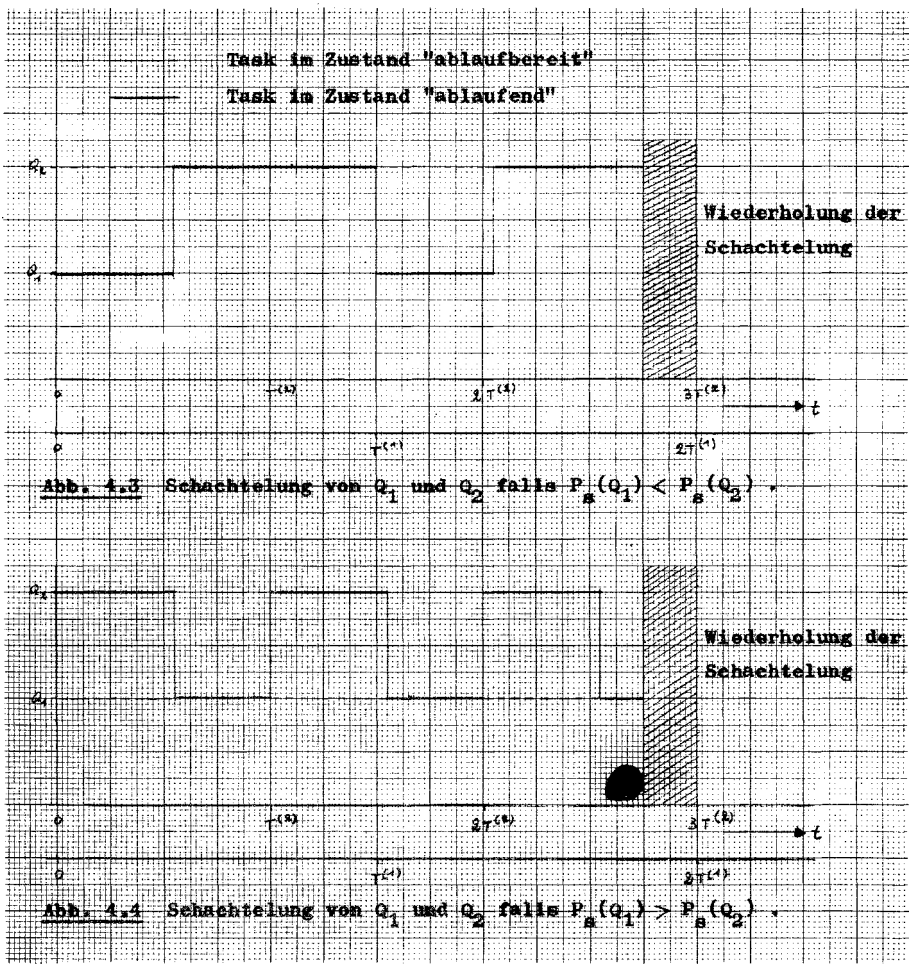


Abb. 4.3 Schachtelung von Q_1 und Q_2 falls $P_s(Q_1) < P_s(Q_2)$.

Abb. 4.4 Schachtelung von Q_1 und Q_2 falls $P_s(Q_1) > P_s(Q_2)$.

Bei der Berechnung der Regelverläufe gemäß der in Abschnitt 4.1 dargestellten Theorie wurde vorausgesetzt, daß die Regelabweichung zu Beginn von Q_1 bzw. Q_2 eingelesen wird und daß die Stellwertausgabe bei Beendigung von Q_1 bzw. Q_2 erfolgt. Da, wie aus Abb. 4.3 und Abb. 4.4 hervorgeht, die Zeitpunkte für das Einlesen der Regelabweichung und für die Ausgabe der diskreten Stellwerte bei der jeweils niederprioren Task schlecht korreliert sind, lässt sich ein schlechter Regelverlauf für diese Task erwarten.

Dies wurde durch die Rechnungen bestätigt. Abb. 4.5 und Abb. 4.6 zeigen den Verlauf der Regelungen für die in Abb. 4.3 und Abb. 4.4 dargestellte Schachtelung. Die Einschwingzeit für die Strecke, welche jeweils durch die Task niedrigerer Priorität geregelt wird, vergrößert sich in beiden Fällen um etwa 1/3 gegenüber der kleinsten möglichen Einschwingzeit (vergl. z. B. den Regelverlauf für Strecke 2 nach Abb. 4.5 und Abb. 4.6).

Bei Verwendung einer restzeitgesteuerten Ablauforganisation ergibt sich, wenn die Ausführungszeit gleich der Abtastzeit gewählt wird, die in Abb. 4.7 dargestellte Schachtelung. Die aus Abb. 4.8 hervorgehenden Regelverläufe zeigen nur geringe Abweichungen vom bestmöglichen Verlauf (siehe z. B. Regelverlauf für Strecke 2 nach Abb. 4.8 und Abb. 4.6).

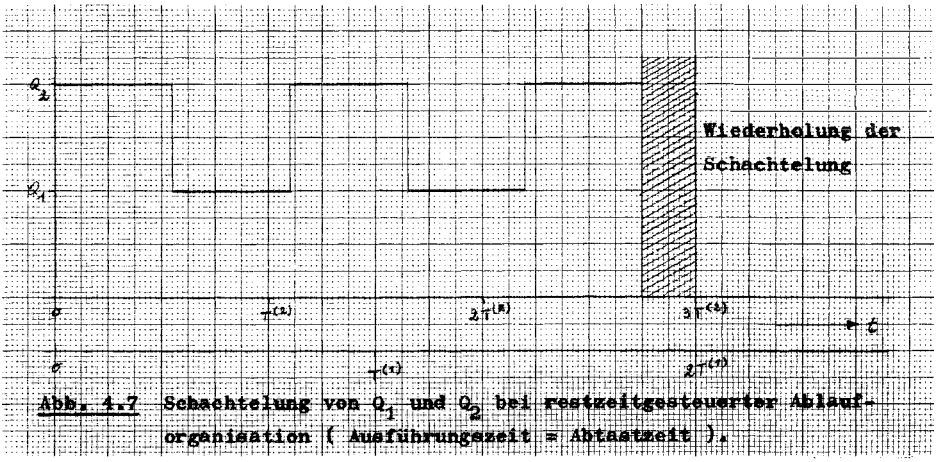


Abb. 4.7 Schachtelung von Q_1 und Q_2 bei restzeitgesteuerter Ablauforganisation (Ausführungszeit = Abtastzeit).

Beispiel 2: Vergleich mit einer nummerngesteuerten Ablauforganisation, die Prioritätsnummern nur bei der Suche nach der wichtigsten Task auswertet.

Wie in Abschnitt 3.1 schon erwähnt wurde, lässt sich die Schachtelung gemäß Abb. 4.7 auch erreichen, wenn die im Zustand "ablaufend" befindliche Task trotz Aktivierung einer anderen Task bis zu ihrer Beendigung in diesem Zustand verbleibt. Die Prioritätsnummern werden nur bei der Suche nach der wichtigsten Task ausgewertet.

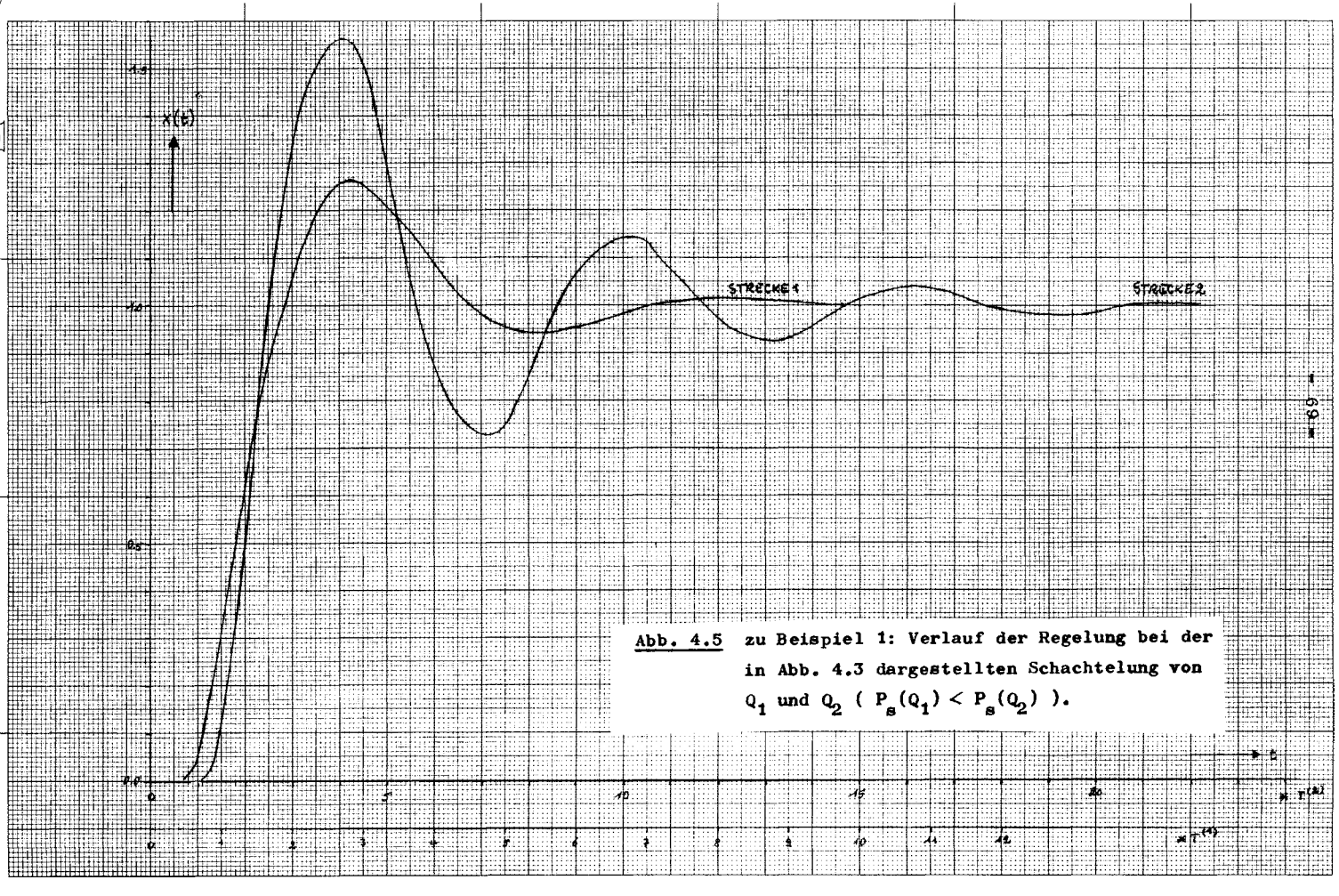


Abb. 4.5 zu Beispiel 1: Verlauf der Regelung bei der in Abb. 4.3 dargestellten Schachtelung von Q_1 und Q_2 ($P_s(Q_1) < P_s(Q_2)$).

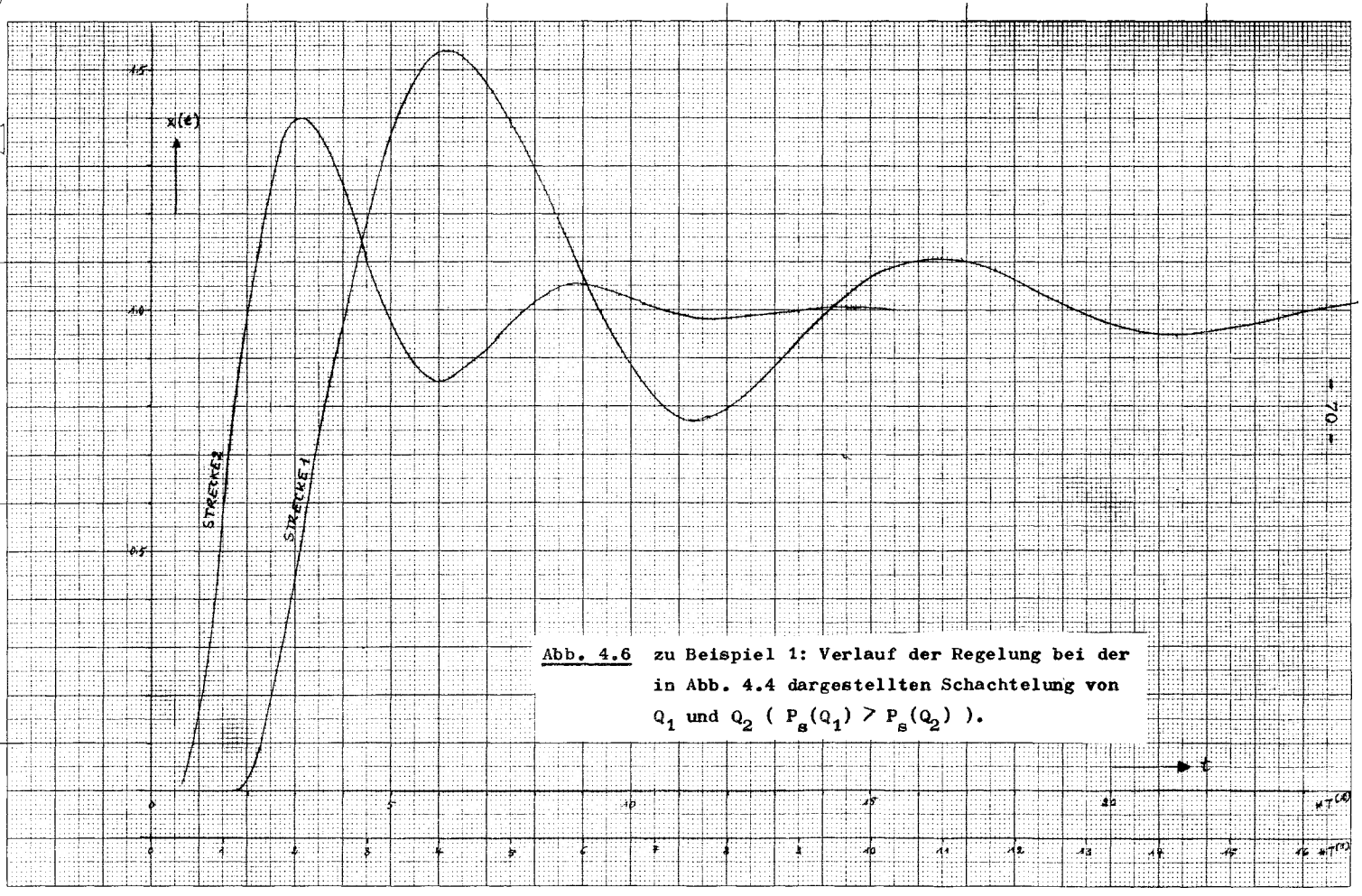


Abb. 4.6 zu Beispiel 1: Verlauf der Regelung bei der in Abb. 4.4 dargestellten Schachtelung von Q_1 und Q_2 ($P_s(Q_1) > P_s(Q_2)$).

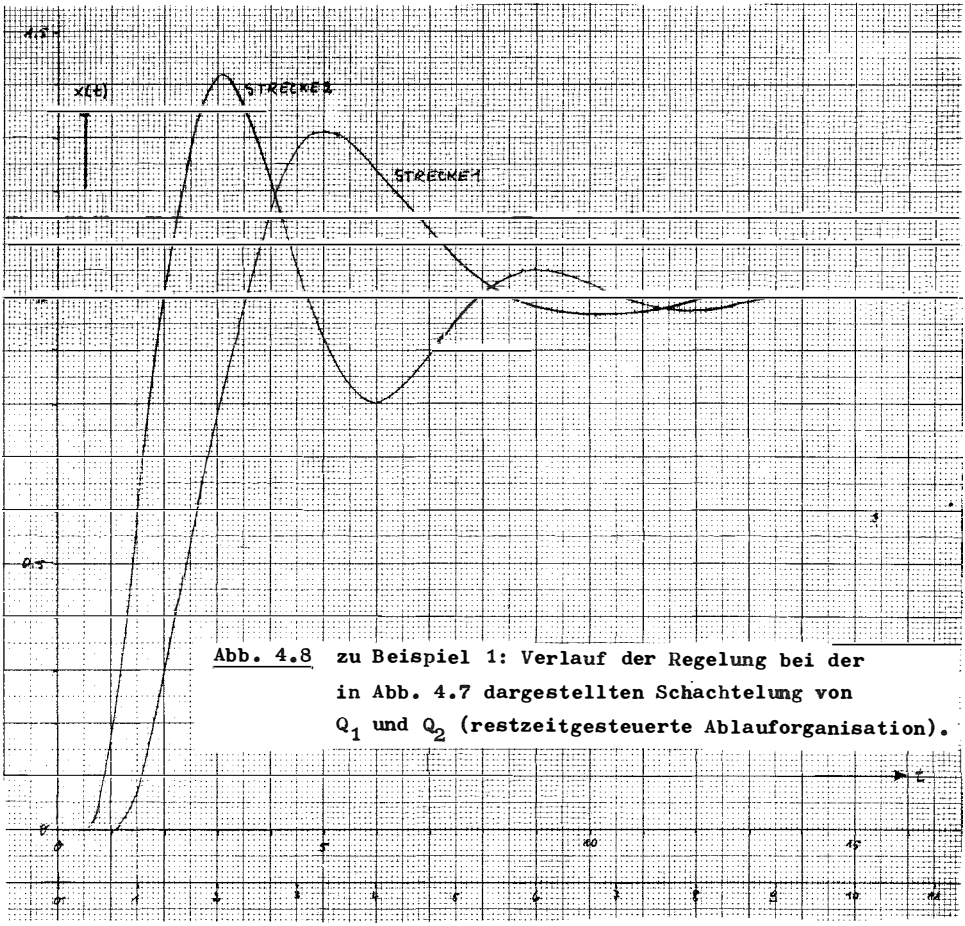


Abb. 4.8 zu Beispiel 1: Verlauf der Regelung bei der in Abb. 4.7 dargestellten Schachtelung von Q_1 und Q_2 (restzeitgesteuerte Ablauforganisation).

Daß diese Strategie, die mit geringfügigen Erweiterungen z. B. im Betriebssystem MARVIS (Prozeßrechner AEG 60-50) realisiert ist, wie die in Beispiel 1 beschriebene Organisation nur in bestimmten Fällen zu gleichguten Ergebnissen wie das in dieser Arbeit vorgeschlagene Verfahren führt, soll mit der folgenden Fallstudie demonstriert werden.

Betrachtet werden wieder zwei Übertragungsglieder 2. Ordnung mit den in Beispiel 1 angegebenen Relationen zwischen den Zeitkonstanten. Weiter seien die Abtastzeiten gemäß:

$$T^{(2)} = \frac{4}{5} T^{(1)}$$

verknüpft und die Ausführungszeit für den Regelalgorithmus betrage:
$\Delta t_{Lmax} = 0.3 \cdot T^{(1)}$.

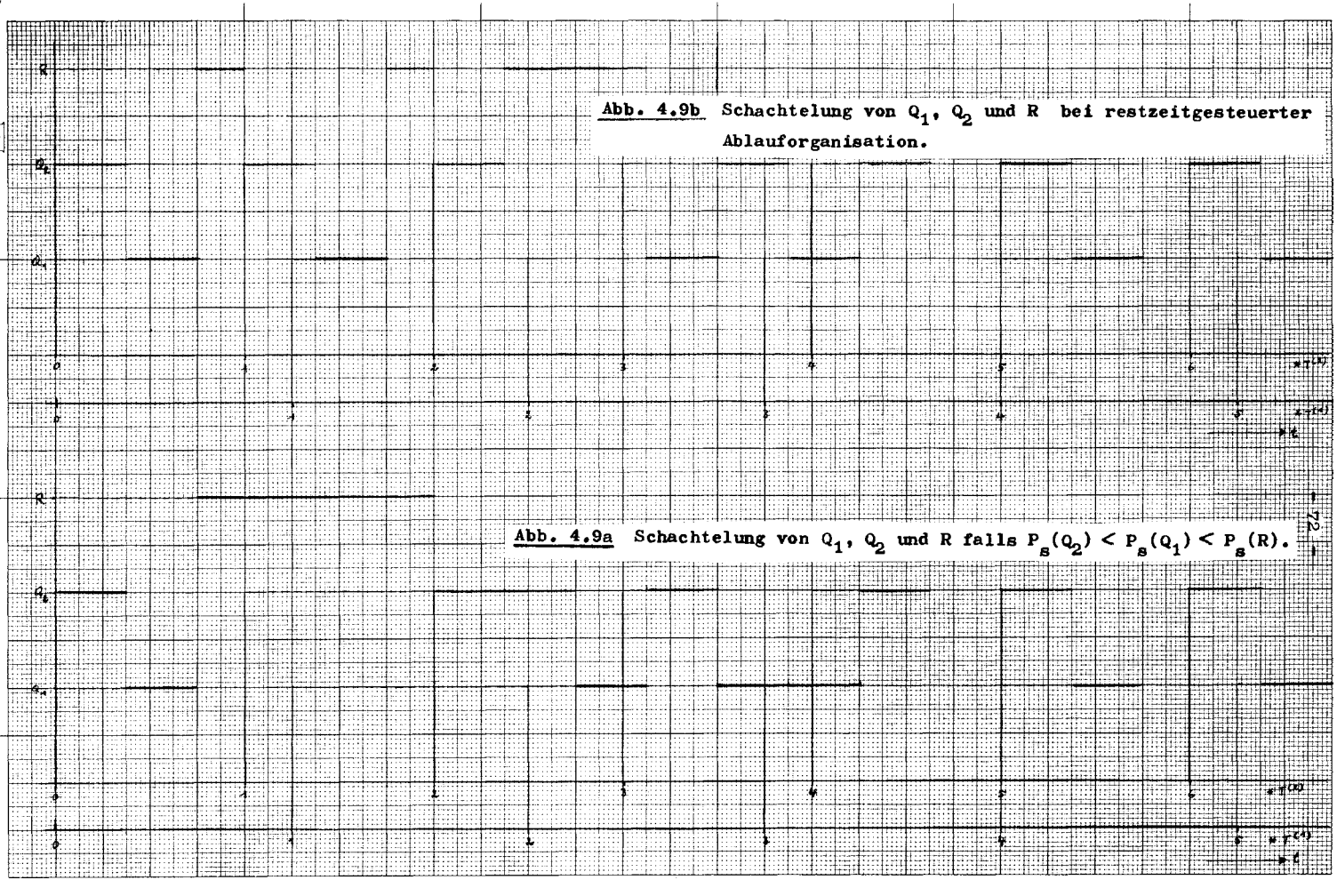


Abb. 4.9b Schachtelung von Q_1, Q_2 und R bei restzeitgesteuerter Ablauforganisation.

Abb. 4.9a Schachtelung von Q_1, Q_2 und R falls $P_s(Q_2) < P_s(Q_1) < P_s(R)$.

Außer den beiden Tasks Q_1 und Q_2 zur Regelung von Strecke 1 und Strecke 2 sei zum Zeitpunkt 0 eine weitere Task R niedriger Priorität ablaufbereit, die spätestens nach der Zeitspanne $3.5 \cdot T^{(2)}$ ausgeführt sein soll und deren Laufzeit mit $T^{(1)}$ übereinstimme.

Legt man die Prioritäten gemäß: $P_s(Q_2) < P_s(Q_1) < P_s(R)$ fest, so ergibt sich die in Abb. 4.9a dargestellte Schachtelung der Tasks Q_1, Q_2 und P. Ersichtlich wird die wichtigere Ausführung der Regelalgorithmen solange unterbunden, bis die unwichtigere Task R ausgeführt ist. Als Folge davon ergibt sich das in Abb. 4.10 dargestellte starke Schwingen der Ausgangsgröße zu Beginn der Regelung.

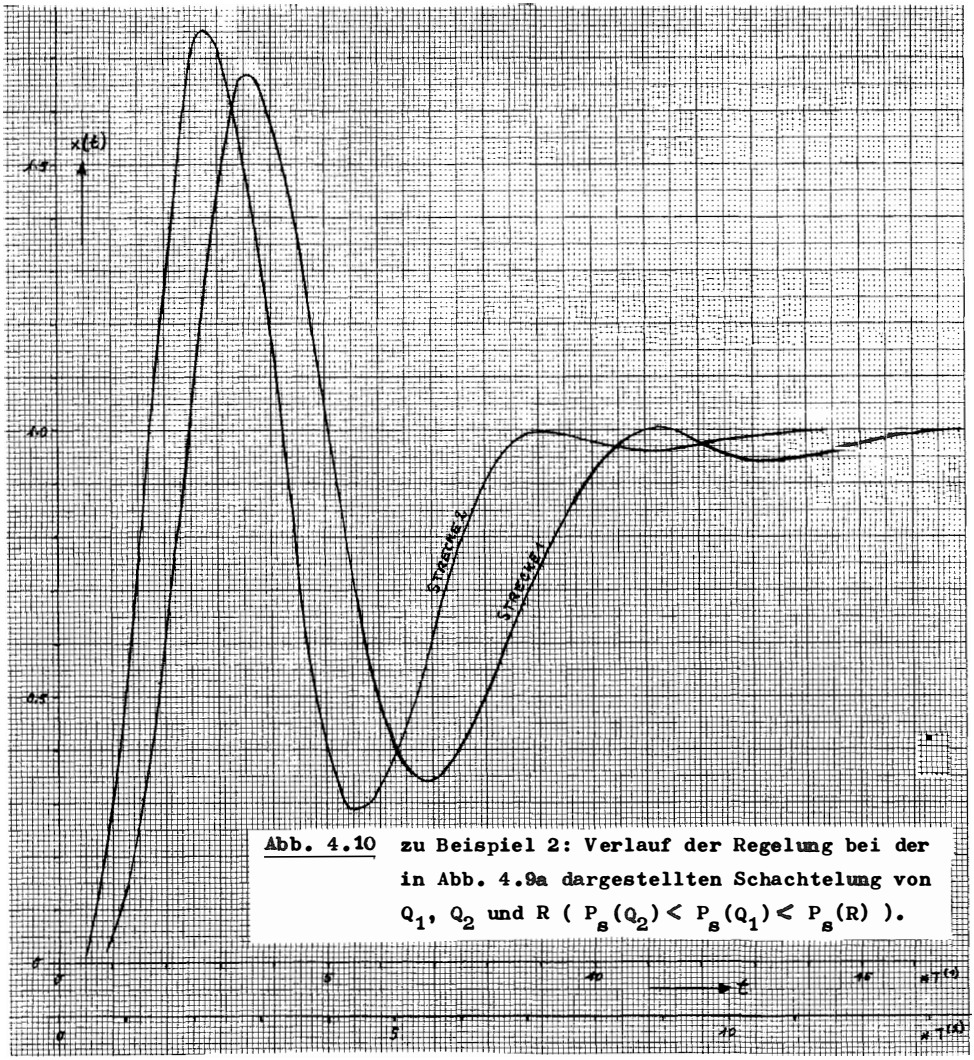


Abb. 4.10 zu Beispiel 2: Verlauf der Regelung bei der in Abb. 4.9a dargestellten Schachtelung von Q_1, Q_2 und R ($P_s(Q_2) < P_s(Q_1) < P_s(R)$).

Dagegen wird durch die in Abb. 4.9b dargestellte Schachtelung der Tasks bei restzeitgesteuerter Ablauforganisation eine gleichmäßigere Aufteilung der Rechenzeit erreicht. Die Task R ist zunächst unwichtig und wird nur dann ausgeführt, wenn sich Q_1 und Q_2 im Zustand "bekannt" befinden. Mit zunehmender Annäherung an ihre Endezeit (dead line) wird R automatisch immer wichtiger und ist zum Zeitpunkt $1.9 \cdot T^{(1)}$ wichtiger als Q_1. Weiter hat zum Zeitpunkt $2.5 \cdot T^{(1)}$ ein Prioritätswechsel zwischen Q_1 und Q_2 stattgefunden. Abb. 4.11 zeigt den Verlauf der Regelung bei restzeitgesteuerter Ablauforganisation.

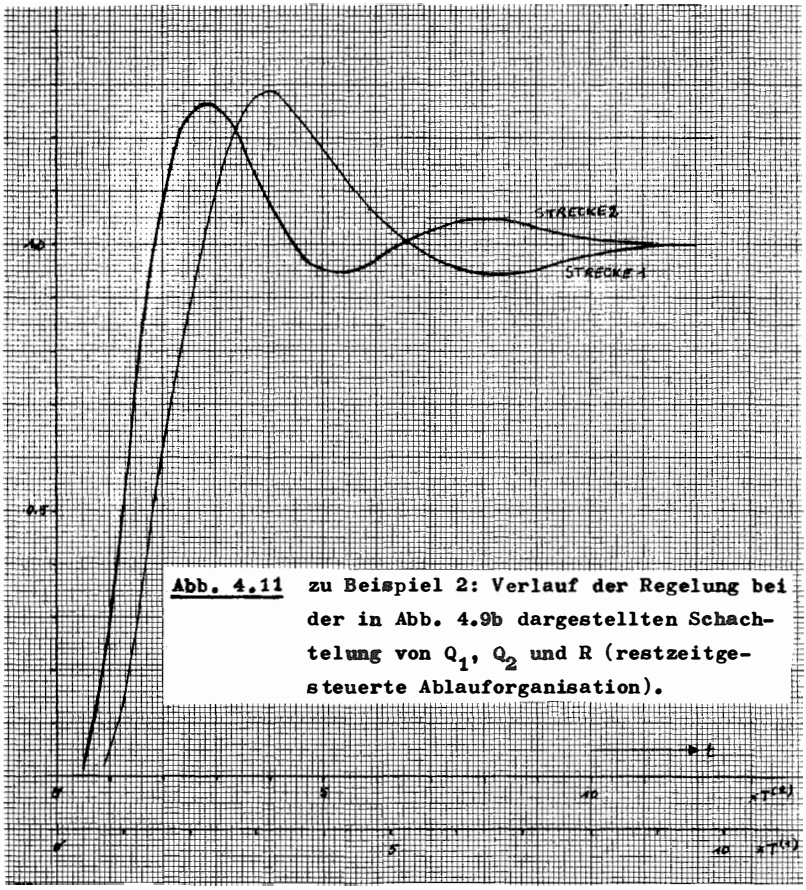


Abb. 4.11 zu Beispiel 2: Verlauf der Regelung bei der in Abb. 4.9b dargestellten Schachtelung von Q_1, Q_2 und R (restzeitgesteuerte Ablauforganisation).

Obwohl die angeführten Beispiele wegen der mit der Anzahl der Tasks schnell zunehmenden Komplexität der Schachtelung relativ einfach gewählt werden mussten, zeigen sie doch deutlich die Überlegenheit der restzeitgesteuerten Ablauforganisation gegenüber gegenwärtig in Prozeßrechner-Betriebssystemen realisierten Ablaufstrategien. Dies ist nach den Diskussionen in Abschnitt 3.2 und Abschnitt 3.3 nicht verwunderlich. Das Prinzip der kürzeren Restzeit ermöglicht nicht nur die unabhängige Erstellung von Automationsprogrammen für unabhängige technische Prozesse, sondern sorgt auch für die möglichst optimale Nutzung der Betriebsmittel eines Rechensystems in Abhängigkeit von der Geschwindigkeit der Vorgänge in technischen Prozessen.

4.3 Zur Bestimmung der Ausführungszeit für Abtast-Regelalgorithmen.

Das im folgenden beschriebene Verfahren liefert die obere Schranke für die Verschiebung von Abtastzeitpunkten, wenn mit dem in Abb. 4.1 dargestellten Regelsystem eine vorgegebene Regelgüte eingehalten werden soll. Die einer vorgegebenen Regelgüte zugeordnete obere Schranke für die Verschiebungen der Abtastzeitpunkte kann als die Ausführungszeit für den in Abschnitt 4.1 angegebenen Regelalgorithmus angesehen werden.

Die Abhängigkeit der Ausführungszeit von einer beliebig definierten Regelgüte kann unter Verwendung zufallsverteilter Verschiebungen der Abtastzeitpunkte nach der Monte-Carlo-Methode (siehe z. B. [55], Kapitel 14) berechnet werden. Statt der bei der Synthese des diskreten Reglers vorausgesetzten Abtastzeitpunkte i.T wird die Rekursionsformel (4.2) für zufallsverteilte Abtastzeitpunkte $t_0, t_1, \ldots;\ i.T \leq t_i \leq (i+1).T$ verwendet, wobei sich der Wert von v(i) aus:

$$v(i) = w - x_0 - \sum_{l=0}^{i-1} y(l).s(t_i - t_l)$$

ergibt.

Bei der Bestimmung von Ausführungszeiten geht man von einer zufallsverteilten Verschiebung der Abtastzeitpunkte mit oberer Schranke T_s (Anfangswert: $T_s = T$) aus und untersucht, ob der Regelverlauf mit der geforderten Regelgüte verträglich ist. Wird diese nicht erreicht, so wird die obere Schranke T_s für die zufälligen Verschiebungen der Abtastzeitpunkte erniedrigt und der Rechenvorgang wiederholt. Die Erniedrigung von T_s wird solange fortgesetzt, bis die Regelgüte bei einer hinreichend großen Anzahl von Rechenvorgängen mit jeweils unterschiedlichen zufallsverteilten Abtastzeitpunkten eingehalten wird. Der dabei gültige Wert von T_s ist die Ausführungszeit für den Abtast-Regelalgorithmus.

Während sich das beschriebene Verfahren empfiehlt, wenn nach einem komplizierten Gütekriterium (z. B. nach minimalem Energieverbrauch in der Strecke während des Regelvorgangs) oder nach mehreren Gütekriterien gleichzeitig optimiert werden soll, ist es bei einfachen Gütekriterien gelegentlich möglich, die Ausführungszeit für den Regelalgorithmus direkt zu bestimmen. Soll beispielsweise nur das Überschwingen der Ausgangsgröße in Grenzen gehalten werden, so kann man von folgenden Abtastzeitpunkten ausgehen:

$$t_i = \begin{cases} i.T & \text{falls i gerade} \\ i.T + T_s & \text{falls i ungerade} \end{cases}$$

und für Werte von T_s aus dem Intervall: $0 < T_s \leq T$ die zugeordneten Maximalwerte der Ausgangsgröße: x_{max} berechnen. Bei vorgegebenem x_{max} kann aus einer Tabelle oder einer graphischen Darstellung die Ausführungszeit abgelesen werden.

Abb. 4.12 zeigt den Zusammenhang zwischen dem Maximalwert der Ausgangsgröße

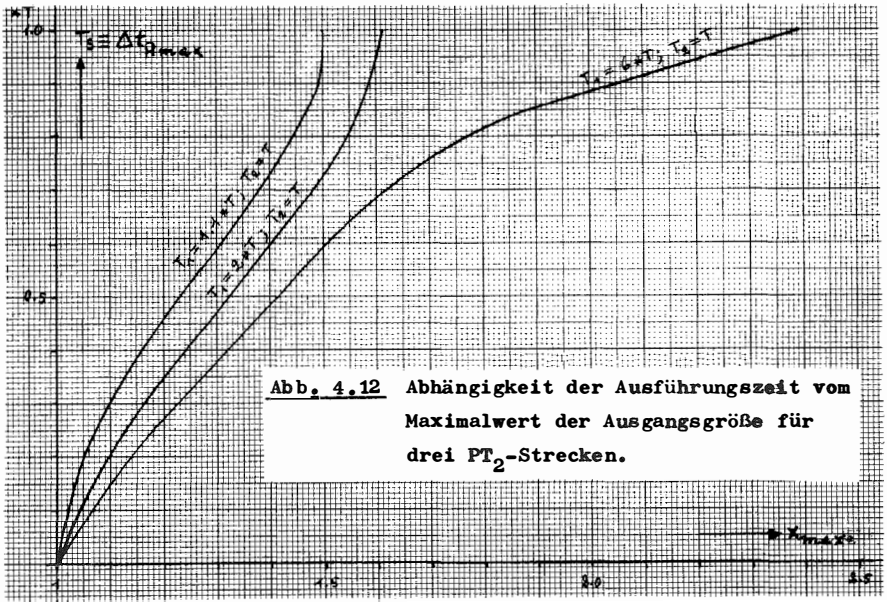


Abb. 4.12 Abhängigkeit der Ausführungszeit vom Maximalwert der Ausgangsgröße für drei PT_2-Strecken.

und der Ausführungszeit für drei PT_2-Strecken, wenn die Rechenzeit für den Regelalgorithmus (4.2) gegenüber der Abtastzeit vernachlässigt werden darf. Aus der Abb. geht hervor, daß sich ein mit dem dead beat response-Verhalten vergleichbarer Regelverlauf für die betrachteten Strecken nur durch drastisches Einschränken der Ausführungszeit für den Regelalgorithmus (4.2) erreichen lässt.

Zusammenfassung

In der vorliegenden Arbeit wurde untersucht, welche Ausdrucksmittel für die Parallelprogrammierung in prozeßorientierten Programmiersprachen benötigt werden. Es wurde festgestellt, daß die bisher vorgeschlagenen prozeßorientierten Programmiersprachen u. a. folgende wesentliche Mängel aufweisen:

1. Um Automatiensprogramme bzw. Programmoduln portabel, d. h. unabhängig von der Konfiguration eines Prozeßrechensystems, erstellen zu können, ist es in einigen Programmiersprachen [14,16,18] möglich, das jeweils eingesetzte Prozeßrechensystem in einem eigenen Programmteil (z. B. Systemteil in PEARL) zu beschreiben. Andere Programmteile (Problemteile) dienen zur konfigurationsunabhängigen Darstellung von Automatisierungsfunktionen.

   Die optimale Nutzung aller parallel arbeitsfähigen Geräte eines Prozeßrechensystems kann bisher jedoch nur erreicht werden, wenn auch bei der Formulierung eines Problemteils die Konfiguration des jeweiligen Zielsystems bekannt ist.

2. Beim Aufbau von Prozeßrechner-Programmsystemen aus unabhängig voneinander erstellten Programmoduln lassen sich i. a. die in Automatisierungsfunktionen angegebenen Ausführungsbestimmungen nicht einhalten.

Zur Behebung des ersten Mangels wurde in dieser Arbeit vorgeschlagen, außer "parallel auszuführenden" Anweisungsfolgen (Tasks) auch die Definition "parallel ausführbarer" Anweisungsfolgen zu ermöglichen. Es wurde gezeigt, daß damit -bei geeignetem Aufbau des Übersetzers für eine prozeßorientierte Programmiersprache- Automatisierungsfunktionen konfigurationsunabhängig darstellbar sind, ohne daß auf die optimale Nutzung der parallel arbeitsfähigen Geräte eines Zielsystems verzichtet werden muß.

Als Ursache des zweiten Mangels ergab sich, daß statische Prioritätsnummern einerseits und Modultechnik andererseits bei der Erstellung von Prozeßrechner-Programmsystemen nicht miteinander verträglich sind. Anders als in kommerziellen Rechensystemen sind die Abläufe in einem Prozeßrechensystem i. a. stark an die Vorgänge außerhalb des Prozeßrechensystems gekoppelt. Die Wichtigkeit von Tasks kann deshalb nicht statisch über Prioritätsnummern festgelegt werden, sondern ergibt sich dynamisch aus den jeweils einzuhaltenden Reaktionszeiten bzw. aus Ausführungszeiten für Automationsalgorithmen.

Davon ausgehend, wurde ein Verfahren vorgeschlagen, das es ermöglicht, die Wichtigkeit von Tasks aus vorgebbaren Ausführungszeiten dynamisch abzuleiten. Es wurde gezeigt, daß die mit Prioritätsnummern verbundenen zahlreichen

Unzuträglichkeiten (z. B. Programmänderungen beim Einfahren von Prozeßautomatisierungssystemen) bei Verwendung von Ausführungszeiten zur Synchronisation des Programm- und Prozeßgeschehens weitgehend entfallen. Anhand von Abtastregelsystemen wurde die Überlegenheit des neuen Verfahrens gegenüber bisher in Prozeßrechensystemen verwendeten Ablaufstrategien demonstriert.

Zur Zeit fehlen noch Erfahrungen hinsichtlich der Implementierung des neuen Verfahrens und hinsichtlich der Organisationszeiten im Echtzeitbetrieb. Erst kürzlich auf Großrechenanlagen in ALGOL60 ausgeführte Vergleichsuntersuchungen denkbarer zeitgesteuerter Ablaufstrategien [56] bestätigen die in Abschnitt 3.3 geäußerte Vermutung, daß zeitgesteuerte Ablaufstrategien zu Verwaltungszeiten führen, die mit dem Zeitbedarf für die bisher eingesetzten nummerngesteuerten Verfahren vergleichbar sind. Es ist geplant, die in Abschnitt 3.5 skizzierte Implementation einer restzeitgesteuerten Ablauforganisation für einen Prozeßrechner zu realisieren und anhand von Modellprozessen im Echtzeitbetrieb zu erproben.