Grundlagen der Realzeitprogrammierung

1. Einführung

Realzeitprogrammierung heißt, Universalrechner so zu programmieren, daß alle Abläufe der jeweiligen Anwendung in den Grenzen vorgegebener Zeitraster ausgeführt werden. Realzeit (Echtzeit) bedeutet nicht "so schnell wie möglich", sondern "unter allen Umständen ins jeweils vorgegebene Zeitraster passend".

EDV-Programmierung und Realzeitprogrammierung
In der klassischen Datenverarbeitung werden einzelne Programme abgearbeitet, um aus gegebenen Daten (Eingangsdaten) die jeweils gewünschten Ergebnisse (Ausgangsdaten) zu bestimmen; das klassische Programm entspricht dem Schema Eingabe – Verarbeitung – Ausgabe (Abb. 1.1).

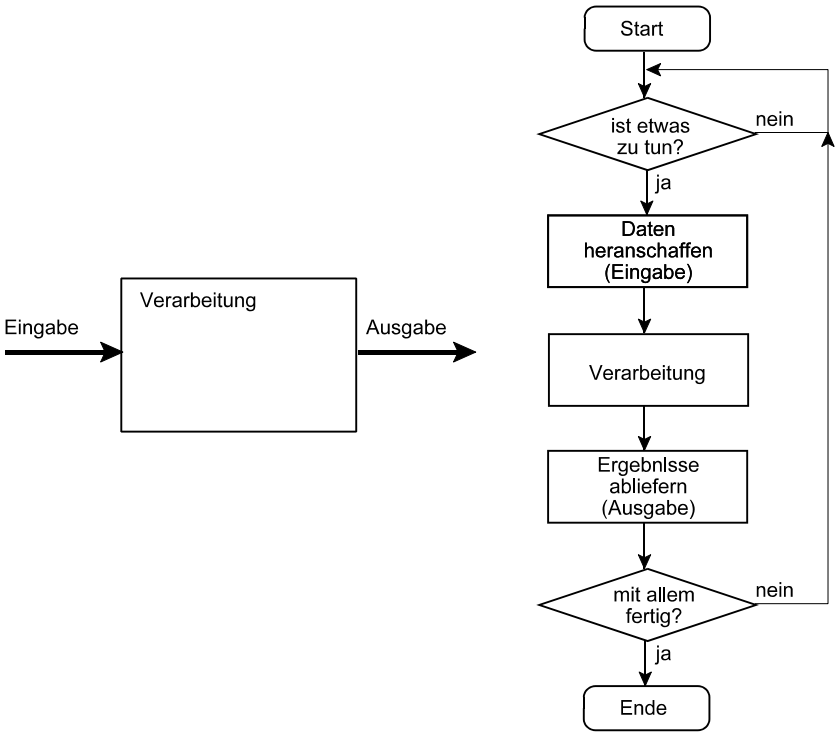


Abb. 1.1 Das grundsätzliche Ablaufschema der klassischen Datenverarbeitung

Simultane Abläufe in Embedded Systems
Viele Anwendungsaufgaben sind so geartet, daß sie nicht in das herkömmliche Programmschema Eingabe – Verarbeitung – Ausgabe passen bzw. nicht ohne weiteres auf dieses Schema zurückzuführen sind:

- es sind mehrere unabhängige Eingaben zu verarbeiten,
- es sind gleichzeitig mehrere Ausgaben zu liefern,
- es sind mehrere Informationswandlungen gleichzeitg auszuführen,
- es sind strikte Zeitbedingungen einzuhalten (ms oder weniger).

Mit anderen Worten: Mehrere Anwendungsfunktionen sind so auszuführen, daß sie der Anwendungsumgebung (ggf. einschließlich eines Nutzers/Bedieners) so erscheinen, als ob sie

gleichzeitig (simultan) ablaufen würden. Solche Probleme treten bereits in vergleichsweise einfachen Anwendungsaufgaben auf, die algorithmisch in keiner Weise kompliziert sind. Sie ergeben sich vor allem deswegen, weil die Wirkungen vorzugsweise mit universeller, programmierbarer Hardware zu erbringen sind[1)]. In Steuer- und Regeleinrichtungen, die als Einzwecksysteme – ohne programmierbare Universalrechner – entworfen werden, gibt es das in Rede stehende Problem praktisch nicht, weil für jede Aufgabe eine eigene Hardware-Anordnung vorgesehen wird, so daß der dem Anwendungsprobem innewohnende (inhärente) Parallelismus gleichsam von selbst in weitestgehendem Umfange zur Wirkung kommt (es müssen sich nicht alle Aufgaben den einen Prozessor teilen).

Das Entwicklungsziel besteht darin, mit der jeweils kostengünstigsten Hardware-Plattform auszukommen, die die Aufgabe in befriedigender Weise löst (niemand bezahlt für Weltrekorde). Hierzu gibt es verschiedene Lösungsprinzipien und gedankliche Ansätze (Philosophien):

- intuitive Rückführung auf sequentielle Programmabläufe,
- Zerlegung in mehrere, zeitgeschachtelt auszuführender Programme (Multitasking),
- Einrichtung virtueller Maschinen,
- Emulation einer fiktiven anwendungsbezogenen Hardware (worin – s. oben – das Problem gar nicht auftritt). Prinzip: als Blockschaltbild entwerfen, als Programm ausführen.
- Rückführung auf automatentheoretische Modelle (Finite State Machines),
- Nutzung mehrerer programmierbarer Einrichtungen (Multiprozessorsysteme),
- Verbundlösungen (programmierbare universelle Prozessoren + spezielle Hardware).

Jeder dieser Ansätze hat seine Berechtigung. Selbständig denken! Jeder Handwerker hat eine Vielzahl von Spezialwerkzeugen – und nicht nur einen einzigen Hammer ...

Die typische Praxis:
Was mit Software auf Wald- und Wiesen- (= Industriestandard-) Hardware zu lösen ist, auch so lösen. Nur dann über Wald- und Wiesen-Hardware hinausgehen, wenn unumgänglich notwendig.

Software
Die eigene Anwendung entscheidet – sonst wären wir ja keine Programmierer, sondern Einkäufer. Viele Probleme können auf der Anwendungsebene erledigt werden – es ist oftmals gar nicht notwendig, aufwendige Systemsoftware zu kaufen. Gekaufte Systeme:

- kosten Geld (Lizenzgebühren),
- brauchen Hardware-Ressourcen (vor allem: Speicherplatz),
- fressen Laufzeit (Overhead),
- machen Arbeit (Einarbeitungszeit, Umgehung ungünstiger Funktionsmerkmale (Workarounds)),
- enthalten Fehler, die wir nicht selbst beseitigen dürfen.

Wann lohnen sich fertige Systemplattformen?
Dann, wenn der geforderte Funktionsumfang so hoch ist, daß es offensichtlich nicht in Frage kommt, alles von Grund auf selbst zu programmieren:

- es sind Netzwerke und andere standardisierte Schnittstellen zu unterstützen,
- gängige (Standard-) Software ist einzusetzen,
- es ist mit Dateisystemen zu arbeiten,
- allgemein: wenn Einarbeitungszeit << Programmierzeit (um alles selbst zu programmieren).

---

1): Nicht selten ergeben sich solche Anforderungen weniger aus dem eigentlichen Anwendungsproblem, sondern daher, weil man dem Rechner möglichst viele – wenn nicht gar alle – Funktionen übertragen will (um Hardware einzusparen oder um zusätzliche Funktionen vorzusehen, etwa in Hinsicht auf Bedienkomfort, Kommunikation usw.).

Wenn immer möglich: eines nach dem anderen...
Soll heißen: eigene Innovationen jeweils auf ein einziges Gebiet beschränken; nicht vielerlei Neues gleichzeitig angehen, z. B.:

- Hardware + Software,
- System + Anwendung.

Solche Vorhaben kosten typischerweise viel mehr Zeit als ursprünglich geplant. Es läßt sich nicht alles parallel erledigen. Erst muß die die Plattform (Hardware, Systemsoftware) enwicklt werden, dann die Anwendung. Die Anwendungsprogrammierung erfordert Vorlauf seitens der Plattformentwicklung.

Grundsatz
Fertige Plattformen nur bestimmungsgemäß einsetzen – keinesfalls tricksen. Erforderlichenfalls Funktionen aufteilen (Partitionierung des Systems). Eine typische Aufteilung:

- Windows, Linux u. dergl. zur Bedienung, Datenspeicherung/Dateiverwaltung, Internetzugang usw.,
- Mikrocontroller für die Realzeitfunktionen.

Verbindung über standardisierte, allgemein übliche Schnitttstellen (seriell, Ethernet).

Die Gefahr beim Tricksen: daß es womöglich schon beim nächsten Release der Systemplattform nicht mehr funktioniert (Herr G. wird auf unsere Eigentümlichkeiten wohl kaum Rücksicht nehmen...).

Mehrere gleichzeitig laufende Programme in der EDV
Die ursprüngliche Verfahrensweise: es wird zu einer Zeit nur ein Programm ausgeführt, das eine Anwendungsaufgabe erledigt. Ggf. müssen, um ein Anwendungsproblem zu lösen, mehrere Programme nacheinander aufgerufen werden, wobei die Ergebnisse der vorhergehenden Programme zu Eingaben der nachfolgenden werden (Zwischenspeicherung).

Dieser Arbeitsablauf wurde bald als unbefriedigend empfunden, und man hat nach Lösungen gesucht, mehrere Programme gleichzeitig ausführen zu können. Typische Entwicklungsziele:

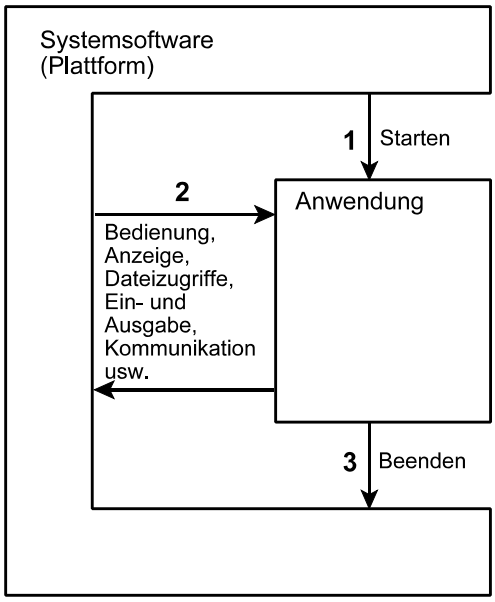
- Wartezeiten (vor allem während der Ein- und Ausgabe) sollten mit nützlicher Arbeit ausgefüllt werden,
- mehrere Anwender sollten die Maschine gleichzeitig nutzen können,
- der einzelne Anwender sollte mit mehreren Programmen gleichzeitig arbeiten und zwischen ihnen freizügig umschalten können.

Auf einem Prozessor kann aber zu einer Zeit nur ein Programm laufen. Die naheliegende Alternative – jedem Programm ein eigener Prozessor – ist offensichtlich unwirtschaftlich. Deshalb kam nur ein abschnittsweise zeitversetztes Abarbeiten der gleichzeitig lauffähigen Programme in Frage (ein Programmabschnitt zu einer Zeit). Die Umschaltung zwischen den Programmabschnitten wurde vorzugsweise mit Software implementiert (Betriebssysteme). Hardwarelösungen wurden zwar schon recht früh entwickelt (das klassische Beispiel: die E-A-Prozessoren des Großrechners CDC 6600 (1963)), haben sich aber bisher am Markt nicht durchsetzen können (sie sind aufwendiger und nicht so flexibel, haben aber auch bedeutsame Vorteile – deshalb versucht man sich immer wieder daran (Beispiel: Intels Hyper-Threading-Technologie)).

Über den Entwicklungsweg EDV – Prozeßrechentechnik – Mikroprozessorsysteme sind einschlägige Programmiermodelle und Systemkonzepte auch im Bereich der Embedded Systems wirksam geworden (Multitasking als Programmierphilosophie, Realzeitbetriebssysteme, Nutzung der typischen PC-Betriebssysteme).

2. Die Software-Plattform

Die Systemsoftware stellt praktisch eine Plattform zum "Laufenlassen" von Anwendungen dar; die einzelnen Anwendungen sind in diese Plattform gleichsam eingebettet (Abb. 2.1). Im allgemeinen Sprachgebrauch wird die Software-Plattform mit dem Betriebssystem gleichgesetzt (so spricht man von einer Windows-Plattform, einer Linux-Plattform usw.). In diesem Sinne gibt Abb. 2.2 einen Überblick über typische Funktionen von Software-Plattformen.



1 - das System startet die Anwendung (die Anwendung erhält hiermit praktisch die Verfügung über die Hardware); 2 - die Anwendung ruft Systemfunktionen bzw. Dienstleistungen auf und wird vom System aus beeinflußt; 3 - die Anwendung wird beendet (sie gibt hiermit die Hardware wieder frei).

Abb. 2.1 Die Anwendung ist in die Software-Pattform gleichsam eingebettet

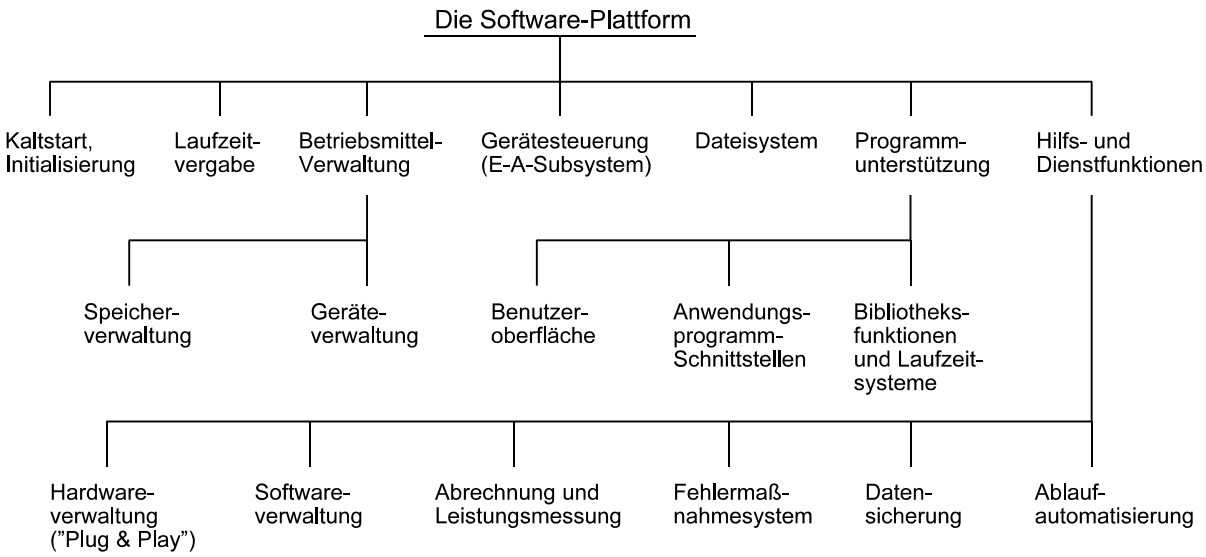


Abb. 2.2 Typische Komponenten einer Software-Plattform

Es gehört zur Systemphilosophie (und – vor allem – auch zum Marketing), welche Funktionen in welchen Programmen untergebracht sind und wie diese verkauft werden. Die Grenzfälle:

1. "alles aus einer Hand": es gibt einen einzigen Programmkomplex (das "Betriebssystem" im herkömmlichen Sinne), der von einem Hersteller angeboten wird,
2. "Modularität" (Baukastenprinzip): das eigentliche Betriebssystem hat nur einen minimalen Umfang (es betrifft z. B. lediglich die Funktionen "Laufzeitvergabe" und "Betriebsmittelverwaltung"). Alle anderen Funktionen werden von zusätzlichen Programmen ausgeübt (und die können womöglich von verschiedenen Anbietern bezogen werden).

Heutzutage gibt es wohl kaum eine marktgängige Plattform, zu der nicht Komponenten von Drittanbietern erhältlich sind – nur der Anteil macht den Unterschied.

Wie Systemplattformen die gleichzeitige Auführung mehrerer Programme unterstützen
Die folgenden Fragen gehören zu den ersten, die sich der Entwickler einer Software-Plattform vorlegen muß:

1. wieviele Nutzer (User) sollen gleichzeitig mit dem System arbeiten können?
2. wieviele Anwendungsprogramme (Tasks) sollen gleichzeitig lauffähig sein?

Tabelle 2.1 nennt die grundsätzlichen Antworten, die es auf beide Fragen gibt.

Anzahl der Nutzer	Anzahl der gleichzeitig lauffähigen Anwendungsprogramme	Bezeichnung	Beispiel
1	1	Single User, Single Task	DOS
1	mehrere	Single User, Multitasking	Windows 3.x/95/98/Me
mehrere	mehrere	Multi User, Multitasking	Windows NT und Nachfolger, Unix

Tabelle 2.1 Unterscheidung von Software-Plattformen nach den Arbeitsmöglichkeiten für Nutzer und Anwendungsprogramme

Single User, Singe Task
Ein einzelner Nutzer sitzt vor dem Computer, und es kann jeweils nur ein Anwendungsprogramm laufen. Der einfachste Fall, der auch an die Laufzeitvergabe die geringsten Anforderungen stellt. Die Plattform wirkt lediglich als Programmstarter (Program Launcher) für die jeweilige Anwendung.

Single User, Multitasking
Ein einzelner Nutzer sitzt vor dem Computer, es können aber mehrere Anwendungsprogramme gleichzeitig lauffähig sein. Hier muß die Plattform die Laufzeit wirklich verwalten. Grundsätzlich handelt es sich darum, die Laufzeit sozusagen stückweise zu verteilen. Beispielsweise läßt das System Programm A 100 ms lang laufen. Dann greift die Laufzeitverwaltung (der Scheduler) ein und teilt Programm B 50 ms zu, dann Programm C 200 ms usw. Dem Benutzer erscheint es so, als ob alle 3 Programme praktisch gleichzeitig arbeiten. Das zu organisieren ist nicht einfach:

- das System muß sich merken, an welcher Stelle einem Programm die Laufzeit entzogen wurde,
- sinngemäß sind Maschinenzustände, Zwischenergebnisse usw. zu retten (beim Entzug der Laufzeit) und wieder einzustellen (wenn dem Programm erneut Laufzeit zugesprochen wird),

- die Speicher- und Geräteverwaltung muß mit der Laufzeitvergabe in Einklang stehen. Beispiel: Programm A hat die ersten Zeichen auf den Drucker ausgegeben. Nun erhält Programm B Laufzeit und möchte auch drucken. Es ist ersichtlich, daß – sollte die Plattform das so durchgehen lassen – auf dem Papier ein Wust durcheinandergewürfelter Zeichen zu sehen sein würde.
- es darf nicht sein, daß bestimmte Programme stets Laufzeit erhalten und andere kaum oder gar nicht zum Zuge kommen,
- es darf nicht sein, daß das Programm, das gerade läuft, Daten eines anderen Programms verfälscht.

Daß es nur einen Nutzer gibt, vereinfacht die Aufgabe etwas, denn es ist nur die Kommunikation über einen einzigen Satz von Bedien- und Anzeigeeinrichtungen zu unterstützen.

Multi User, Multitasking
Mehrere unabhängige Nutzer können gleichzeitig mit dem System arbeiten. Typischerweise sind an ein solches System mehrere Bildschirme, Tastaturen usw. angeschlossen (Mehrplatzsysteme). Diese Auslegung stellt die höchsten Anforderungen an die Plattform: die einzelnen Nutzer sind angemessen mit Laufzeit zu versorgen – und sie sind voreinander zu schützen (es darf z. B. nicht sein, daß ein Nutzer auf vertrauliche Datenbestände eines anderen zugreifen kann).

Was ist eine Task?
Der Begriff "Task" wird oft in allgemeinem Sinne verwendet: 1 Task = 1 in sich abgeschlossenes, lauffähiges (Anwendungs-) Programm. Darüber hinaus liegt es aber auch nahe, Teilprogramme innerhalb einer Anwendung im Multitasking zu "fahren". Hat dies überhaupt einen Sinn, wenn es nur einen Prozessor gibt? – Durchaus:

- bestimmte Funktionen können sozusagen nebenher bzw. "im Hintergrund" erledigt werden. Sind derartige Funktionen als unabhängige Tasks implementiert, so sorgt der Scheduler schon dafür, daß sie auch Laufzeit erhalten; es ist also nicht erforderlich, die entsprechenden Aufrufe jeweils auszuprogrammieren (Beispiel: die automatische Datensicherung in gewissen Zeitabständen).
- bei Übergang auf ein System mit mehreren Prozessoren können die einzelnen Tasks auf verschiedene Prozessoren verteilt werden, wodurch sich eine tatsächliche Leistungssteigerung ergibt.

Wenn es um bestimmte Plattformen geht, müssen wir uns die jeweils kennzeichnenden Begriffe genauer ansehen.

Beispiel: Windows NT und Nachfolger
Eine Task im allgemeinen Sinne (in sich abgeschlossenes, lauffähiges (Anwendungs-) Programm) heißt *Prozeß* (Process), ein unabhängig lauffähiges Teilprogramm innerhalb eines Prozesses heißt *Thread* (wörtlich = Faden). Das gleichzeitige Ausführen mehrerer Threads nennt man Multithreading. In der Terminologie anderer Anbieter wird hingegen der Thread Task genannt und unsere Task Program. Dann heißt das Multitasking "Multiprogramming", und das Multithreading heißt "Multitasking".

Nutzer und Anwendungen in Embedded Systems
Der typische Nutzer des PCs sitzt die meiste Zeit vor dem Bildschirm. In Embedded Systems hingegen sind Bedienung und Anzeige nur Teilaufgaben – und nicht selten vergleichsweise untergeordnete (der Mensch soll schließlich sowenig wie möglich eingreifen müssen (Automatisierung)). Die typischen "Nutzer" sind hier vor allem zu steuernde Prozesse, Regelungsaufgaben usw., wobei für jeden derartigen "Nutzer" bestimmte Anwendungen auszuführen sind (Meßwerterfassung, Steuerungsabläufe, Regelungsalgorithmen usw.). Die Übertragbarkeit des grundsätzlichen Programmiermodells ist offensichtlich. In der Praxis haben wir es mit zwei Formen der Ausnutzung zu tun:

1. die Übernahme von Prinzipien (als Anregungen für eigene Lösungen),
2. die direkte Nutzung entsprechender Systeme (Windows, Unix usw.)

Multitasking in Embedded Systems
Es gibt mehrere Möglichkeiten, die einschlägigen Prinzipien anzuwenden:

1. Multitasking als Programmierphilosophie in der Anwendungsprogrammierung
Wir zerlegen das Anwendungsproblem in Tasks, verwenden aber kein besonderes Betriebssystem, sondern organisieren die Taskverwaltung im Rahmen der Anwendungsprogrammierung. Typische Vorteile:

- in jeder Systemumgebung durchführbar,
- solche Anwendungen können unter einem gegebenen Betriebssystem (z. B. DOS oder Windows oder Linux) laufen,
- höhere Programmiersprachen oftmals durchgehend einsetzbar (Assemblerprogrammierung nur selten notwendig),
- auch in kleinsten Konfigurationen lauffähig (Stand-Alone-Programme für kleine Mikrocontroller),
- nur Anwendungsprogrammierung (Einarbeitung in die Spitzfindigkeiten der Systemprogrammierung nicht erforderlich),
- es ist alles unter programmseitiger Kontrolle (keine Probleme mit unerwarteten Interrupts, Fehlern in der Unterbrechunsbehandlung usw.).

Der grundsätzliche Nachteil: ein vergleichsweise grobes Realzeitraster. Latenzzeiten typischerweise im Millisekundenbereich (keine Unterbrechung von Befehlsabläufen, sondern programmgesteuerte Abfrage/Umschaltung).

2. Elementare Multitasking-Kernfunktionen werden selbst programmiert
Wir kaufen kein besonderes Betriebssystem, sondern programmieren die Schnittstellen zwischen Hard- und Software selbst. Viele Anwendungsprobleme lassen sich mit einfachen Abläufen der Unterbrechungsbehandlung lösen (als Beispiel vgl. Kapitel 8); kompliziertere Multitasking-Funktionen werden gar nicht benötigt. Typische Vorteile:

- in jeder Systemumgebung durchführbar,
- auch in kleinsten Konfigurationen lauffähig (Stand-Alone-Programme für kleine Mikrocontroller),
- geringste Latenzzeiten (Mikrosekunden). Der für umfangreichere Systeme typische Overhead entfällt; somit lassen sich auch mit kleinen Mikrocontrollern usw. scharfe Realzeitanforderungen erfüllen.

Typische Nachteile:

- Rückgriff auf Assemblerprogrammierung erforderlich,
- Fehlersuche (Debugging) schwieriger.

3. Multitasking auf Grundlage voll ausgebauter Systemplattformen
Bei hohen Anforderungen an den Funktionsumfang der Anwendung werden zumeist kommerzielle Realzeitbetriebssysteme oder Systeme aus dem PC-Bereich (Windows, Linux) eingesetzt. Die Entscheidung für ein solches System hat zur Folge, daß eine hinreichend ausgestattete Hardware-Plattform gewählt werden muß (z. B. ein Industrie-PC) – mit kleinen Mikrocontrollern wird es dann nichts mehr. Der Vorteil liegt vor allem in der Nutzbarkeit fertiger Software. Auf PCs beruhende Plattformen sind in dieser Hinsicht allen anderen Angeboten weit überlegen, haben aber auch typische Probleme:

- lausige Latenzzeiten (etliche Millisekunden – wobei auch noch so viele GHz und GBytes nicht viel helfen),
- mangelhafte Zuverlässigkeit (Abstürze),
- fortlaufende Betreuung erforderlich (Updates).

Ein naheliegender Ausweg: Partitionierung, d. h. Aufteilung der auszuführenden Funktionen auf mehrere Teilsysteme. Kleinere Konfigurationen bestehen z. B. aus einem Industrie-PC (für Bedienung, Anzeige, Dateisystem, Netzwerkkommmunikation usw.) und angeschlossenen Mikrocontrollern, die die zeitkritischen Funktionen ausführen. Abb. 2.3 veranschaulicht eine extreme Ausführung.

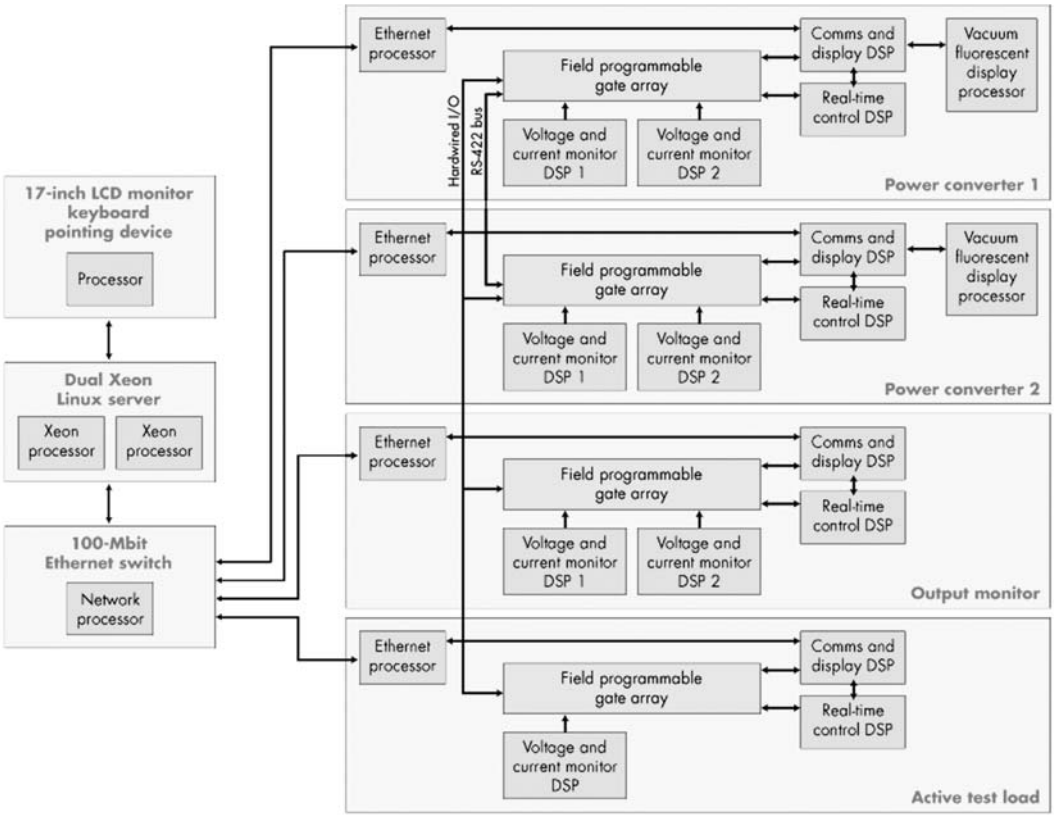


Abb. 2.3 Das obere Ende (nach Electronic Design): Dieses System enthält insgesamt 29 Prozessoren. Es sind drei verschiedene Betriebssysteme im Einsatz. Zur Programmentwicklung wurden vier Programmiersprachen verwendet. Man beachte die Kopplung über standardisierte Schnittstellen (Ethernet)

3. Grundlagen des Multitasking

Die einzelne Task ist ein für sich lauffähiges Programm. Sollen in einem Prozessor mehrere Tasks zeitverschachtelt ausgeführt werden, so sind stets folgende Probleme zu lösen:

- Taskzustände: in welchen Zuständen können sich die einzelnen Tasks befinden?
- Taskumschaltung: was läuft ab, wenn der einen Task Laufzeit entzogen wird und eine andere Laufzeit erhält?
- Aktivierung: wie wird die Ausführung einer bestimmten Task gestartet?
- Laufzeitzuteilung: auf welche Weise und nach welchen Gesichtspunkten wird den einzenen Tasks ihre Laufzeit zugeteilt?

Hinweis:
Diese Probeme sind grundsätzlicher Natur; sie müssen in allen Multitasking-Systemen irgendwie gelöst werden. Es gibt allerdings beträchtliche Unterschiede hinsichtlich der Komplexität (von der einfachen Inerruptbehandlung in Mikrocontrollern bis hin zu Windows usw.) und auch in den Begriffsbildungen (jedes System, jeder Lehrbuchautor hat seine eigene Terminologie).

3.1 Taskzustände

In der Rechnerarchitektur bezeichnet der Begriff "Task" allgemein die Umgebung (Register, Speicherbereiche usw.), in der ein einzelnes Programm lauffähig ist. Multitasking bedeutet, daß mehrere Programme gleichzeitig lauffähig sein können. In einem Einzelprozessor kann aber jeweils nur ein Programm tatsächlich arbeiten. Die betreffende Task heiß die arbeitende (Running) Task. Andere Programme können arbeitsfähig sein, das heißt, mit der Befehlsausführung beginnen, sobald die betreffende Task Laufzeit erhält. Solche Tasks heißen aktive (Busy) Tasks. Weiterhin ist es möglich, Task-Umgebungen im Speicher zu halten, die nicht unmittelbar arbeitsfähig sind und sie nur unter bestimmten Bedingungen arbeitsfähig zu machen. Solche Tasks heißen inaktive (Not Busy bzw. Idle Tasks).

Im einfachsten Fall hat die Task nur zwei Zustände (Abb. 3.1): den Ruhezustand (Idle) und den Laufzustand (Running). Dieses Schema ist dann anwendbar, wenn das in der Task laufende Programm nur vergleichsweise wenig Zeit braucht, um seine Arbeit zu erledigen. Richtwerte zur maximalen Laufzeit: ca. 5...50 ms[1)]. Hierbei bestimmt die Laufzeit der jeweils aktiven Task die Latenzzeit der anderen Tasks (das jeweils aktive Programm muß erst durchgelaufen sein, bevor ein anderes starten kann). Als Beispiel vgl. Kapitel 8 (wobei wir die Interruptroutinen als Tasks ansehen wollen).

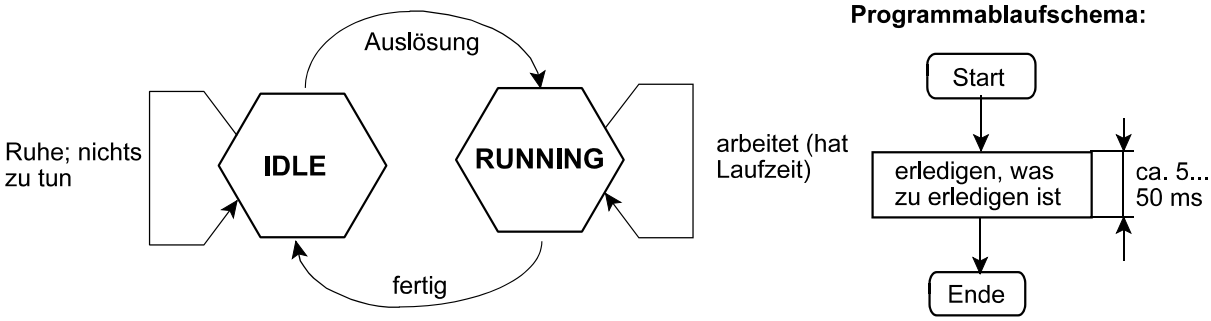


Abb. 3.1 Der einfachste Fall: nur zwei Taskzustände

In folgenden Fällen ist es erforderlich, der jeweils laufenden Task Laufzeit zu entziehen (Taskumschaltung):

- es gibt Tasks mit längerer Laufzeit,
- es gibt Tasks, die "ewig" laufen (Endlosschleifen),
- die Latenzzeiten sind zu hoch.

Die grundsätzliche Lösung besteht darin, einen weiteren Taskzustand einzuführen (Abb. 3.2): die Task ist zwar noch aktiv, hat aber momentan keine Laufzeit (Busy). In höherentwickelten Systemen gibt es weitere Taskzustände, die den elementaren Zuständen gemäß Abb. 3.1 hinzugefügt wurden. Derartige Zustände haben mit Feinheiten der Laufzeitvergabe, mit der Unterstützung des Stromsparens und mit der Einhaltung von Realzeitanforderungen zu tun. Hierbei gibt es beträchtliche Unterschiede zwischen den einzelnen Systemen (welche Zustände es gibt und wie sie eingeleitet und verlassen werden). Abb. 3.3 veranschaulicht ein Beispiel.

1) Richtwerte: 8 ms = Netzhalbwelle bei 60 Hz, 50 ms = 20 Bedienfeldabfragen/s

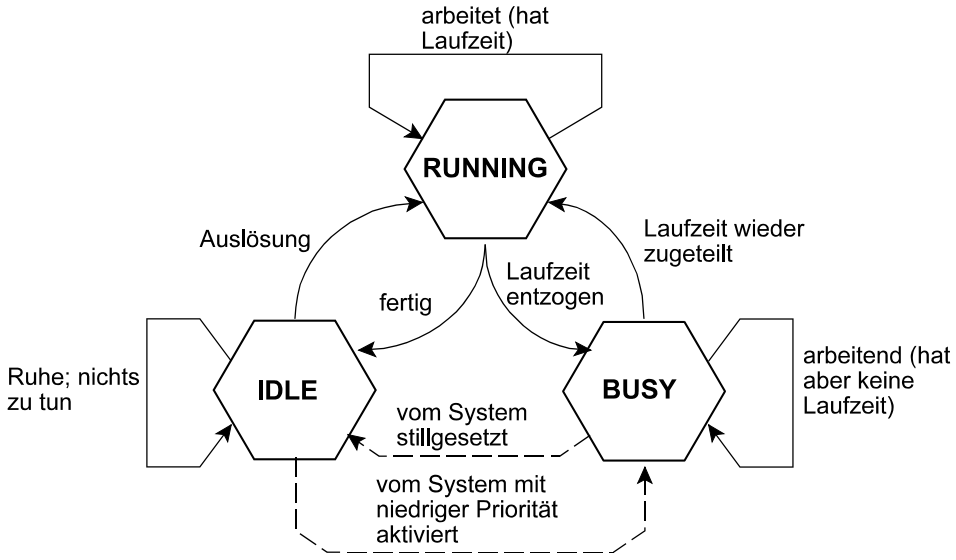


Abb. 3.2 Die grundsätzlichen Taskzustände beim Mutitasking mit Laufzeitverwaltung

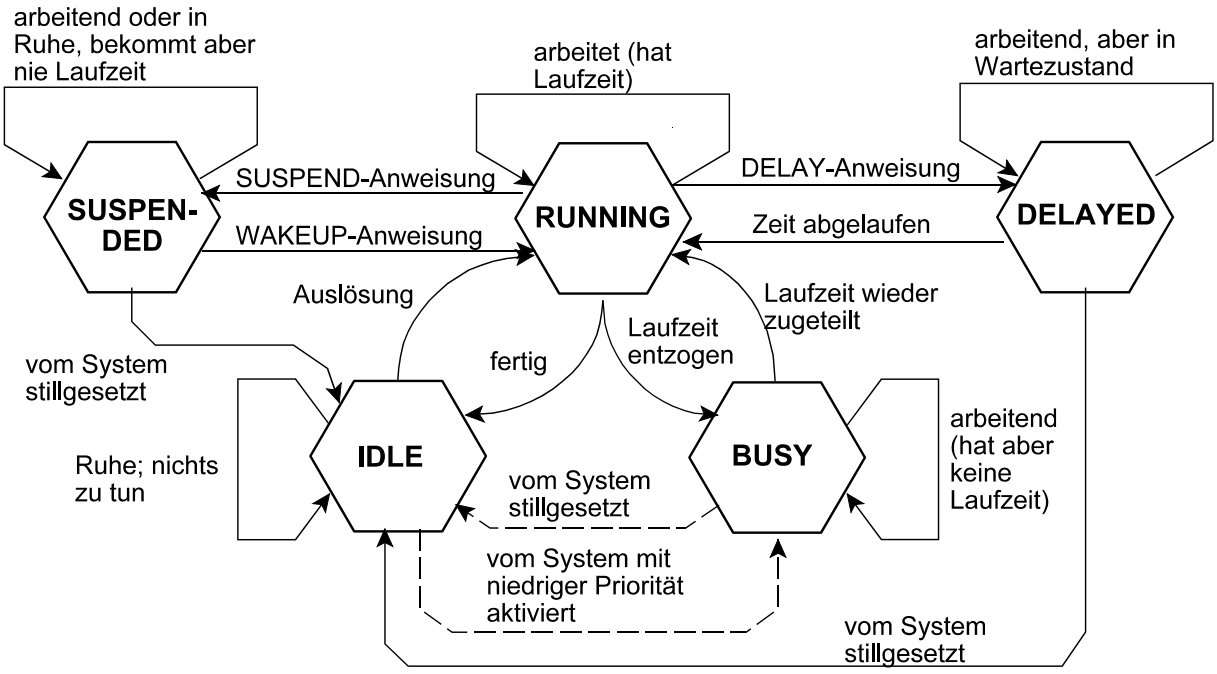


Abb. 3.3 Typische erweiterte Taskzustände

Im Beispiel von Abb. 3.3 gibt es zwei zusätzliche Zustände:

1. Suspendiert (Suspended)
Befindet sich eine Task in diesem Zustand, so bekommt sie nie Laufzeit. Einleiten: mit Systemanweisung SUSPEND. Verlasssen: von einer anderen Task aus mit Systemanweisung WAKEUP (Normalfall) oder durch direkten Eingriff des Systems (z. B. zwecks Fehlerbehandlung). Typischer Nutzungsfall: Stromsparen. Wenn die von der betreffenden Task gesteuerte Hardware in einen Stromsparzustand versetzt wird, kann auch die zugehörige Task stillgelegt werden; es ist dann nicht mehr nötig, sie bei der Laufzeitvergabe zu berücksichtigen.

2. Verzögert (Delayed)
Die Task ist noch aktiv, erhält aber solange keine Laufzeit, bis eine bestimmte Verzögerungszeit abgelaufen ist. Einleiten: mit Systemanweisung DELAY (wobei die Verzögerungszeit (z. B. in Millisekunden)) als Parameter übergeben wird. Verlasssen: nach Ablauf der Verzögerungszeit (Normalfall) oder durch direkten Eingriff des Systems (z. B. zwecks Fehlerbehandlung). Typischer Nutzungsfall: es sind soundsoviel Millisekunden zu warten. Währenddessen können andere Tasks Laufzeit erhalten. Ein weiterer Vorteil gegenüber dem programmseitigen Auszählen des Zeitintervalls (Wartschleife): die Verzögerungszeit wird unabhängig vom Programmablauf mit Hilfe eines Systemzeitgebers bestimmt. Sie kann deshalb – unabhängig von den Zeitverhältnissen der Befehlsausführung (Instruction Timing) – mit hoher Genauigkeit eingehalten werden.

3.2 Taskumschaltung

Multitasking-Vorkehrungen stellen unabhängige Umgebungen für unabhängige Programme zur Verfügung (Abb. 3.4). Nur ein Programm kann aber zu einer Zeit laufen. Wenn ein Programm läuft, so belegt es die Register des Prozessors. Taskumschaltung heißt, die entsprechenden Angaben zu retten und durch die Angaben jener Task zu ersetzen, die als nächste Laufzeit erhalten soll (Abb. 3.5).

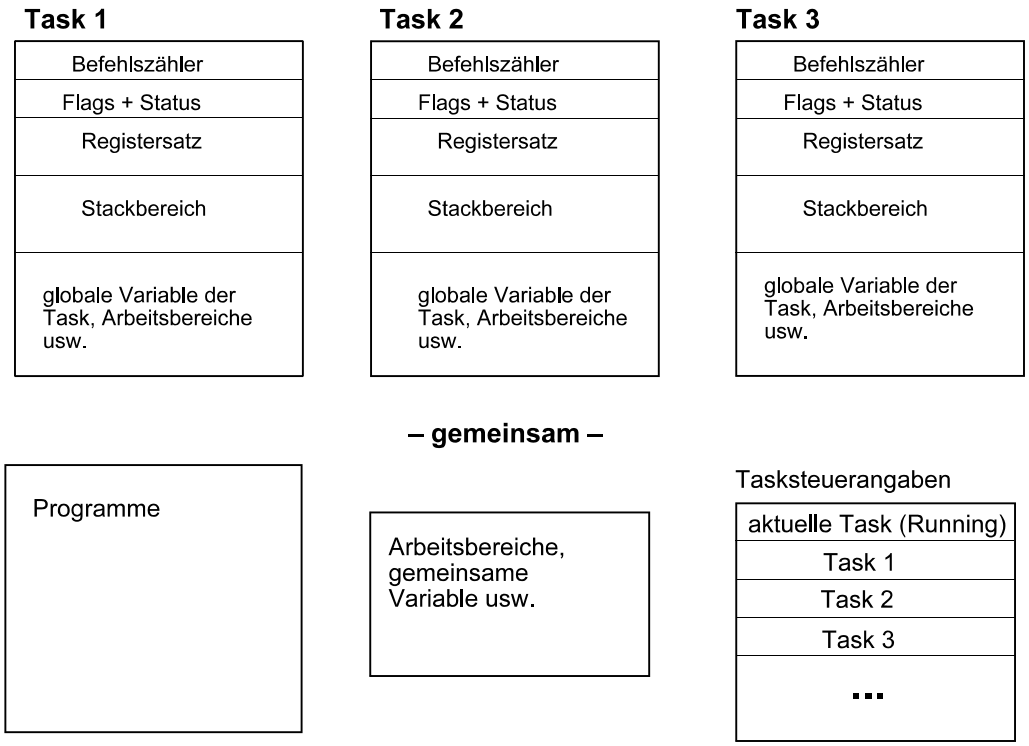


Abb. 3.4 Task-Umgebungen im Prozessor

Die "Umgebung" (der Kontext) einer Task besteht aus den Inhalten der Prozessorregister, ihrem Stackbereich, eigenen Arbeitsbereichen usw. Hinzu kommen Steuerangaben, die den Taskzustand beschreiben. Für jede Task sind Speicherbereiche erforderlich, die die Angaben der Task-Umgebung aufnehmen können.

Wieviel Zeit eine Taskumschaltung erfordert, hängt vor allem vom Umfang der jeweiligen Taskumgebung ab. Komfortable Systeme haben extrem umfangreiche Taskumgebung. So müssen beim Umschalten zwischcn zwei Windows-Anwendungen nicht nur die Prozessorregister, sondern auch Fenster, geöffnete Dateien, anhängige Kommunikationsvorgänge usw. berücksichtig werden – deshalb dauert eine solche Kontextumschaltung so viele Milllisekunden. Demgegenüber erfordert es viel weniger

Zeit, zwischen zwei Teilprogrammen (Threads) innerhalb einer Anwendung umzuschalten.

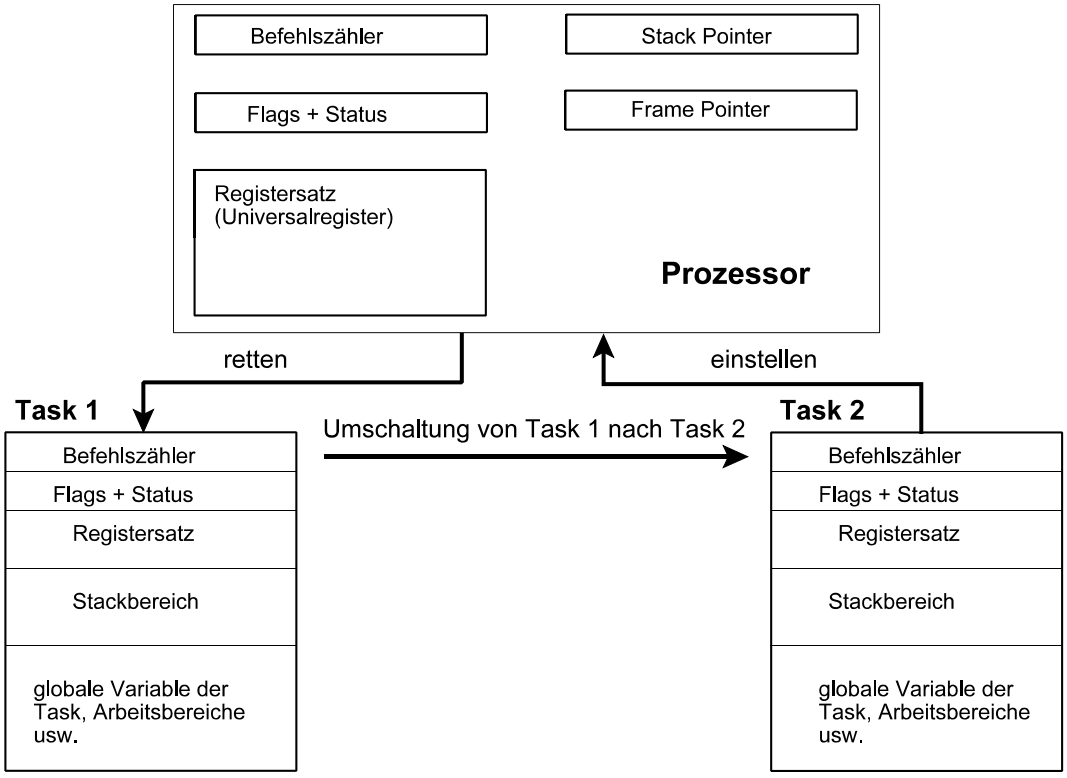


Abb. 3.5 Der grundsätzliche Ablauf einer Taskumschaltung

Jede Task hat einen Rettungsbereich für die Prozessorregister. Wird einer Task Laufzeit entzogen, so werden die Registerinhalte in den betreffenden Rettungsbereich transportiert. Es ist "alles" zu retten (einschließlich Befehlszähler, Stackpointer usw.), so daß später – nach erneuter Laufzeitzuteilung – die Task ihre Arbeit gleichsam lückenlos wieder aufnehmen kann. Die Registerinhalte der Task, die Laufzeit erhalten soll, werden aus derem Rettungsbereich in den Prozessor gebracht. In diesem Zusammenhang erledigt sich auch die Umschaltung des Stacks, der privaten Arbeitsbereiche usw. (durch enstprechendes Laden des Stackpointers, des Frame Pointers, der Adreßregister usw.). Als letztes wird der Befehlszähler aus dem Rettungsbereich geladen. Anschließend wird der erste Befehl der neuen Task ausgeführt.

Hinweise:

1. Manche Prozessoren unterstützen die Taskumschaltung durch besondere Vorkehrungen (Beispiel: die Taskzustandssegmente (TSS) der IA-32-Prozessoren).
2. Schnelle Taskumschaltungen erfordern besondere Sorgfalt bei der Festlegung der Taskumgebungen (was ist zu retten, was nicht?).
3. Die Taskumschaltung ist gelegentlich "zu Fuß" auszuprogrammieren (Assemblerprogrammierung in genauer Anpassung an die jeweiligen Gegebenheiten und Anforderungen); generelle Vorkehrungen (z. B. die Unterstützung von Taskzustandssegmenten) sind manchmal zu langsam.

3.3 Aktivierung

Es gibt zwei grundsätzliche Verfahren, um die Abarbeitung von Tasks (Aktivierung, Programmstart) zu veranlassen:

1. Abfrage (Polling),
2. Unterbrechungsauslösung (Interrupt).

Die Abfrage beruht auf einer Programmschleife, die endlos umläuft und dabei alle Bedingungen abfragt, die das Starten von Tasks veranlassen können (Abb. 3.6). Wurde eine solche Bedingung erkannt, so verzweigt die Abfrageschleife zur betreffenden Task. Hat die Task ihre Arbeit beendet, verzweigt sie zur Abfrageschleife zurück.

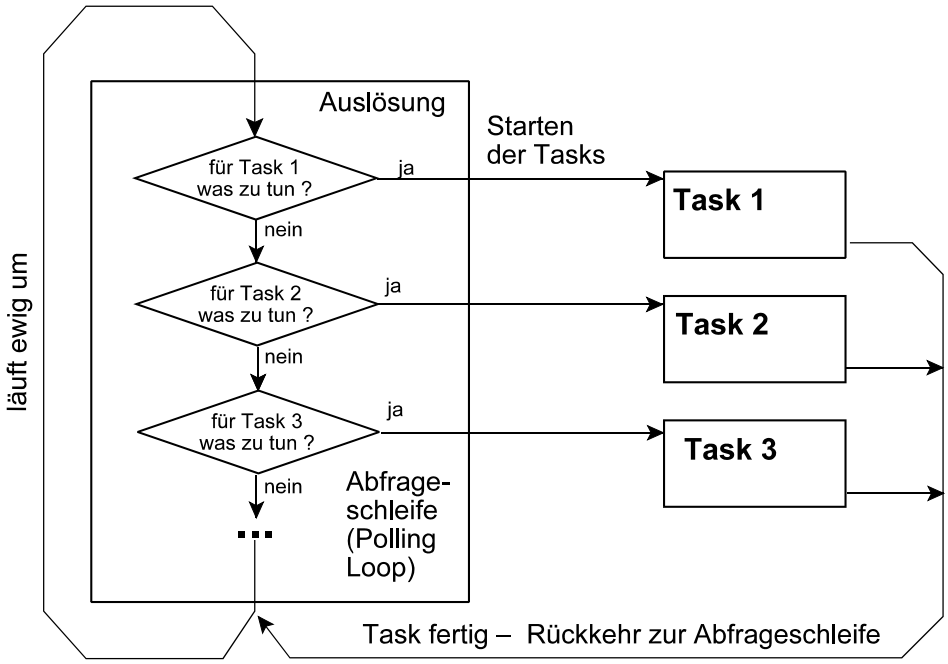


Abb. 3.6 Programmstart durch Abfrage

Dieses Schema unterscheidet sich auf den ersten Blick nicht von der herkömmlichen, gleichsam naiven Anwendungsprogrammierung (vgl. Abb. 1.1). Der Übergang vom gewöhnlichen Anwendungsprogramm mit Abfrageschleife zur Multitasking-Organisation ist fließend – es ist vor allem eine Frage der Größenordnung und der internen Organisation:

- es handelt sich um einzelne Programme (die ggf. unabhängig voneinander compiliert werden können),
- jedes Programm hat eigene Arbeitsbereiche, deren Inhalt auch dann erhalten bleibt, wenn es inaktiv ist,
- es gibt reguläre, wohldefinierte Schnittstellen zwischen der Abfrageschleife und den Tasks,
- die Abfrageschleife ist eine zentrale Systemfunktion (und läuft ggf. im Systemzustand (mit unbeschränkten E-A-Zugriffen usw.)), die Tasks sind Anwendungsprogramme.

In einer alternativen Auslegung (Abb. 3.7) gibt es keinen Trennung zwischen Abfrageschleife und Tasks. Vielmehr hängen alle Tasks gleichsam an einem Faden und bilden somit selbst eine Endlosschleife (von Task 1 geht es weiter zu Task 2, von dort zu Task 3 uw. und schließlich wieder zu Task 1). Hierbei kann man jede Task als herkömmliches Anwendungsprogramm (mit eigener Abfrage) auslegen (was gelegentlich ein Vorteil ist).

Diese einfachen Formen der Abfrageorganisation führen zu langen Latenzzeiten (schlimmstenfalls ein kompletter Schleifenumlauf bei maximaler Laufzeit einer jeden Task). Auch dürfen die einzelnen Tasks

ihrerseits keine Endlosschleifen enthalten.

Abhilfe: es werden echte Umschaltfunktionen (mit Kontextrettung im Sinne von Abb. 3.5) vorgesehen (Abb. 3.8).

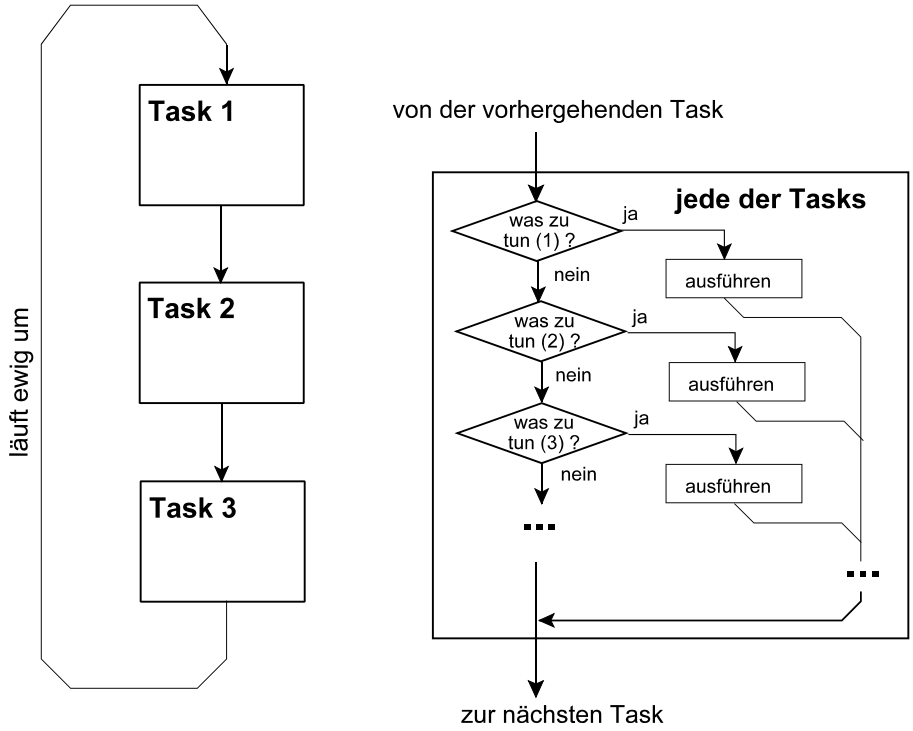


Abb. 3.7 Programmstart durch Abfrage – eine alternative Organisationsform

Abb. 3.8 veranschaulicht eine einzelne Task. Befindet sie sich im Ruhezustand, so werden – vgl. Abb. 3.7 – die einzelnen Bedingungen abgefragt. Befindet sie sich nicht im Ruhezutand. so wird die bisherige Arbeit fortgesetzt. Dazu ist der aktuelle Taskzustand wieder einzustellen (vgl. Abb. 3.5). In Abläufe, die viel Laufzeit brauchen, sind gelegentlich Systemaufrufe (BREAK) einufügen, die den anderen Tasks Laufzeit zukommen lassen. Im Beispiel leistet BREAK folgendes:

- die Task wird als arbeitend gekennzeichnet,
- die Task-Umgebung (Kontext) wird gerettet (vgl. Abb. 3.5),
- es wird die nächste Task aufgerufen (vgl. Abb. 3.7)

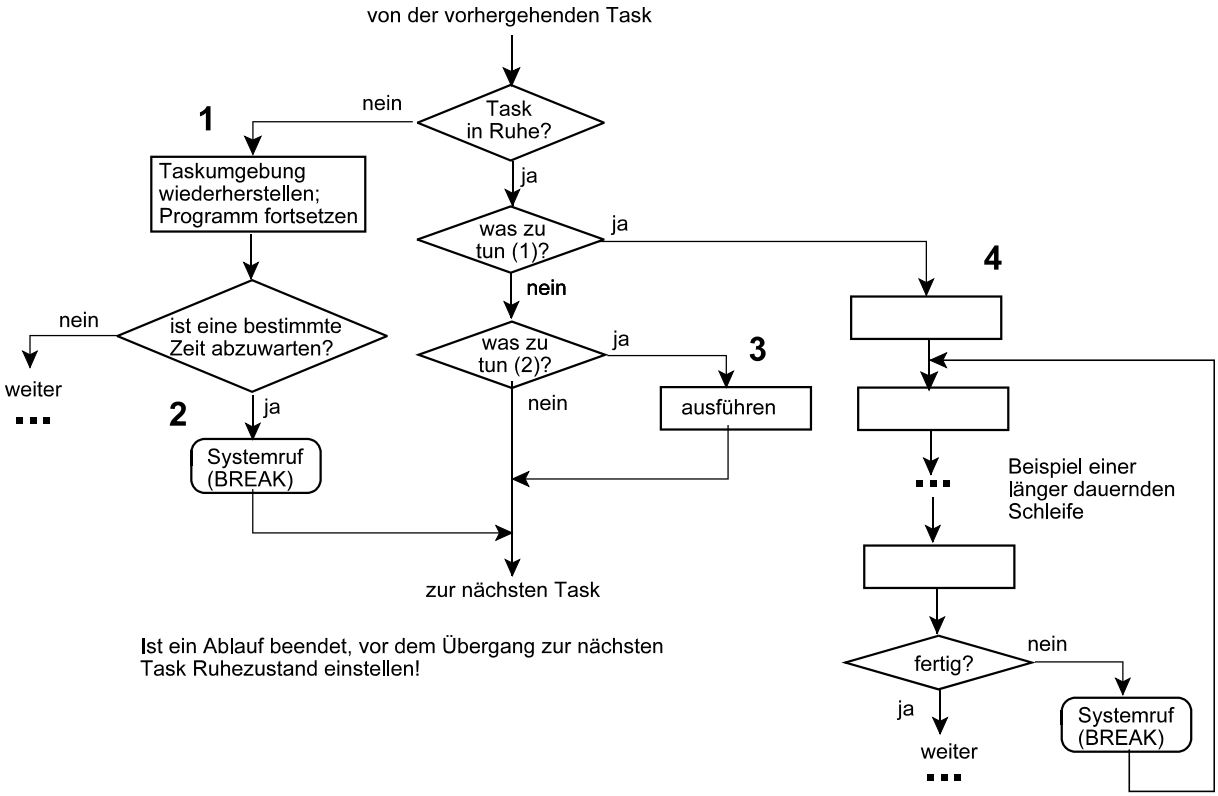
Bei geschickter Gestaltung kann man auf diese Weise Latenzzeiten < 1 ms erreichen.

Braucht man kurze Latenzzeiten unabhängig von laufenden Programm, muß man die Task über Unterbrechungsauslösung (Interrupt) starten (Abb. 3.9).

Ein Interrupt ist im Grunde ein hardwareseitig erzwungener Unterprogrammaufruf, wobei die Startadresse des Unterprogramms in einem Prozessorregister oder in einer im Speicher untergebrachten Interrupttabelle gehalten wird (Näheres in Kapitel 7) . Die Auslösung erfordert eine entsprechende Hardware. In einfachen Fällen genügt es, die entsprechende Signalleitung an einen Interrupteingang des Controllers oder Prozessors anzuschließen. Ansonsten sind zusätzliche Schaltmittel erforderlich (Interruptcontroller).

In manchen Anwendungfällen können alle Tasks über Interruptauslösung gestartet werden. Wenn nichts zu tun ist (Ruhezustand), läuft im Prozessor eine einfache Endloschleife.

Der Interrupt ist eine bloße Ja-Nein-Angelegenheit – er kann im Grunde nur ein einziges Bit an Information vermitteln (nämlich daß das betreffende Signal aufgetreten ist). Deshalb kommt man schon in vergleichsweise einfachen Fällen nicht mit Interrupts allein aus. Beispiel: das System hat ein Bedienfeld mit 64 Tasten. Soll die Tastenbetätigung allein über Interrupts erfaßt werden, müßte jede Taste einen eigenen auslösen; es wären also 64 verschiedene Interrupts erforderlich. Der Ausweg: man verwendet Sammel-Interrupts, die durch Parameter ergänzt werden. Diese sind typischerweise programmtechnisch einzulesen und auszuwerten (Beispiel: das Bedienfeld meldet die Tatsache einer Tastenbetätigung über einen einzigen Interrupt und stellt zudem einen Tastencode bereit, der angibt, welche Taste betätigt wurde). Die Interruptauslösung erfordert also oftmals vorgeordnete Aufbereitungs- und Abfragevorgänge. Es liegt nahe, hierzu auch den Prozessor heranzuziehen – er hat im Ruhezustand (vgl. Abb. 3.9) ohnehin nichts zu tun (Abb. 3.10).



1 - Fortsetzen der bisherigen Arbeit; 2 -wenn länger zu warten ist, können auch andere Tasks Laufzeit erhalten; 3 - in kurzer Zeit auszuführende Funktion (muß keine Laufzeit abgeben); 4 - diese Funktion dauert länger. Deshalb sind Systemaufrufe eingefügt, die anderen Tasks Laufzeit zukommen lassen.

Abb. 3.8 Abfrageorganisation mit programmierbare Kontextrettung

Eine typische Arbeitsteilung: Bedingungen, bei deren Verarbeitung es auf die Latenzzeit kaum ankommt (einige Millisekunden mehr schaden nichts), werden durch Abfrage erkannt (Beispiel: Bedienvorgänge). Bedingungen, auf die sofort reagiert werden muß, veranlassen hingegen Unterbrechungen (Beispiel: Nulldurchgang der Netzwechselspannung).

Abb. 3.10 veranschaulicht zudem, wie eine Interrupttabelle als einheitliche Schnittstelle zum Starten von Tasks eingesetzt werden kann. Alle Startadressen stehen in der Interrupttabelle, die entweder hardwareseitig oder programmseitig adressiert wird. Manche Prozessoren unterstützen sog. Software-Interrupts. Das sind verkürzte Unterprogrammrufbefehle, die sich auf die Interrupttabelle beziehen (Beispiel: Intel x86/IA-32).

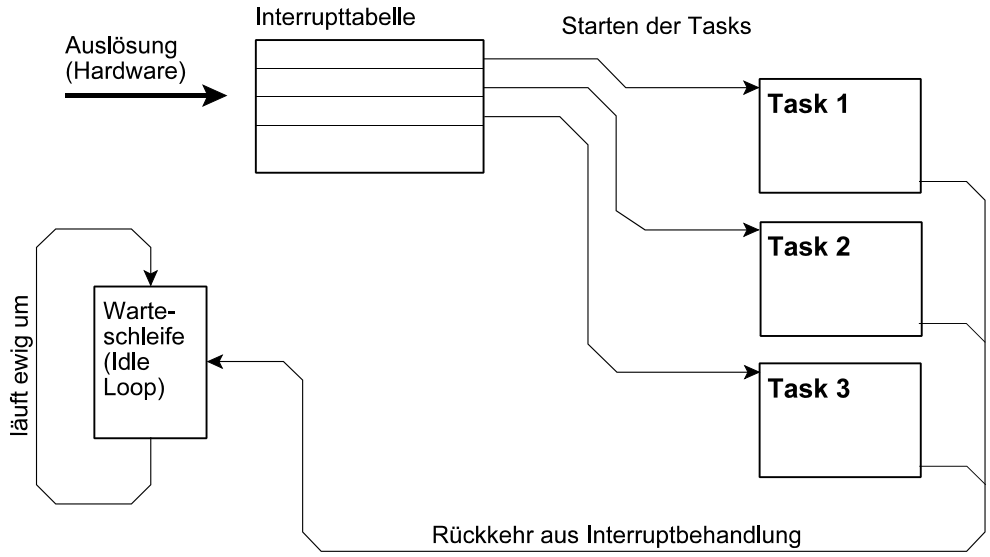


Abb. 3.9 Programmstart durch Unterbrechungsauslösung (Interrupt)

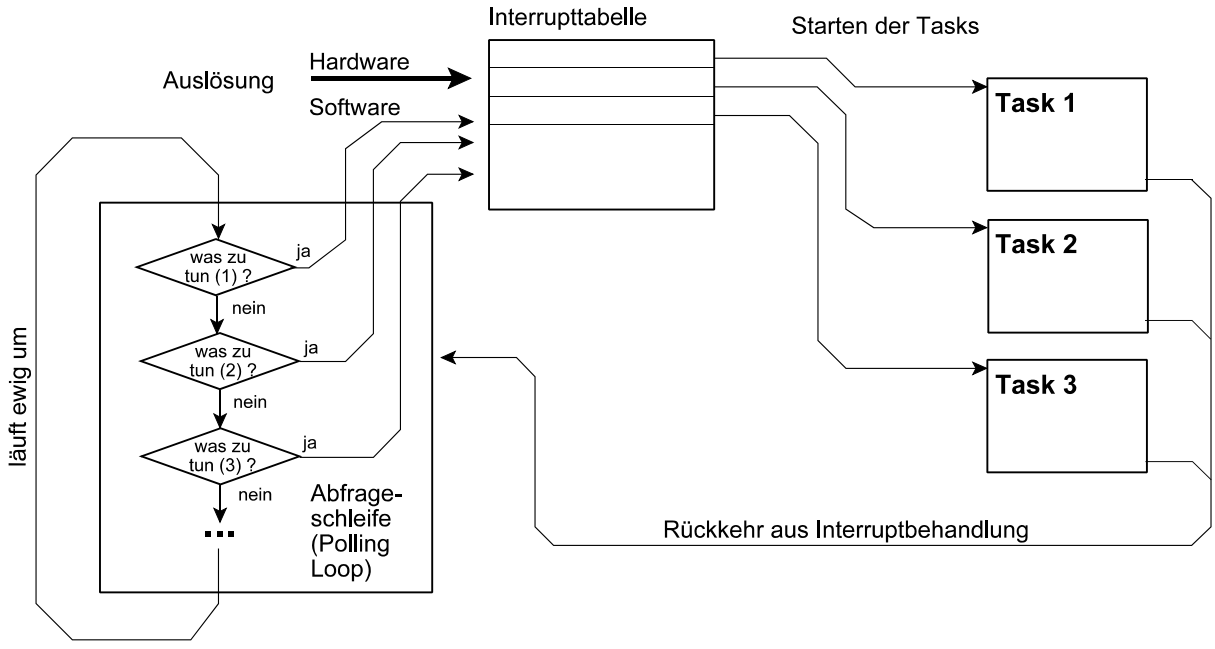


Abb. 3.10 Unterbrechungsauslösung und Abfrage kombiniert

3.4 Prinzipien der Laufzeitzuteilung

Selbstverwaltung: kooperatives Multitasking
Das gerade laufende Programm entscheidet selbst, wann es andere Programme zum Zuge kommen läßt. Beispielsweise gibt es hierfür einen entsprechenden Systemaufruf (nennen wir ihn BREAK; vgl. Abb. 3.8). Der Programmierer muß nun solche BREAK-Anweisungen gelegentlich in sein Programm einstreuen (hierfür gibt es Erfahrungswerte). Das System reagiert auf BREAK folgendermaßen:

- steht ein weiteres Programm bereit, das auch Laufzeit haben möchte, so erhält es tatsächlich Laufzeit,
- fordert kein anderes Programm Laufzeit an, wird das ursprüngliche Programm im Anschluß an BREAK fortgesetzt.

Dies ist ein seit langem bewährtes Prinzip, das in vielen Embedded Systems (und auch in Windows 3.x)[1)] angewendet wird. Es kommt ohne zusätzliche Hardware und ohne Unterbrechungssystem aus. Zudem hat es der Programmierer in der Hand, die Kontrolle über die Maschine nur dann abzugeben, wenn es seinem Programm wirklich nicht schadet.

Zwangsverwaltung: preemptives Multitasking
Die Nachteile des kooperativen Multitasking:

- vergleichsweise hohe Latenzzeiten: ein wartendes Programm erhält erst dann Laufzeit, wenn im gerade laufenden eine BREAK-Anweisung ausgeführt wird (typischerweise im Abstand von mehreren...vielen Millisekunden),
- manche Programme sind nicht wirklich kooperativ (da sie z. B. zuwenig BREAK-Anweisungen (genauer: SLEEP-Anweisungen) enthalten). Dies ist typisch für etliche Windows 3.x-Anwendungen.

Die Abhilfe: wenn es nicht freiwillig geht, muß eben Zwang dahinter. Hierfür braucht man aber gewisse Vorkehrungen in der Hardware:

- einen vom Programmablauf unabhängigen Zeitgeber,
- ein Unterbrechungssystem,
- besondere Befehle (oder Systemfunktionen) zum zeitweiligen Verhindern der Taskumschaltung.

Wir wollen hier nur den einfachsten Fall betrachten: die Zuteilung sog. Zeitscheiben (zyklisches Weiterschalten; Time Slicing). Der besagte Zeitgeber erzeugt in festen Abständen, z. B. alle 10 ms, eine Unterbrechung. Diese veranlaßt, daß die Laufzeitverwaltung des Systems (Scheduler) gerufen wird, die ihrerseits dem nächsten lauffähigen Programm Laufzeit zuteilt (Abb. 3.11). Nehmen wir an, es seien 3 Tasks A, B, C lauffähig. Dann wird die Laufzeit folgendermaßen zugewiesen: 10 ms an Task A A – 10 ms an Task B – 10 ms an Task C – für 10 ms an Task A usw. Dieses Prinzip der Laufzeitvergabe ist typisch für alle höherentwickelten Plattformen (Windows ab 95/98, Unix usw.).

Vorrangsteuerung
Das eben beschriebene zyklische Weiterschalten zwischen den einzelnen lauffähigen Tasks (Fachbegriff: Round Robin) hat den Vorteil, "fair" zu sein (jede Task bekommt den gleichen Anteil). Es führt aber nach wie vor zu beachtlichen Latenzzeiten (wurde z. B. gerade Task A mit Laufzeit versorgt, so dauert es 20 ms, bis Task C drankommt). Auch kann es gelegentlich sein, daß wir z. B. die Ergebnisse von Task B unbedingt brauchen, während es bei Task A auf einige hundert ms nicht ankommt. Hierfür hat man eine Vorrangsteuerung über Prioritäten (Priorities) eingeführt. Dies kann z. B. so aussehen, daß einer Task höherer Priorität mehrere Zeitscheiben zugewiesen werden. Beispiel: wir weisen der Task B eine höhere Priorität zu. Der Scheduler vergibt nun zyklisch 10 ms an Task A, 20 ms an Task B, 10 ms an Task C usw. Weitere Prioritätsfestlegungen können z. B. bewirken, daß bestimmte Unterbrechungen aus der Hardware heraus unmittelbar – also gleichsam außer der Reihe – die Zuteilung von Laufzeit an eine bestimmte Tasks veranlassen.

1): Die entsprechende Funktion der Windows-API heißt SLEEP.

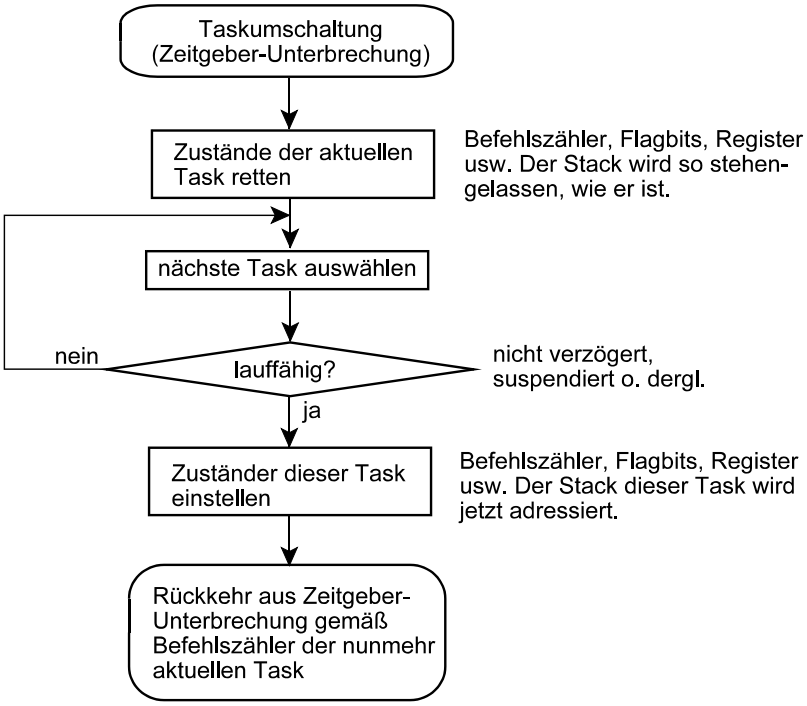


Abb. 3.11 Laufzeitzuteilung durch zyklisches Weiterschalten (Time Slicing)

Vorder- und Hintergrund
Diese Begriffe aus der EDV-Programmierung kennzeichnen eine grobe Einteilung von Programmen gemäß ihrer Priorität.

Vordergrund (Foreground)
Diese Programme haben höhere Prioritäten. Typischerweise läßt man jene Programme im Vordergrund laufen, mit denen die Nutzer (am Bildschirm) unmittelbar zusammenarbeiten.

Hintergrund (Background)
Die Programme müssen sich mit dem Rest an Laufzeit begnügen, der übrigbleibt (sie erhalten immer dann Laufzeit, wenn die Vordergrundprogramme gerade nichts zu tun haben). Ein typisches Beispiel ist eine im Hintergrund laufende Datensicherung. Wichtig: Hintergrundprogramme sollten "automatisch" durchlaufen, also keine Bedieneingriffe erfordern.

Mit Prioritäten umgehen
Entsprechende Plattformen (z. B. Unix, Windows NT und Nachfolger) ermöglichen es, Prioritäten einzustellen bzw. zu ändern. Die jeweiligen Bedienhandlungen sind in der Systemdokumentation beschrieben. Zum Experimentieren: nichts überstürzen. Man kann das Zeitverhalten eines Systems ohne viel Mühe "verschlimmbessern", ja sogar dafür sorgen, daß buchstäblich nichts mehr funktioniert. Einschlägige Versuche sind in der Regel nur auf Grundlage von Leistungsmessungen sinnvoll.

Vorder- und Hintergrund in Embedded Systems
Hintergrundprogramme laufen zyklisch um und haben im Grunde Zeit (es kommt auf ein paar Millisekunden nicht an). Beispiel: Bedienfeldabfrage. Vordergrundprogramme werden hingegen bedarfsweise gestartet (typischerweise durch Interruptauslösung). Sie müssen vergleichsweise scharfe Realzeitbedingungen einhalten.

Gegenseitige Behinderungen

Deadlocks
Wenn beispielsweise Task A darauf wartet, daß Task B eine bestimmte Datei freigibt, B aber darauf, daß A endlich eine bestimmte Ausgabe erledigt hat, so bleiben beide Tasks hängen. Ein solcher Zustand heißt Verklemmung oder Deadlock. Dies ist ein harter Fehlerzustand. Die Gefahr von Deadlocks besteht dann, wenn – wie im Beispiel angedeutet – Tasks voneinander abhängig sind.

Livelocks
Als Livelock (Blockierung) bezeichnet man Zustände der Behinderung, die nicht durch gegenseitige Abhängigkeiten hervorgerufen werden. So kann z. B. ein System "klemmen", wenn Task A eine hohe Priorität hat und Task B eine niedrige, Task A aber das System laufend belegt, so daß Task B nie zum Zuge kommt.

Hinweise:

1. Sind die Tasks vollkommen unabhängig voneinander, so kann es auch keine Deadlocks geben.
2. Eine Laufzeitvergabe nach dem Round-Robin-Prinzip (zeitgesteuerte zyklische Weiterschaltung) vermeidet Livelocks. Trotzdem kann z. B. eine ungeschickter Prioritätseinstellung zu einer merklichen (bisweilen untragbaren) Verschlechterung der Systemleitung führen (typischerweise an extrem langen Antwortzeiten erkennbar).
3. Round Robin schützt nicht grundsätzlich gegen Deadlocks.
4. Der Theorie nach ist es möglich, bestimmte Deadlock-Situationen aus der Analyse der Programm-Quelltexte zu erkennen.
5. Manche Deadlocks könne durch Ändern von Prioritäten tatsächlich abgestellt werden, manche nicht. Ein Experimentieren mit Prioritäten kann dann bestenfalls die Wahrscheinlichkeit verringern, daß eine Deadlock-Situation auftritt, die Gefahr selbst ist aber nach wie vor latent (man verschiebt das Auftreten lediglich auf einen späteren Zeitpunkt).

4. Reale und virtuelle Maschinen

Reale Maschinen
Eine reale Maschine ist gegenständlich vorhanden – es ist letzten Endes der Prozessor, dessen Hardware tatsächlich Maschinenbefehle ausführt. Dies bildet die Grundlage der meisten derzeitigen Systeme.

Virtuelle Maschinen auf Grundlage der Maschinenbefehle des Prozessors
Eine einleuchtende Idee, um das Problem zu lösen, auf ein und demselben Prozessor mehrere Programme laufen zu lassen, ja sogar mehrere Software-Plattformen. Wir stellen – zunächst gedanklich – jedem Programm den ganzen Prozessor zur Verfügung: (fast) alle Maschinenbefehle, (fast) alle Register usw. Jedes Programm sieht einen Arbeitsspeicher, der ganz vorn anfängt (mit Adresse 0) und ganz hinten aufhört (idealerweise am Ende des Speicheradreßraums). Jedes Programm hat so einen eigenen *virtuellen* (scheinbaren) Prozessor ganz für sich allein. Damit das funktioniert, brauchen wir ein übergeordnetes Steuerprogramm. Dessen grundsätzliche Wirkungsweise ist nicht einmal sonderlich kompliziert. Nehmen wir an, es seien 3 virtuelle Maschinen A, B, C zu implementieren. Dann richten wir 3 Dateien DA, DB, DC ein, die jeweils den gesamten Arbeitsspeicherinhalt sowie alle erforderlichen Angaben aus dem Prozessor (Registerinhalte, Befehlszähler usw.) aufnehmen können. Wir starten zunächst die virtuelle Machine A, indem wir ihr Speicherabbild aus der Datei DA in den Arbeitsspeicher laden und einen Programmstart gemäß Abb. 3.5 veranlassen. Nach einer gewissen Zeit halten wir den Verarbeitungsablauf an und schaffen den Arbeitsspeicherinhalt sowie den aktuellen Prozessorzustand wieder in die Datei DA. Anschließend laden wir die Datei DB in den Arbeitsspeicher, um die virtuelle Maschine B zu starten usw. In der Praxis wird dieser einfache Ablauf mit vielfältigen Tricks

angereichert, damit die Systemleistung nicht allzu sehr abfällt (es wäre doch recht langweilig, würde man z. B. stets den gesamten Arbeitsspeicherinhalt (etliche MBytes) auf Dateien auslagern). Derartige Prinzipien werden seit den 70er Jahren angewendet, und die IA-32-Architektur sieht sogar eigens eine Betriebsart vor, um die Ausführung von DOS- und anderen 16-Bit-Anwendungen zu unterstützen: den virtuellen x86-Modus (VM86).

Virtuelle Maschinen auf Grundlage fiktiver Befehlslisten
Diese Idee hat man sich vor allem deshalb einfallen lassen, um die Entwicklung von Compilern zu erleichtern. Jede höhere Programmiersprache lebt schließlich auch von dem Versprechen, daß man ein Programm nur einmal schreiben muß und sich dann für verschiedene Plattformen übersetzen lassen kann. Jede Plattform erfordert aber eine Compiler. Um den Entwicklungsaufwand zu verringern, sind die Compiler-Autoren auf folgendes Prinzip gekommen:

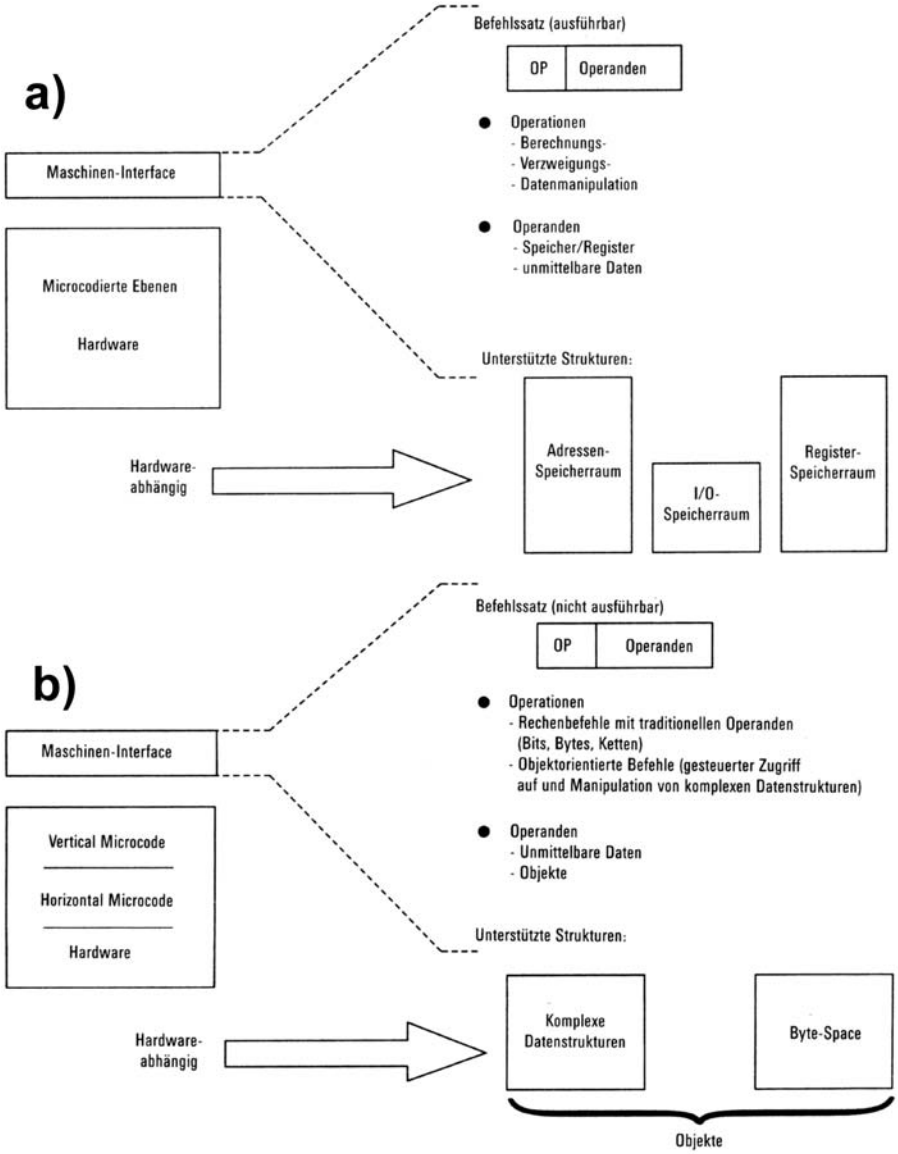
- der Compiler erzeugt das Programm zunächst in einer Zwischensprache. Das ist typischerweise die Assemblersprache eines fiktiven Prozessors, also eines Prozessors, den es als Hardware gar nicht gibt.
- wollen wir das Programm auf einem System X laufen lassen – und kommt es nicht auf wirklich höchste Geschwindigkeit an – , so schreiben wir einen Interpreter für diese Zwischensprache. Das ist ein Programm, das Befehl für Befehl (des fiktiven Prozessors) holt und deren Wirkung mit den Befehlen des Systems X (des Zielsystems) nachbildet – ein zwar nicht triviale, aber durchaus lösbare Aufgabe.
- kommt es hingegen auf Leistung an, so schreiben wir einen weiteren Compiler, der das Programm des fiktiven Prozessors in eines des Zielsystems umwandelt.

Beispiele solcher Zwischensprachen bzw. virtueller Maschinen: der sog. P-Code (für die Programmiersprache Pascal), (2) die Java Virtual Machine (JVM).

Sprachumgebungen als virtuelle Maschinen
Das ist der nächste Schritt – weshalb muß das Anwendungsprogramm überhaupt die doch recht primitiven Maschinenbefehle - wie ADD, MOVE usw. – zu sehen bekommen? Statt dessen bieten wir von vornherein eine hochentwickelte Programmschnittstelle an, die sowohl elementare Operationen als auch den Umgang mit komplexen Datenstrukturen unterstützt. Dieses Prinzip wurde z. B. im System AS/400 verwirklicht (Abb. 4.1).

Die Vorteile: (1) deutlich höhere Sicherheit – kein Anwendungsprogramm ist in der Lage, in der Hardware irgend etwas zu verstellen, (2) Hardware-Unabhängigkeit. So hatten die ersten AS/400-Modelle einen speziellen Prozessor, während die neueren Typen mit PowerPC-Prozessoren bestückt sind (der Anwender merkt eine solche Umstellung gar nicht).

Der Nachteil: der hohe Entwicklungsaufwand. Die Sache wird nur dann etwas, wenn alles aus einer Hand kommt – und auf die typischen Anwendungserfordernisse abgestimmt ist. Das Problem bei allen derartigen Lösungen mit hohem Vorfertigungsgrad: wenn die fertigen Schnittstellen saugend zum Problem passen, dann ist die Leistung der von herkömmlichen Systemen sogar überlegen (denn die Schnittstellen-Entwickler können intern unbedenklich alle möglichen Tricks anwenden und so die Hardware voll ausnutzen). Paßt es aber nicht, so ist der Anwender gezwungen, seine Funktionen aus den elementaren Operationen der Schnittstelle zusammenzustückeln – und dann wird es langsam (weil zwischen Schnittstellen-Befehl und Hardware die diversen Schichten des internen Mikrocodes liegen).



a) herkömmliche Schnittstelle. Die Befehle, die der Anwender aufruft, wirken direkt auf die Hardware. Der Anwender sieht verschiedene Adreßräume (Speicher, E-A, Register).

b) die AS/400-Schnittstelle. Die Befehle, die der Anwender aufruft, werden von internen Programmen (hier Mikrocode genannt) interpretiert. Der Anwender sieht nur Objekte (das können Bytes sein, aber auch komplexe Datenstrukturen (z. B. eines Datenbanksystems)).

Abb. 4.1 Programmschnittstellen von Prozessoren (Maschinen-Interfaces) im Vergleich (IBM)

Tasks und virtuelle Maschinen

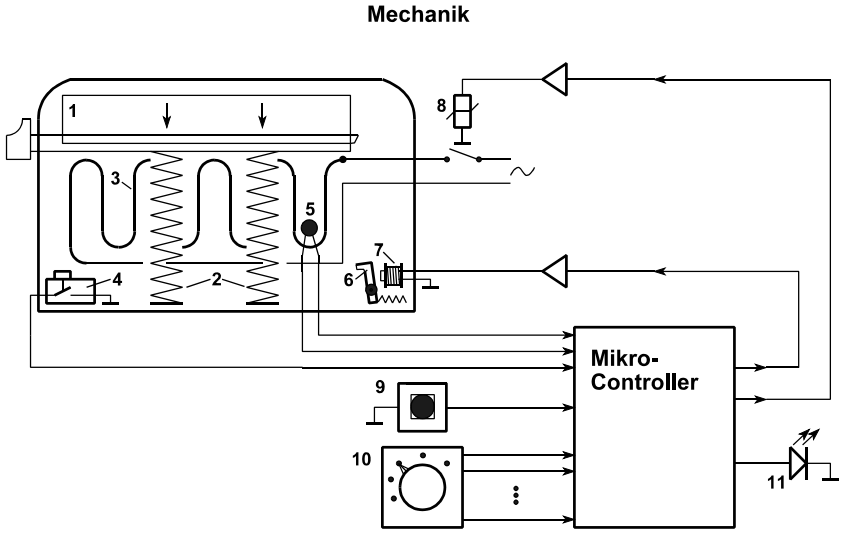
Beides kann man als Vorkehrungen ansehen, um mehrere Anwendungsprogramme zeitmultiplex ausführen zu können. Die Übergänge sind fließend (Tabelle 4.1).

Multitasking	virtuelle Maschinen
• mehrere Anwendungen können zeitmultiplex ausgeführt werden, • jede Anwendung hat fast die gesamte Maschine zur Verfügung – aber eben nicht alles (nur bestimmte Speicherbereiche, E-A-Adressen, Register usw.), • Plattform organisiert das Umschalten zwischen den Tasks, • Tasks müssen sich an Konventionen der Plattform halten, • nur ein Betriebssystem für alle Tasks, • Umschaltung zwischen Tasks vergleichsweise schnell. *Eine "feinfühlige", gezielt dosierbare Ressourcenverwaltung, die aber den einzelnen Anwendungen Beschränkungen auferlegt.*	• mehrere Anwendungen können zeitmultiplex ausgeführt werden, • jede Anwendung hat (scheinbar(virtuell)) die gesamte Maschine zur Verfügung, • Plattform organisiert das Umschalten zwischen den virtuellen Maschinen, • in jeder virtuellen Maschine kann ein eigenes Betriebssystem laufen, • Umschaltung zwischen virtuellen Maschinen ggf. zeitaufwendig (hier kommt es auf die Einzelheiten der Auslegung an). *Eine pauschale Ressourcenverwaltung, die sich nicht mit Kleinigkeiten abgibt – dafür aber ihre Zeit braucht.*

Tabelle 4.1 Tasks und virtuelle Maschinen im Vergleich

5. Ein ganz elementares Programmbeispiel

Beginnen wir mit einer ganz einfachen Anwendung. Es handelt sich darum, einen Toaster mit einem Mikrocontroller zu steuern (Abb. 5.1). Hier kommt man offensichtlich noch ohne Multitasking aus; der gesamte Programmablauf läßt sich intuitiv als Flußdiagramm angeben (Abb. 5.2).



1 - Korb; 2 - Druckfeder; 3 - Heizwendel; 4 - Endlagenkontakt; 5 - Temperatursensor; 6 - Rastung; 7 - Auslösemagnet; 8 - Schaltrelais (oder Triac); 9 - Stoptaste (zum Abbrechen des Toast-Vorgangs); 10 - Drehschalter zum Einstellen des Bräunungsgrades; 11 - Kontrollanzeige (Leuchtdiode).

Abb. 5.1 Ein Toaster

Was ist zu steuern?
Wir legen die Brotscheiben ein und drücken den Korb 1 nach unten. Er rastet in dieser Lage ein (Rastung 6). Dieser Betriebszustand wird mittels des Endlagenkontaktes 4 signalisiert. Dies bewirkt,

daß der Toast-Vorgang beginnt. Um ihn zu beenden, wird der Auslösemagnet 7 erregt und somit Rastung 6 ausgelöst. Daraufhin drückt die Druckfeder 2 den Korb 1 wieder nach oben. Zur Beeinflussung des Ablaufs sind eine Stoptaste 9 (vorzeitiges Beenden) und ein Drehschalter 10 (zum Einstellen des Bräunungsgrades) vorgesehen. Das Toasten selbst beruht auf einer Erregung der Heizwendel 3. Hierzu muß das Schaltrelais 8 erregt werden.

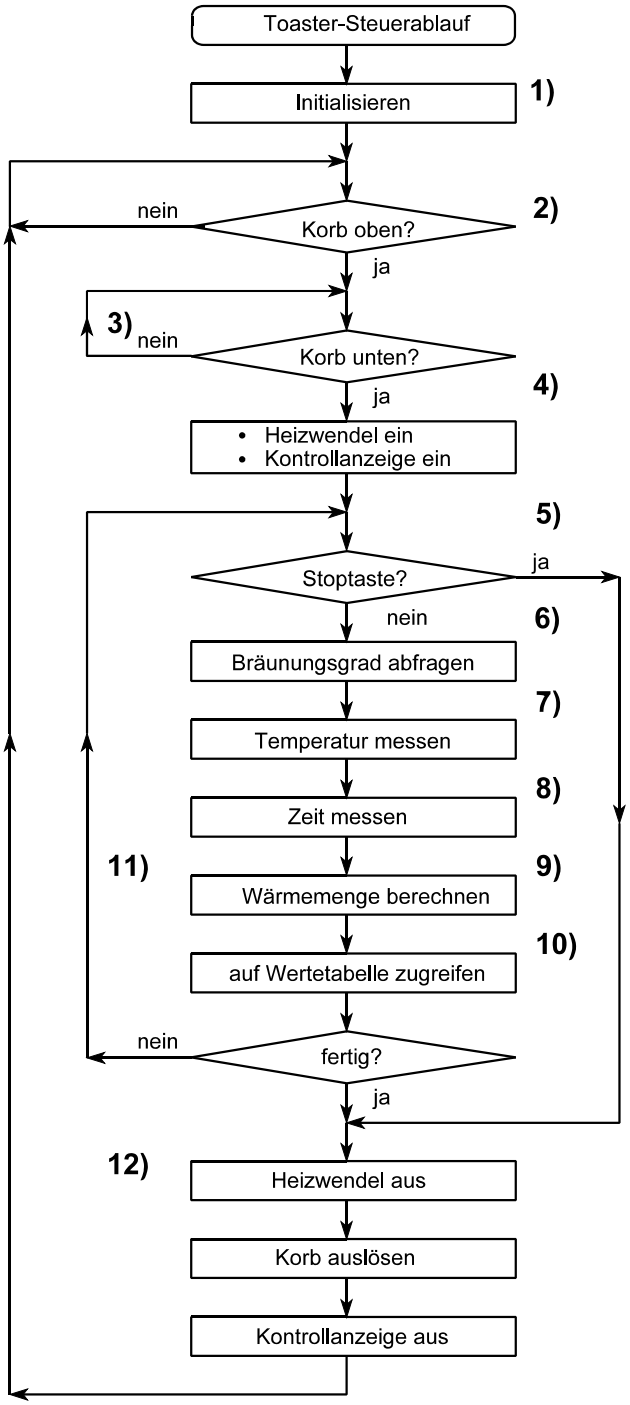


Abb. 5.2 Ein naheliegender Programmablauf

1) nach dem Einschalten wird alles in "Grundstellung" versetzt (Fachbegriff: Initialisierung). Ggf. wird auch der Korb 1 ausgelöst.
2) der Korb 1 darf nicht (in der unteren Lage) eingerastet sein. Ggf. warten, bis der Endlagenkontakt 4 abgeschaltet hat.
3) Warteschleife im Ruhezustand,
4) mit dem Einrasten des Korbes 1 (Meldung über Endlagenkontakt 4) beginnt der Toast-Vorgang,
5) die Stoptaste 9 wird immer wieder abgefragt, um zu erkennen, ob der Vorgang abgebrochen werden soll,
6) der Drehschalter 10 wird immer wieder abgefragt, um den gewünschten Bräunungsgrad auch ggf. mitten im Ablauf ändern zu können (Bedienkomfort),
7) mittels Temperatursensor 5,
8) durch interne Zeitzählung im Mikrocontroller,
9) die bisher umgesetzte Wärmemenge wird aus Temperatur und Zeit errechnet (und in jedem Schleifenumlauf aufsummiert). Es handelt sich hier um eine interne Hilfsgröße, die nicht normgerecht (als SI-Einheit J (Joule)) ermittelt werden muß.
10) dies ist ein einfaches, oft angewendetes Prinzip: man rechnet nicht mit komplizierten Formeln, sondern man hat im Mikrocontroller eine Wertetabelle fest gespeichert, die zu jedem einstellbaren Bräunungsgrad die zugehörige Wärmemenge angibt (die Werte wurden während der Entwicklung durch Versuch bestimmt),
11) die Schleife des Toast-Vorgangs,
12) zurück zur Warteschleife – hat der Korb die untere Endlage verlassen, kann ein neuer Toast-Vorgang gestartet werden.

Das Organisationsprinzip eines typischen Anwendungsprogramms
Eine solche Anwendung braucht gar keine Systemsoftware (Stand-Alone-Programm). Unser Beispiel zeigt das Organisationsprinzip eines typischen Anwendungsprogramms:

- es wird gestartet (irgendwie muß es schließlich einmal losgehen (hier: mit dem Einschalten)),
- es richtet sich erst einmal ein (Initialisierung),
- es fragt ab, ob etwas zu tun ist,
- ist etwas zu tun, so wird dies ausgeführt,
- ist die Arbeit erledigt, wird wiederum abgefragt, ob etwas zu tun ist (Abfrage- bzw. Warteschleife).

6. Ereignissteuerung

Betrachten wir nochmals den Toaster. Eingabeeinrichtungen sind: der Endlagenkontakt 4, der Temperatursensor 5, die Stoptaste 9 sowie der Drehschalter 10. Alle diese Einrichtungen werden vom Programm abgefragt. Der Gedanke ist an sich einfach: wir schreiben ein Programm, das irgendwie anfängt und das die einzelnen Einrichtungen abfragt, um in Erfahrung zu bringen, was es als nächstes tun soll (Abb. 6.1). Die Nachteile des Abfrageprinzips werden deutlich, wenn wir uns ein großes, kompliziertes Programm vorstellen:

- es ist viel abzufragen (Tasten, Meldungen von Geräten usw.),
- um auf irgendeine Eingabe überhaupt reagieren zu können, müssen wir im Programm auch daran vorbeikommen – ein Programm, das z. B. gerade mit einem Regelalgorithmus beschäftigt ist, kann währenddessen keine Tasten abfragen,
- je nach Funktionszustand ist auf gleiche Eingaben (z. B. Tastenbetätigungen) jeweils anders zu reagieren,
- in den Warteschleifen wird nicht mehr getan als Laufzeit zu verbrauchen.

Der Ausweg: eine grundsätzlich andere Gestaltung der Programme. Dabei geht man nicht mehr vom "endlos laufenden" Programm aus, sondern von den Ereignissen (Events), die außen ausgelöst werden. Ein Ereignis ist z. B. eine Tastenbetätigung, eine Mausbewegung oder die Meldung eines Sensors. Der Programmierer schreibt dann kein großes Programm mehr, in dem alles "an einem Faden hängt". Vielmehr löst er das Gesamtproblem in entsprechend viele sog. Ereignisbehandler (Event Handlers) auf. So liegt es nahe, in unserem Toaster-Beispiel folgende Ereignisse anzusetzen:

1. Gerät ist eingeschaltet worden,
2. Endlagenkontakt schließt (Korb unten),
3. Stoptaste betätigt.

Der Programmablauf des Toasters zerfällt somit in drei Ereignisbehandler (Abb. 6.2).

.

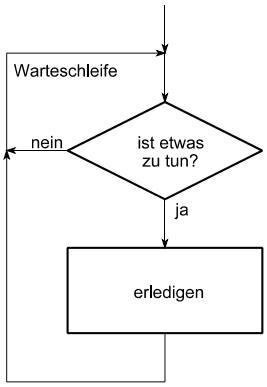


Abb. 6.1 Prinzip der Abfragesteuerung

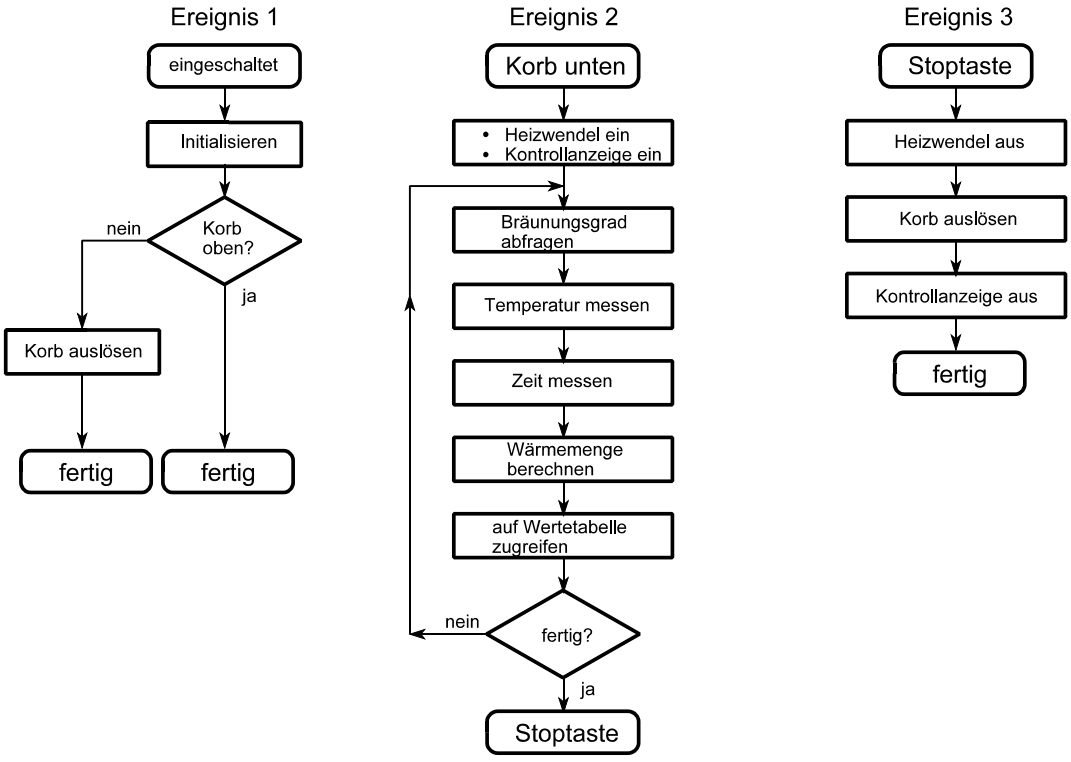


Abb. 6.2 Die Ereignisbehandler zur Steuerung des Toasters

Hinweise:

1. Der Programmierer schreibt einfach die drei unabhängigen Programmstücke und überläßt es dem System, das jeweils passende aufzurufen.
2. "Fertig" bedeutet: Ereignis ist behandelt; zurück zur Plattform.
3. Achten Sie auf das Ende des 2. Ereignisses: die nachfolgenden Funktionen entsprechen jenen, die auch beim Betätigen der Stoptaste auszuführen sind. Der Programmierer macht es sich daher einfach und ruft den Behandler des Stoptasten-Ereignisses direkt auf.

Der 3. Hinweis deutet einen Vorteil dieser Programmierweise an: dem Behandler ist es gleichgültig, von wem er aufgerufen wird. Es genügt also, für jede Funktion ein einziges Programm zu schreiben und dieses von allen in Frage kommenden Ereignisbehandlern aufrufen zu lassen.

Wie werden die Ereignisse wirksam?
Ein technisches Problem. Wir brauchen irgend etwas, das z. B. auf Grund einer Tastenbetätigung das Starten eines bestimmten Programms veranlaßt. Im allgemeinen Sinne erfordert das eine Unterbrechungsauslösung, die mit der Übergabe der erforderlichen Parameter gekoppelt ist. Hierzu sind entsprechende periphere Schaltmittel vorzusehen. Typische Lösungen:

- hardwareseitige Unterbrechungsauslösung, die erforderlichenfalls von Programmabläufen im Prozessor unterstützt wird (vgl. Abb. 3.10),
- E-A-Steuerhardware (Controller), die Interrupts auslösen und Parameter übergeben kann. Beispielsweise sind an der Auslösung von Tastaturereignissen in herkömmlichen PCs zwei Mikrocontroller beteiligt (davon einer in der Tastataur), in neumodischen PCs der Mikrocntroller in der Tastatur und der USB-Hostadapter auf dem Motherboard.
- vorgeordnete programmierbare Einrichtungen (programmierbare Interruptcontroller). Es gibt Mikrocontroller, die eigens für solche Anwendungen entwickeltt wurden. Komplexere Systeme enthalten typischerweise mehrere Mikrocontroller oder Prozessoren, die die Schnittstellensteuerung und Interfaceanpassung übernehmen (als extremes Beispiel vgl. Abb. 2.3).

Zudem muß eine Software-Plattform vorhanden sein, die weiß, was der Prozessor tun soll, wenn ein Ereignisbehandler "fertig" geworden ist. Infolgedessen lohnt sich eine voll ausgebaute Ereignissteuerung erst von einer gewissen Größe des Systems an (einen Toaster, einen Staubsauger usw. wird man in der Praxis wohl kaum so steuern, durchaus aber z. B. den Motor im Auto).

Ereignissteuerung in einem Mikroprozessorsystem
In diesem System ist das in Abb. 6.2 veranschaulichte Prinzip – die Auflösung der Anwendungsprogramme in einzelne Ereignisbehandler – von Grund auf verwirklicht worden. Es gibt physische und logische Ereignisse. Physische Ereignisse sind Bedingungen, die in der Hardware auftreten, z. B. Tastenbetätigungen, Meldungen von Sensoren usw. Das Betriebssystem arbeitet nur mit logischen Ereignissen. Ein logisches Ereignis ist durch einen Ereignissteuerblock (Event Control Block ECB) gekennzeichnet. Physische Ereignisse sind in logische umzusetzen. Hierzu sind entsprechende Ereignisannahmeroutinen erforderlich (Abb. 6.3). Abb. 6.4 veranschaulicht – ergänzt durch die Tabellen 6.1 bis 6.3 – die Struktur eines Ereignissteuerblocks ECB. Der Ereignissteuerblock gibt an, in welcher Task das Ereignis behandelt wird und welches Programm dafür zu starten ist.

Zunächst seien einige typische Spitzfindigkeiten der Ereignissteuerung angeführt:

1. Es kann vorkommen, daß sich die Ereignisse gleichsam überschlagen: ein Ereignis wurde ausgelöst und dessen Behandlung läuft. Währenddessen tritt aber die auslösende Bedingung schon wieder auf. Hierbei darf nichts durcheinanderkommen (es darf u. a. nicht sein, daß die Auslösung Parameter verändert, mit denen die Behandlung gerade arbeitet).

2. Was man sich praktisch nicht leisten kann: die Ereignisannahme solange zu verzögern, bis die laufende Ereignisbehandlung erledigt ist – derartige Systeme reagieren ausgesprochen zähe und werden von ihren Nutzern kategorisch abgelehnt. Wichtig: Elastizität vorsehen. Soll heißen: Parallelarbeit und Ablaufüberschneidungen zulassen und zur Aufrechterhaltung der Ordnung immer wieder zwischenpuffern (FIFOs in der Hardware, Warteschlangen in der Software). Beispiel: an der Tastenbehandlung im PC – die aus Sicht der Realzeitanforderungen vergleichsweise harmlos ist – sind wenigstens drei Puffer beteiligt: in der Tastatur, im Tastaturcontroller des Motherboards und im System (z. B. die Nachrichtenwarteschlange (Message Queue) von Windows). Der Aufwand für die Parametertransporte und das Verwalten der Puffer ist auf den ersten Blick womöglich abschreckend – aber was sein muß, muß sein ... (Die Erfahrung zeigt, daß solche Maßnahmen das Realzeitverhalten des Systems merklich verbessern – obwohl die Abläufe kompliziert sind und ihrerseits Laufzeit erfordern.)
3. Es können Ereignisse auftreten, die vorrangig behandelt werden müssen. Hierbei genügen oftmals zwei Prioritätsstufen: (1) Erzwingen der ersten Priorität, wobei die weitere Ereignisreihenfolge, die Taskzustände usw. erhalten bleiben (das Ereignis drängt sich gleichsam nur vor); (2) Erzwingen des Programmstarts ohne Rücksicht auf aktuelle Zustände (das Ereignis setzt sich um jeden Preis durch, z. B. um Notmaßnahmen einzuleiten oder Fehlerbedingungen zu behandeln). Zweckmäßig ist, wenn dabei die Pufferbelegungen, Zustandsangaben usw. zunächst erhalten bleiben, so daß der Ereignisbehandler die Situation eingehend analysieren kann.
4. Ereignisannahme und Ereignisbehandlung sollten flexibel umsteuerbar sein, z. B. in Abhängigkeit von Bedienhandlungen, Bildschirmdarstellungen, Maschinenzuständen usw.
5. Weitere oftmals nützliche Funktionen der Ereignisbehandlung (Auswahl):
   - das Ereignis bewirkt keinen Programmstart, sondern das Herausführen des Programms aus einem Wartezustand,
   - das Ereignis wird angenommen, aber nur registriert, so daß ein Programm dessen Auftreten abfragen kann,
   - Ereignisse lassen sich abweisen, ohne den Programmablauf an sich zu beeinträchtigen,
   - Überlaufbedingungen (z. B. von Warteschlangen) werden erkannt und können programmseitig behandelt werden,
   - die Auslösung bestimmter Ereignisse kann einer Zeitkontrolle unterzogen werden (Watchdog-Prinzip).

Es folgt ein kurzer Überblick über wesentliche Einzelheiten. (Die Funktionen, Abläufe, Informationsstrukturen usw. kommen in nahezu jedem einigermaßen komplexen Realzeitsystem vor. In der Implementierung – und auch in den Begriffsbildungen – gibt es allerding große Unterschiede.)

Ereignisauslösung
Ereignise werden von anderen Prozessoren, speziellen E-A-Einrichtungen oder programmseitig. Ein Ereignis kann erst dann ausgelöst werden, wenn der zugehörige ECB frei ist (BUSY = 0). Die Auslösung besteht darin, den ECB als besetzt zu kennzeichnen (BUSY = 1), die Parameter einzutragen und einen Interrupt auszulösen bzw. zum Ereignisbehandler des Systems zu verzweigen (Abb. 6.5).

Ereignisannahme
Die Parameter werden in den Parameterbereich der jeweiligen Task transportiert (Abb. 6.6). Dann wird der ECB wieder als frei gekennzeichnet (BUSY = 0). Innerhalb der Task kommt das Ereignis zur Wirkung. Was im einzelnen abläuft, hängt vom Ereignistyp ab (vgl. Tabelle 6.2).

Der Tasksteuerblock (Task Control Block TCB)
Der TCB enthät die Angaben der jeweiligen Task-Umgebung (Prozessorregister, Stack, globale Variable usw.; vgl. Abb. 3.4). Aus Abb. 6.6 ist ersichtlich, daß hier auch die Parameter der zu behandelnden Ereignisse untergebracht sind. Ereignisse, die ausgelöst wurden, aber auf ihre Behandlung warten, werden in eine Auftragswarteschlange eingetragen.

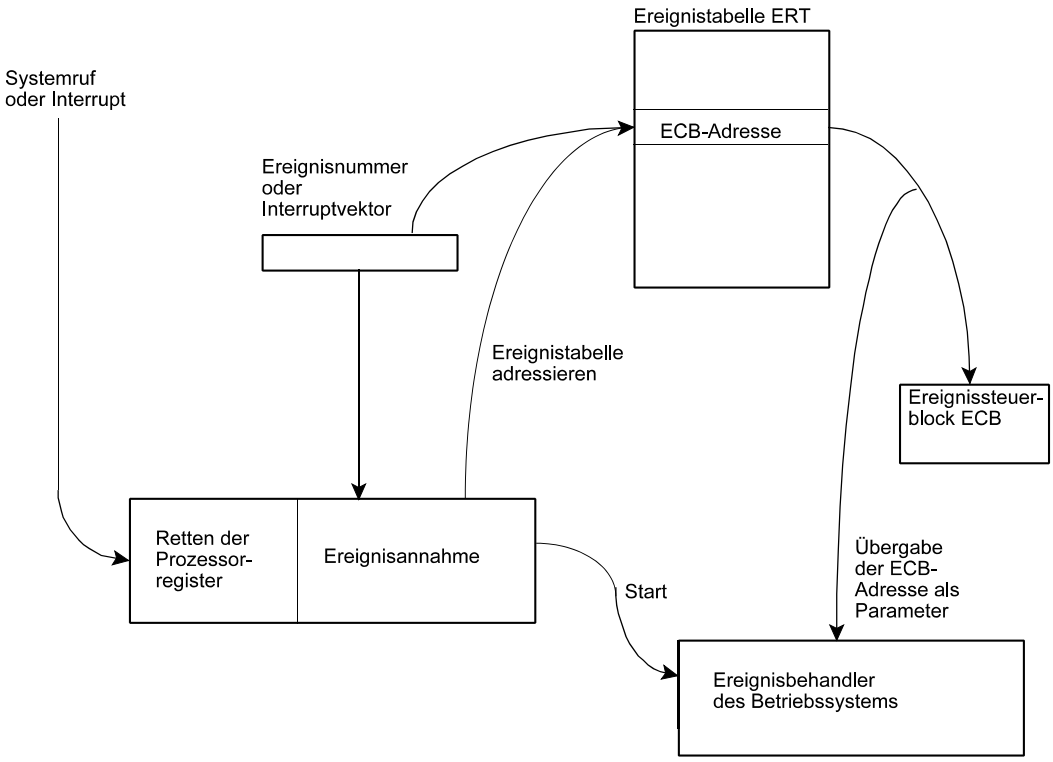


Abb. 6.3 Vom physischen zum logischen Ereignis. Die Adresse des zugehörigen Ereignissteuerblocks ECB steht in einer Ereignistabelle (Event Reference Table ERT)

Steuerbits	Typ
Task	
Programm	
Initiator-Code	
Maske	
WCB Pointer	
Parameter	

Abb. 6.4 Die grundsätzliche Struktur eines Ereignissteuerblocks ECB

Warteschlangen
Es gibt insgesamt drei Typen:

- Auftragswarteschlange. Eine je Task. Nimmt die Ereignisse auf (Parameter, Programmstartangaben), die ausgelöst wurden und auf Behandlung warten.
- Laufzeitwarteschlange. Eine je Prozessor. Enthält die arbeitenden Tasks, die aufZuteilung von Laufzeit warten.
- Ereigniswarteschlange. Eine je Prozessor. Nimmt die Ereignisse auf (Parameter, Programmstartangaben), die im Systemzustand ausgelöst wurden (die Betriebssystemabläufe sind ihrerseits unterbrechbar – eine Maßnahme, die sich – trotz der zusätzlichen Komplexität – sehr

vorteilhaft auf das Realzeitverhalten auswirkt).

Eintrag	Erläuterung
Typ	Näheres in Tabelle 6.2
Steuerbits	Näheres in Tabelle 6.3
Task	kennzeichnet die Task, in der das Ereignis behandelt werden soll
Programm	kennzeichnet das behandelnde Programm (Startadresse)
Initiator-Code	kennzeichnet, von welcher Einrichtung das Ereignis ausgelöst wurde
Maske	dient zu Steuerzwecken. Wirkung hängt vom Ereignistyp ab
WCB-Pointer	Zeiger auf einen Steuerblock zur Zeitüberwachung (WCB = Watchdog Control Block)
Parameter	hier werden die zur Ereignisauslösung gehörenden Parameter übergeben

Tabelle 6.1 Der Inhalt des ECB

Ereignistyp	Wirkung	Besonderheiten
- (Empty)	ECB leer; gar keine Wirkung	ggf. wird eine Maschinenfehler-behandlung ausgelöst
NOP	disjunktive Verknüpfung der Maske mit dem OCCURRENCE-Vektor der Task (nur Anzeige der Auslösung)	Programmangabe wird ignoriert. Parameterübergabe entspricht Typ I
I (Initiate)	Starten des angegebenen Programms in der angegebenen Task. Befindet sich die Task nicht im Ruhezustand (Idle; vgl. Abb. 3.3), so werden die Parameter und Programmstartangaben in die Auftragswarteschlange der Task eingetragen (am Ende)	vor dem Programmstart bzw. dem Eintragen in die Warteschlange wird die Maske mit dem INHIBIT-Vektor der Task konjunktiv verknüpft. Ist das Ergebnis ungleich Null (INHIBIT-Bedingung), so wird das Ereignis als erledigt deklariert, aber nicht weiter behandelt
IP (Initiate Priorized)	Starten des angegebenen Programms in der angegebenen Task. Befindet sich die Task nicht im Ruhezustand, so werden die Parameter und Programmstartangaben in die Auftragswarteschlange der Task eingetragen (am Anfang, wobei die anderen Einträge anch hinten geschoben werden)	
C (Cancel)	Starten des angegebenen Programms in der angegebenen Task. Der Programmstart wird ohne Rücksicht auf den Taskzustand erzwungen	
P (Proceed)	konjunktive Verknüpfung der Maske mit dem WAIT-Vektor der Task. Befindet sich die Task im Wartezsustand, so wird dieser verlassen, wenn das Verknüpfungsergebnis gleich Null ist, andernfalls beibehalten	Programmangabe wird ignoriert

Tabelle 6.2 Ereignistypen

Steuerbit	Bedeutung/Wirkung	
	Bit = 0	Bit = 1
BUSY	ECB ist frei und kann Parameter aufnehmen	ECB ist besetzt
SERIALIZE	nach der Ereignisauslösung und dem Übertragen der Parameter zur jeweiligen Task wird BUSY automatisch ausgeschaltet (ECB wird wieder frei)	BUSY bleibt aktiv. Um den ECB wieder freizumachen, ist eine FREE-Anweisung zu geben
TIME SLICE	zyklisches Weiterschalten (vgl. Abb. 3.11) aus	zyklisches Weiterschalten ein
LOCK	Systemzustand wird am Ende der Ereignisbehandlung verlassen	Systemzustand wird beibehalten
TRACE	keine Aufzeichnung	die ECB-Belegung zum Zeitpunkt der Aulösung wird im TRACE BUFFER gespeichert (Debugging-Funktion)
INHIBIT SOUND	keine Wirkung	ist die INHIBIT-Bedingung wirksam geworden (Abweisung der Ereignisauslösung), so wird ein akustisches Signal (Piepston) abgegeben
SUSPEND INHIBIT	Task kann in den Zustand Suspended (vgl. Abb. 3.3) überführt werden	Task kann nicht in den Zustand Suspended überführt werden (SUSPEND-Anweisung wirkungslos)
OVERFLOW HANDLING	trifft ein Ereignis vom Typ I oder IP auf eine volle Auftragswarteschlange, so wird eine Maschinefehlerbehandlung ausgelöst	trifft ein Ereignis vom Typ I oder IP auf eine volle Auftragswarteschlange, so wird das Ereignis ignoriert. BUSY wird gelöscht, und eine Überlaufanzeige wird gesetzt
WDG CTL	diverse Steuerbits für die Zeitkontrolle der Ereignisauslösung	

Tabelle 6.3 Steuerbits

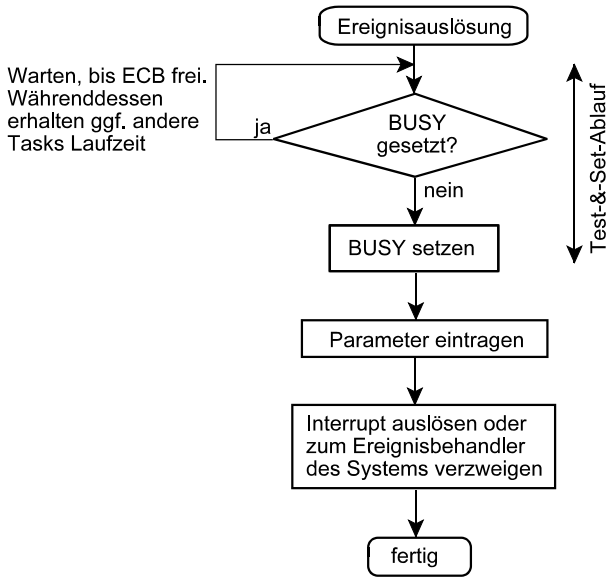


Abb. 6.5 Ereignisauslösung

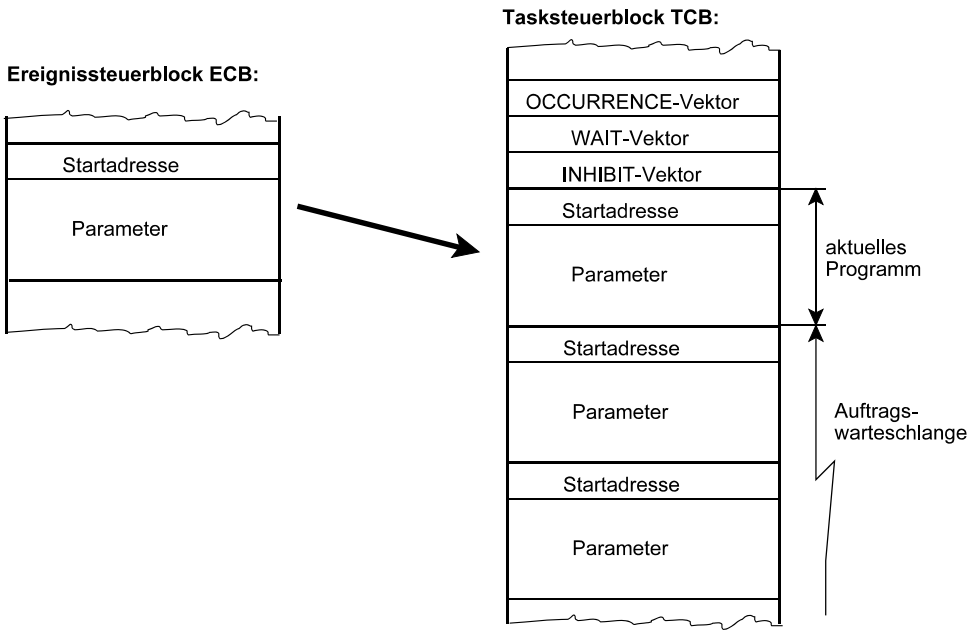


Abb. 6.6 Von der Ereignisannahme zur Behandlung in der jeweils angegebenen Task

Prioritäten
Die Erfahrung hat gezeigt, daß eine Art Schwarz-Weiß-Schema in vielen Fällen ausreicht. Demgemäß gibt es nur zwei Prioritäten:

1. Einordnung am Ende der Warteschlange (“ erst einmal hinten anstellen“). Behandlungreihenfolge entspricht reihenfolge der Auslösung.
2. sofortige Behandlung.

Taskpriorität
Wird ein Ereignis ausgelöst, und kann (oder muß) es sofort behandelt werden (Task in Ruhe oder im Wartezustand oder Ereignis vom Typ C), so erhält die betreffende Task sofort Laufzeit. Ansonsten werden die Parameter übergeben, die Ausführungsreihenfolge wird aber nicht beeinflußt.

Priorität der Ereignisbehandlung innerhalb der Task
Die Priorität hängt vom Ereignistyp ab (vgl. Tabelle 6.2). Die nomale Ereignisbehandlung (Ereignisse vom Typ I) entspricht der Reihenfolge der Ereignisauslösung. Solche Ereignissse werden ggf. (wenn sie auf eine arbeitende Task treffen) in die Auftragswarteschlange eingetragen.

Kein Programmstart, sondern programmseitige Abfrage
NOP-Ereignisse setzen Bits im OCCURRENCE-Vektor des Tasksteuerblocks (typischerweise setzt ein Ereignis ein einziges Bit, um anzuzeigen, daß es aufgetreten ist). Diese Bits können vom laufenden Programm abgefragt werden.

Ereignisabweisung
Ereignisse vom Typ I können abgewiesen werden (Beispiel: Bedienereingriffe oder Sensormeldungen, die in einem bestimmten Betriebszustand bedeutungslos sind). Die Abweisung wird über den INHIBIT-Vektor im TCB und die Maske im ECB gesteuert.

Wartezustände
Programme können auf Ereignisse warten. Hierbei wirken Ereignisse vom Typ P und WAIT-Anweisungen zusammen. Die WAIT-Anweisung lädt einen WAIT-Vektor in den TCB und versetzt die Task in den Wartezustand (eine wartende Task wird bei der Laufzeitvergabe nicht mehr berücksichtigt). Ereignisse vom Ty P können vor oder nach einer WAIT-Anweisung eintreffen. Ist es vorher eingetroffen, wird dessen Behandlung bis zum Ausführung der WAIT-Anweisung verschoben. Das Ereignis löscht die in der Maske angegebenen Bitpositionen des WAIT-Vektors. Die Programmausführung in der Task wird dann fortgesetzt, nachdem alle Bits im WAIT-Vektor gelöscht wurden. Durch entsprechendes Setzen von Bits in den Masken der ECBs und im WAIT-Vektor können vielfältige Ereigniskombinationen erfaßt werden (Abb. 6.7).

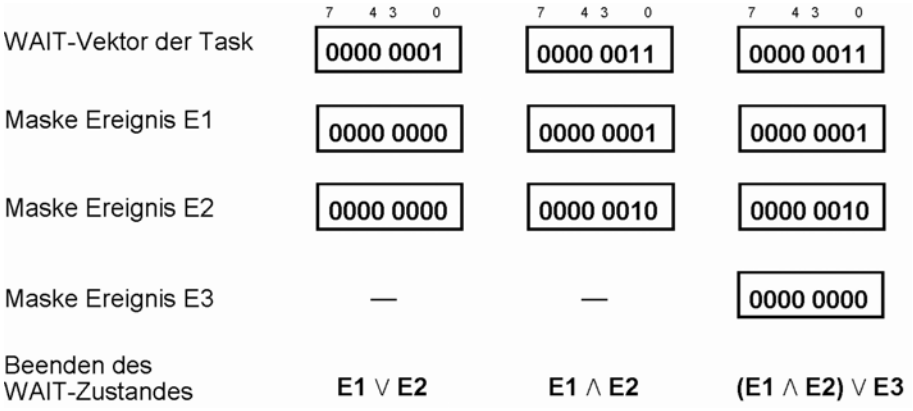


Abb. 6.7 Warten auf Ereigniskombinationen (drei Beispiele)

Ereignissteuerung unter Windows
Das vorstehend beschriebene Prinzip ermöglicht es, Systeme mit geringen Latenzzeiten zu implementieren, die flexibel steuerbar sind. In der Anwendungsprogrammierung tritt aber das Problem auf, die ECBs so einzurichten, daß sie stets auf die richtigen Behandlungsprogramme verweisen. Schwierig wird es dann, wenn sich diese Zuordnung immer wieder ändert. Beispiel: die Tastenbehandlung in einem fensterorientierten Menüsystem. Jedes Fenster ist einer Task zugeordnet, und in jeder Task haben die Tasten andere Funktionen. Jedesmal, wenn ein Fenster zur Bedienung ausgewählt wird, sind also die ECBs entsprechend umzuladen.

Windows umgeht diese Schwierigkeiten. Der Anwendungsprogrammierer, kann – dem Prinzip nach – so programmieren, als ob er eine herkömmliche C-Anwendung und die zugehörigen Bedienelemente (Tastatur, Maus usw.) für sich allein hätte (Abfrageschleife und Verzweigen zu den einzelnen Behandlungsroutinen). Ereignisse werden über Nachrichten bzw. Meldungen (Messages) signalisiert. Für jede laufende Anwendung verwaltet Windows eine Nachrichtenwarteschlange. Die Anwendung muß diese Warteschlange zyklisch abfragen. Diese Ereignisabfrage ist in die übliche C-Programmierung eingebunden – im Grunde handelt es sich um eine herkömmliche Abfrageschleife, die allerdings nach einem starren Schema aufzurufen ist (Abb. 6.8).

Aufgrund dieses Prinzips ergeben sich aber viele Systemaufrufe, also viel Overhead und damit lausige Latenzzeiten (es geht immer zwischen Anwendung und System hin und her...).

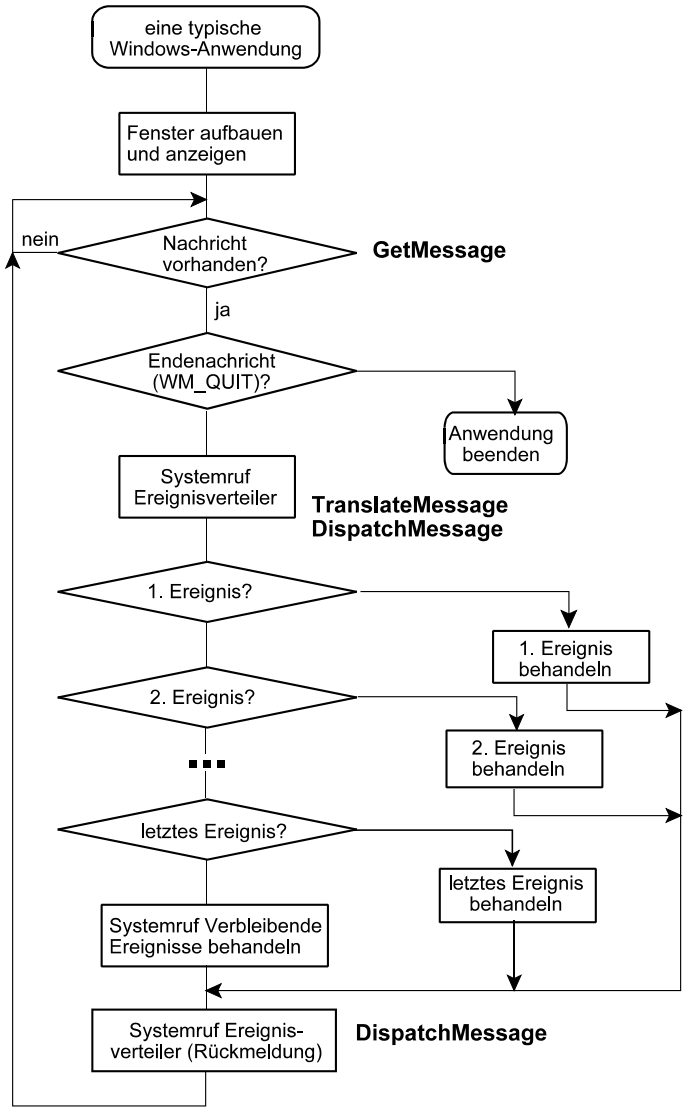


Abb. 6.8 Prinzip der Ereignisbehandlung in Windows-Anwendungen

Abfragebeispiel:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }
```

msg ist eine Struktur, die die aktuelle Nachricht aufnimmt.

GetMessage liefert den Wert Null zurück, wenn eine Endenachricht (*WM_QUIT*) empfangen wird. Dann wird die Anwendung beendet, wobei der Nachrichtenparameter *wParam* zurückgegeben wird.

Ansonsten werden zwei Systemfunktionen aufgerufen:

- *TranslateMessage* übersetzt Tastatureingaben,
- *DispatchMessage* ruft das Programmstück der Anwendung auf, das das jeweilige Ereigneis behandelt (Fensterprozedur WndProc).

Beispiel eines Ereignisbehandlers:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_CREATE :
               PlaySound ("hellowin.wav", NULL, SND_FILENAME | SND_ASYNC) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               DrawText (hdc, "Hello, Windows!", -1, &rect,
                              DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }


     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

Die Parameter der Fensterprozedur (WndProc) entsprechen den ersten vier Feldern der Nachrichtenstruktur *msg*.

Der Parameter iMsg enthält einen Zahlenwert, der kennzeichnet, um welche Art von Nachricht es sich handelt.

Diese Angabe wird mit einer *switch*-Anweisung ausgewertet, um zu den einzelnen Behandlungsabläufen zu verzweigen.

Gibt es für die betreffende Nachricht keine Behandlungsroutine, so ist die Systemfunktion *DefWindowProc* aufzurufen.

Ereignisbehandler unter Windows
Was wir hier kurz vorgestellt haben, betrifft die Anwendungsprogrammschnittstelle des Systems (Windows API). Es gibt Entwicklungsumgebungen (z. B. Visual Basic oder Delphi), die dem Programmierer die Auswertung der Nachrichten abnehmen und den Steuerelementen, Dialogen usw. entsprechende Ereignisse zuweisen. Solche Ereignisse heißen z. B. KeyDown (Taste gedrückt), KeyUp (Taste losgelassen) und KeyPress (allgemeine Tastenbetätigung). Der Anwendungsprogrammierer muß

sich nur noch mit den ausgelösten Funktionen beschäftigen und für jede Funktion einen Ereignisbehandler schreiben. Aus Sicht von Visual Basic, Delphi usw. entspricht Windows dem Schema von Abb. 6.2 (einzelne Ereignisbehandler, die vom System gestartet werden). Wie der Start eines Ereignisbehandlers im einzelnen abläuft, bekommt ein VB- oder Delphi-Programmierer nie zu sehen ...

Ereignisse in PL/1
PL/1 ist eine Programmiersprache, die vor allem für die EDV-Programmierung entwickelt wurde. Sie enthält Vorkehrungen zur Unterstützung des Multitasking. Ereignisse dienen zum Synchronisieren von Tasks. Eine Ereignisvariable hat zwei Werte:

- Vollendungswert. Besagt, ob das Ereignis eingetreten ist oder nicht. Wird bei Eintritt in eine angeschlossene Task auf 0 gesetzt. Bei Beendigung dieser Task hat er den Wert 1.
- Zustandwert. Zeigt an, ob das Ereignis durch eine abnorme Bedingung eingetreten ist.

Definition einer Ereignisvariablen: EVENT (Ereignisname).

Abfrage, ob ein Ereignis eingetreten ist:

COMPLETION (Ereignisname). 0 = nicht eingetreten; 1 = eingetreten.

Abfrage, ob das Ereignis normal oder abnormal ausgelöst wurde:

STATUS (Ereignisname). 0 = normal; 1 = abnormal.

Warten auf das Eintreten von Ereignissen:

WAIT (Ereignisname 1, Ereignisname 2..., Anzahl der eingetretenen Ereignisse). Über die Anzahlangabe kann gesteuert werden, wieviele Ereignisse eingetreten sein müssen, um den Programmablauf fortzusetzen.

Beispiel 1: konjunktive Verknüpfung (es müssen alle Ereignisse eingetreten sein):

```
CALL P1 Event (E1);
CALL P2 Event (E2);
WAIT (E1, E2) (2);
```

Hier geht es erst dann weiter, nachdem beide Tasks P1 und P2 beendet wurden.

Beispiel 2: disjunktive Verknüpfung (es genügt, daß eines der Ereignisse eingetritt):

```
CALL P1 Event (E1);
CALL P2 Event (E2);
WAIT (E1, E2) (1);
```

Es geht bereits dann weiter, nachdem eine der beiden Tasks P1, P2 beendet wurde.

7. Das Unterbrechungssystem

Das Unterbrechungssystem sorgt dafür, daß der Prozessor auf Ereignisse reagieren kann, ohne die auslösenden Bedingungen mit Befehlen ausdrücklich abfragen zu müssen. Auslösende Bedingungen können außerhalb des Prozessors entstehen oder innerhalb des Prozessors wirksam werden. Externe Bedingungen führen zu *Unterbrechungen* (Interrupts); interne Bedingungen führen zu *Ausnahmen* (Exceptions).

Jeder Interrupt bewirkt, daß der Prozessor vom laufenden Programm auf die Ausführung eines neuen Programms umgeschaltet wird, wobei Vorkehrungen getroffen werden, um später das unterbrochene Programm fortsetzen zu können. Das Programm, das durch die Unterbrechung aktiviert wird, heißt *Behandlungsprogramm* bzw. Behandler (Interrupt bzw. Exception Handler).

Unterbrechung oder Abfrage?
Im folgenden geht es, ganz allgemein gesagt, darum, wie man den Computer dazu bringt, auf Ereignisse zu reagieren. Einige weitere Beispiele solcher Ereignisse:

- der Support der zu steuernden Drehbank fährt gegen den rechten Anschlag,
- es kommt ein Divisionsbefehl zur Ausführung und der Divisor ist Null (es wird also versucht, durch Null zu dividieren),
- beim Rechnen mit ganzen Binärzahlen tritt eine Überlaufbedingung auf,
- das aktuelle Programm versucht, ein Byte im Speicherbereich des Betriebssystems zu überschreiben; gleichgültig, ob infolge einer fehlerhaft errechneten Adresse, ob aus krimineller Absicht (Virus),
- die Netzspannung unterschreitet einen vorgegebenen Mindestwert.

Daß man für jedes derartige Ereignis ein Programm schreiben kann, das darauf zweckmäßig reagiert (das Ereignis behandelt), ist an sich klar. Uns interessiert hier nicht die Art und Weise der jeweiligen Behandlung, sondern lediglich, wie man das Ereignis und das (an sich gegebene) behandelnde Programm zusammenbringt.

Voraussetzungen seitens der Hardware
Um die auslösenden Bedingungen zu erkennen, sind technische Mittel erforderlich (Abb. 7.1). Für Bedingungen, die innerhalb des Computers wirksam werden, braucht man entsprechende Überwachungs- bzw. Kontrollschaltungen. Im Falle externer Bedingungen, beispielsweise aus dem zu steuernden Embedded Systems, sind oft noch Einrichtungen zur Signalwandlung und zum Anschluß an das jeweilige Interface erforderlich. Wir wollen aber hier von solchen Einzelheiten absehen und annehmen, daß je Ereignis eine Signalleitung von den Erkennungs-Schaltungen zum Prozessor vorgesehen ist.

Das Abfrageprinzip
Wir können diese Leitungen so anschließen, daß sie mit Eingabe- oder anderen Transportbefehlen abfragbar sind, beispielsweise indem ein solcher Befehl den Belegungszustand dieser Leitungen in ein Universalregister überführt (Fachbegriff: Polling). Wollen wir auf ein Ereignis reagieren, müssen wir einen derartigen Transportbefehl ausführen, danach die betreffende Bitposition im besagten Register abfragen und schließlich zum jeweiligen Behandlungsprogramm verzweigen. Damit der Computer auch weiterhin Ereignisse behandeln kann, muß das Behandlungsprogramm zur Abfrage zurückkehren; es handelt sich also um eine Programmschleife aus Abfrage- und Behandlungsprogrammen (Abb. 7.2). Wenn "nichts" passiert, also keines der abgefragten Ereignisse aufgetreten ist, muß die Abfrage ständig wiederholt werden. Solche Schleifen werden als Abfrage- bzw. Steuerschleifen (Polling bzw. Control Loops) bezeichnet.

Das Unterbrechungsprinzip
Alternativ dazu kann man die Signalleitungen mit Schaltmitteln verbinden, die bei Auslösung eines Ereignisses den Prozessor *zwangsweise* auf die Ausführung des Behandlungsprogramms umschalten (Abb. 7.3). Dabei muß natürlich die Möglichkeit gegeben sein, das bisher laufende Programm nach der Behandlung wieder fortzusetzen.

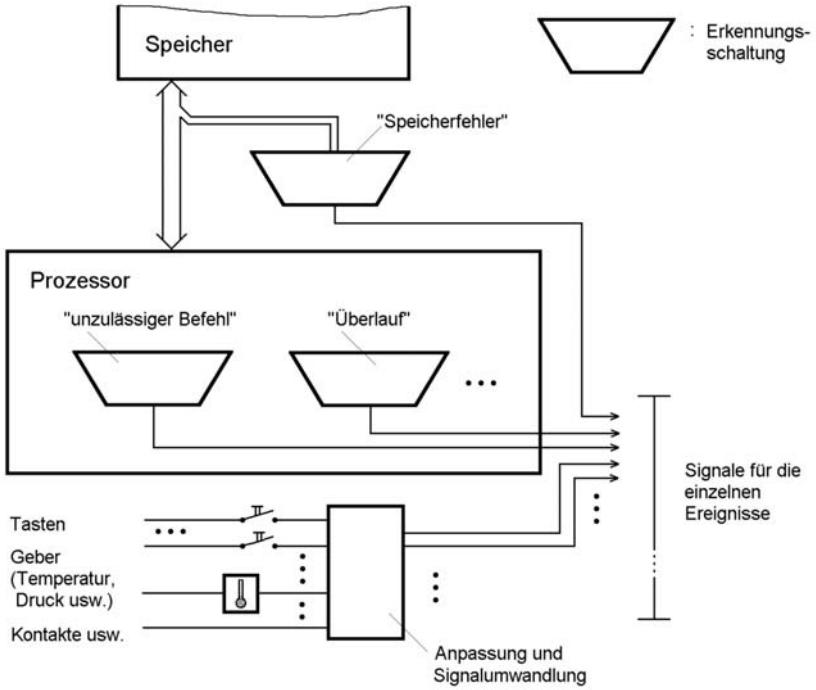


Abb. 7.1 Hardwareseitige Voraussetzungen, um auf Ereignisse reagieren zu können

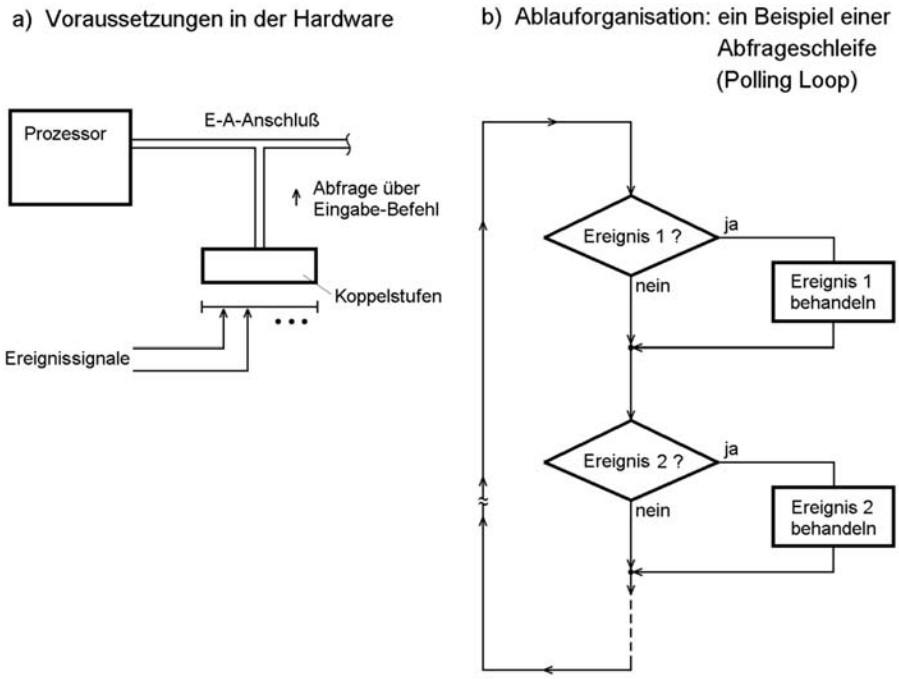


Abb. 7.2 Abfrageprinzip

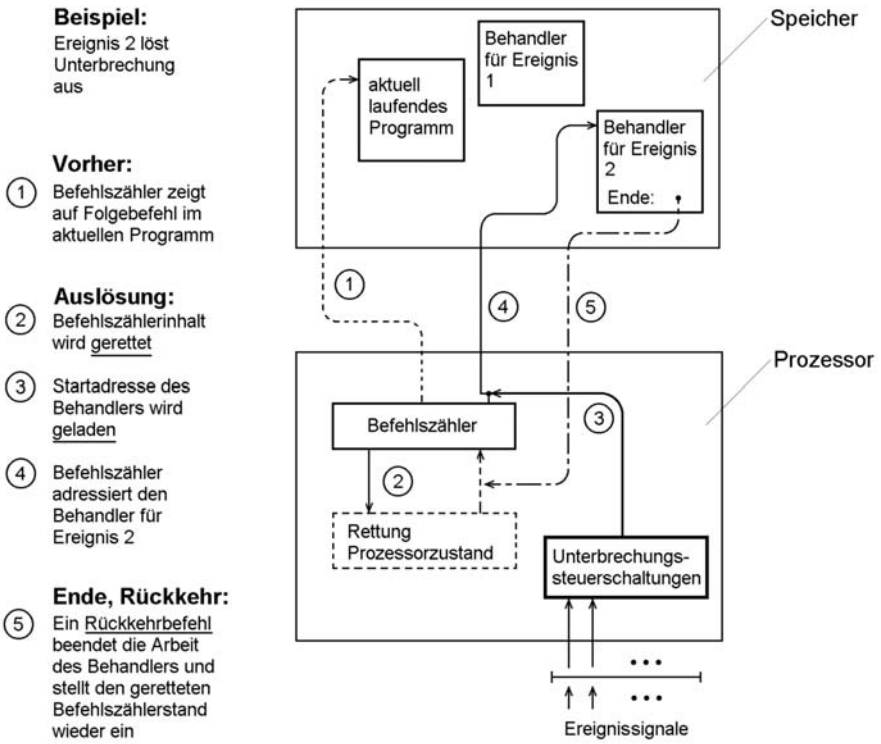


Abb. 7.3 Unterbrechungsprinzip

Wie können wir sicherstellen, daß das unterbrochene Programm fortgesetzt werden kann? - Hierzu sind architektur- und hardwareseitige Vorkehrungen erforderlich, die folgendes leisten müssen:

1. es muß möglich sein, den jeweiligen Prozessorzustand (1) in unbeeinflußter Form zu retten (es darf nichts verstellt werden) und (2) - bei Rückkehr aus der Unterbrechungsbehandlung - wieder einzustellen (das unterbrochene Programm darf gar nicht merken, daß es unterbrochen worden ist),
2. in Sonderfällen (u. a. dann, wenn die erste Forderung zeitweilig nicht erfüllt werden kann) muß es möglich sein, die Unterbrechung zu verhindern.

Der Prozessorzustand
Wir müssen zum Zeitpunkt der Unterbrechung den Prozessorzustand retten und bei der Rückkehr aus dem Behandlungsprogramm wieder einstellen (Fachbegriffe: Save/Restore; sprich: Seef/Riehstohr). Zum Prozessorzustand gehört auf jeden Fall die Adresse des Folgebefehls. Darüber hinaus ist aber all das zu berücksichtigen, was sowohl vom unterbrochenen Programm als auch vom Unterbrechungsbehandler gleichermaßen verändert werden kann, beispielsweise Flagbits, Universalregister oder Speicherbereiche.

Retten und Wieder-Einstellen
Wenn es sich nur um den Befehlszähler handelt, so gibt es einen ähnlichen Ablauf: den Unterprogrammruf. In dieser Sichtweise ist eine Unterbrechung ein von der Hardware erzwungener Unterprogrammruf mit einer besonders bereitgestellten Aufrufadresse.

Im Extremfall muß beiden Programmen je eine komplette Umgebung (eine virtuelle Maschine) mit unabhängigen Registersätzen, Speicherbereichen usw. bereitgestellt werden. In dieser Sichtweise ist die Unterbrechung eine von der Hardware erzwungene Taskumschaltung.

Unterbrechungsauslösung
Wie kommen wir von den Signalen, die den einzelnen Ereignissen zugeordnet sind, zur Startadresse des jeweiligen Behandlungsprogramms?

Fest zugeordnete Adressen
Jedem Ereignis wird eine Startadresse zugeordnet. Diese kann fest verdrahtet sein oder aus ladbaren Registern entnommen werden.

Tabellenadressierung über Interruptvektoren
Jedem Ereignis wird eine laufende Nummer zugeordnet. Solche – binär codierten – laufenden Nummern heißen Interruptvektoren. Mit dem jeweiligen Interruptvektor wird auf eine Tabellenstruktur im Arbeitsspeicher (Interrupttabelle) zugegriffen, die für jeden Interruptvektor die zugehörige Startadresse enthält.

Abfragen und Verzweigen
Grundsätzlich reicht eine einzige Festadresse aus, jedes beliebige Ereignis zu behandeln, wenn der Unterbrechungsbehandler mit einer Abfrage beginnt und entsprechend zu den einzelnen Behandlungsprogrammen verzweigt.

Fest zugeordnete Adressen sind nur bei vergleichsweise wenigen Ereignissen praktikabel.Deshalb hat sich das Tabellenprinzip weithin durchgesetzt. Üblich ist ein 8-Bit-Interruptvektor, mit dem man insgesamt 256 verschiedene Ereignisse durchnumerieren kann (Beispiel: x86/IA-32). In der Praxis wird dieses Prinzip (Vectored Interrupts) häufig mit dem Abfrageprinzip kombiniert. Man hat dann die Ereignisse zu Gruppen zusammengefaßt, wobei für jede Gruppe zunächst über einen Interruptvektor ein Behandler gestartet wird, der dann durch gezielte Abfragen jeweils zum eigenlichen Behandlungsprogramm verzweigt.

In modernen Architekturen mit schnell ablaufenden Elementarbefehlen (RISC) werden oft nur wenige Eintrittspunkte (Interruptvektoren) vorgesehen. Die genaue Ursache der Unterbrechung ist dann durch Abfragen zu bestimmen (was man sich, ohne Nachteile hinsichtlich der Latenzzeit, auch leisten kann, da die Abfragebefehle vergleichsweise schnell ausgeführt werden). So gibt es in der Mips-Architektur nur drei Festadressen zum Starten der Unterbrechungsbehandlung.

Die Reihenfolge der Ereignisbehandlung: Prioritäten
Wenn mehrere Ereignisse gleichzeitig wirksam werden: in welcher Reihenfolge sind sie zu behandeln? Dafür ist ein Prioritätsschema vorzusehen. Die genaue Reihenfolge ist Sache des jeweiligen Anwendungsfalls; im Rahmen der Architektur selbst ist kaum mehr als eine Art grober Vorsortierung möglich. Offensichtlich müssen Notfälle Vorrang haben vor Ereignissen, die im Rahmen der normalen Nutzung auftreten, und schwerwiegende Notfälle sind wichtiger als Unregelmäßigkeiten im Verarbeitungsablauf. Beispielsweise ist ein verstellter Stackpointer ein schwererer Fehler als der Versuch, durch Null zu dividieren, und ein in den nächsten 20 ms drohender Zusammenbruch der Stromversorgung noch wichtiger als alles andere. Wird die genaue Unterbrechungsursache durch Abfragen ermittelt, so ergibt sich die Prioritätszuordnung aus der Abfrage-Reihenfolge, liegt also in der Verantwortung des Programmierers. Werden die einzelnen Ursachen mittels Interruptvektoren voneinander unterschieden, muß die Vorrangsteuerung bereits in der Architektur vorgesehen sein. Für Ereignisse, die im Prozessor enstehen, sind dort entsprechende Schaltmittel vorzusehen. Ansonsten ist die Prioritätszuordnung Schaltmitteln außerhalb des Prozessors zu übertragen. Dafür gibt es besondere Schaltkreise (Interruptcontroller).

Muß man Unterbrechungen verhindern können?
Jeder Prozessor, der über ein Unterbrechungssystem verfügt, hat eine Möglichkeit, Unterbrechungen programmseitig zu verhindern bzw. zu erlauben (Interrupt Disable/Enable). Dafür sind entsprechende Befehle vorgesehen.

Wozu ist so etwas notwendig? – Im Interesse geringer Latenzzeiten wird der Systementwickler bemüht sein, Unterbrechungen immer zuzulassen (in manchen Architekturen sind sogar Befehle unterbrechbar, die längere Ausführungszeiten haben können, z. B. Blocktransport- und Vergleichsoperationen). In Realzeitsystemen ist ständige Unterbrechbarkeit besonders wichtig. Manchmal muß man aber Unterbrechungen zeitweilig verhindern, und zwar, um zu gewährleisten, daß ein bestimmtes Programmstück "ungestört" ablaufen kann.

Wann sind Unterbrechungen zeitweilig zu verhindern? - Einige Beispiele:

- beim Retten und Wiederherstellen von Systemzuständen,
- beim Einrichten bzw. Verändern von Informationsstrukturen der Systemverwaltung,
- in Vermittlungsabläufen,
- bei Mangel an Ressourcen,
- beim Übergeben von Parametern an andere Programme (Tasks). Jede Task muß stets korrekte Parameter sehen, nicht irgendwelche Mischungen alt-neu.
- wenn bestimmte Antwortzeiten einzuhalten sind,
- wenn bestimmte Betriebsmittel nicht verfügbar sind.

Zeitkennwerte
Wie lange es dauert, das jeweilige Ereignis zu behandeln, ist – betrachten wir das System aus Prozessor, Speicher usw. als gegeben – anwendungs- und problemspezifisch. (Die Ereignisbehandlung hat die jeweils gewünschten Funktionen zu erbringen – die eben ihre Zeit erfordern. Es ist offensichtlich ein Unterschied, ob z. B. eine Tastenbetätigung nur eine einfache Cursorbewegung bewirkt oder ob sie einen komplizierten Rechenablauf auslöst.) Man unterscheidet deshalb zwei Zeitkennwerte:

1. die Zeitspanne vom Eintreffen des Ereignisses bis zum Start des Behandlungsprogramms (Latenzzeit (Latency)). Dieser Wert kennzeichnet das entsprechende Leistungsvermögen der Architektur (z. B. soundsoviel µs nach Auftreten der Unterbrechungsbedingung das Behandlungsprogramm starten zu können).
2. die Zeitspanne vom Auftreten eines Ereignisses bis zum Abschluß der anwendungsseitig gewünschten Reaktion des Systems (Zeit “über alles”). Dieser Wert kennzeichnet das entsprechende Leistungsvermögen der gesamten Anwendungslösung. Übliche Fachbegriffe: Reaktionszeit, Antwortzeit (Response Time). *Hinweis:* Diese Begriffe werden gelegentlich auch benutzt, um die Zeitspanne zu bezeichnen, die wir hier Latenzzeit nennen (im Fall des Falles also genau nachlesen, was eigentlich gemeint ist).

Einfache (lightweigth) Interrupts
Ein Interrupt ist dann einfach, wenn er schnell erledigt ist und wenig Ressourcen benötigt. Ablaufschema:

1. Interruptauslösung,
2. es wird nur das Nötigste gerettet (z. B. ein Teil des Registersatzes). Dabei gibt es typischerweise keine Stackumschaltung (Rettung auf dem gerade aktuellen Stack).
3. der Interruptbehandler begnügt sich mit den wenigen Registern, die gerettet wurden; er greift nur auf eigene Speicherbereiche zu und hat mit dem Rest des Systems nichts zu tun. Die Behandlungsroutine ist nicht unterbrechbar.
4. nach der Interruptbehandlung werden die geretteten Register wieder eingestellt, und die Rückkehr wird veranlaßt.

Anwendung: immer dann, wenn schnell reagiert werden muß (geringe Latenzzeiten) und wenn innerhalb der Interruptbehandlung nicht viel zu tun ist. Beispiele:

- Datenübertragung mit E-A-Einrichtungen,
- Reaktion auf Nulldurchgang des Netzsinus,
- Impulserzeugung (Triac, PWM, Schrittmotor).

Es gibt Mikrocontroller, die einfache Interrupts mit Hardware unterstützen (Beispiel: Z 80 (Austauschregister)).

Abfrage oder Unterbrechung?
Obwohl die geschichtliche Reihenfolge dies nahelegt, sollte man Abfrage- und Unterbrechungsprinzip nicht als aufeinanderfolgende Entwicklungsstufen ansehen; sie haben beide ihre Berechtigung. Wir wollen das Für und Wider etwas näher betrachten, und zwar unter den Gesichtspunkten (1) der Zeit und (2) der Programmorganisation. Dabei sehen wir das Behandlungsprogramm an sich weiterhin als gegeben an.

Was ist schneller?
Wenn es nur wenige Ereignisse gibt, die Abfrageschleife also kurz ist, hat das Abfrageprinzip oft die geringere Latenzzeit. Schließlich muß nichts gerettet werden. Als Latenzzeit muß man schlimmstenfalls die Dauer der längsten Schleife (unter Berücksichtigung eines gerade erst gestarteten Behandlers) ansetzen. Das Abfrageprinzip wird nur dann langsam, wenn viele Ereignisse abzufragen sind bzw. wenn die einzelnen Behandler lange Laufzeiten haben.

Praxistip:
Um festzustellen, ob die jeweiligen Anforderungen mit einem bestimmten System erfüllt werden können, bleibt oft nichts weiter übrig, als die Behandlung auszuprogrammieren und die Gesamtzeit durch Auszählen aus den Ausführungszeiten der einzelnen Befehle zu bestimmen. Diesem Wert sollte ein Sicherheitszuschlag hinzugerechnet werden. Bereich der Erfahrungswerte: zwischen 20% und vielleicht 60%, je nachdem, wieviele Änderungen noch zu erwarten sind und wie genau wir die Zeitverhältnisse der Befehlsausführung überblicken können.

Vorsicht, Falle:
Gerade bei Hochleistungsprozessoren ist dies nicht ganz einfach. In ungünstigen Fällen kann durch Nachladen von Caches oder – falls genutzt – durch Aktivitäten der Seitenverwaltung die Antwortzeit untragbar absinken. Noch schlimmer kommt es, wenn wir uns nicht direkt auf die Hardware oder auf ein wirklichess Realzeitbetriebssystem stützen (Richtwert der Interrupt-Latenzzeit unter Windows NT: über 20 ms...). Probleme in dieser Hinsicht äußern sich oft als zeitweilige (transiente) Fehler, die wie gelegentliche Fehlfunktionen der Hardware erscheinen. ("Gelegentlich" kann hier bedeuten, daß der Fehler in 14tägigem Abstand auftritt.)

Programmorganisation
Das Abfrageprinzip hat einen wichtigen Vorteil: den ungestörten Ablauf des Behandlungsprogramms. Nach dem Verzweigen zum Behandler steht vom "Rest" der Außenwelt nichts mehr zu befürchten; das Behandlungsprogramm läuft ab, ohne in irgendeiner Weise von außen gestört zu werden. Wir haben also klare, überschaubare Verhältnisse. Genau diese Tatsache kann aber auch in zweierlei Hinsicht von Nachteil sein: (1) was nicht abgefragt wird, kann auch nicht behandelt werden (anders herum: wenn wir irgend etwas behandeln wollen, müssen wir immer entsprechende Abfragen mitprogrammieren), (2) die Abfrageschleife belegt ständig den Prozessor; man kann die Zeiten, in denen keine Ereignisse eintreffen, nicht mit nützlicher Arbeit ausfüllen.

Um den ersten Nachteil zu veranschaulichen: es geht um seltene Ereignisse, im besonderen um Fehler, die normalerweise nie auftreten. Wenn entsprechende Vorkehrungen im Unterbrechungssystem fehlen, müßte man beispielsweise vor jedem Divisionsbefehl den Divisor auf Wert Null prüfen oder nach jeder Addition die Überlaufbedingung abfragen.

Abfragen sind dann einfach beherrschbar, wenn der zeitliche Abstand zwischen zwei Ereignissen immer größer ist als die längste Umlaufzeit der Schleife. Moderne Prozessoren sind so schnell, daß dies oft gegeben ist. (Andernfalls wird es kompliziert: man muß dann in die Behandler bedarfsweise weitere Abfragen einstreuen.)

Unterbrechungen sind das Mittel der Wahl, wenn es um "seltene" Ereignisse geht oder um Ereignisse, die eine vorrangige Behandlung erfordern, also notfalls auch die Unterbrechung eines bereits laufenden Ereignisbehandlers. In welcher Form "normale" Ereignisse behandelt werden, ist von Fall zu Fall zu entscheiden. In der Praxis finden wir oft Kombinationen beider Prinzipien.

Mikrocontroller trickreich beeinflussen – weitere Varianten
Es geht grundsätzlich darum, den Mikrocontroller zu veranlassen, auf externe Signale zu reagieren. Zusätzlich zu den bisher beschriebenen Lösungen – Abfrage oder Unterbrechung– gibt es gelegtlich weitere Möglichkeiten. Sie sind in folgenden Fällen anwendbar:

- ohne Rücksicht auf den aktuellen Programmablauf ist ein Programmstart zu erzwingen. Lösung: Rücksetzen, ggf. kombiniert mit Abfragevorkehrungen, um die jeweilige auslösende Ursache bestimmen zu können,
- es ist auf externe Ereignisse zu warten. Lösungen: die Befehlsausführung in der Hardware anhalten, z. B. durch Abschalten des Taktes oder Entzug der Busherrschaft.

Hinweise:

1. Die Taktabschaltung hat die geringste Latenzzeit unter allen Implementierungen der Wartefunktion. Es ist aber zu bedenken, daß davon auch Zeitgeber (Counter/Timer) betroffen sein können. Ausweg: Zeitgeber mit besonderem Takt versorgen (wird in manchen Mikrocontrollern unterstützt).
2. Treten Anforderungen höherer Priorität auf, müssen hart wirkende Wartezustände abgebrochen werden.

Tabelle 7.1 gibt einen Überblick über typische Alternativen, einen Mikrocontroller von außen zu beeinflussen. Tabelle 7.2 veranschaulicht typische Zeitverhältnisse (wobei es sich von selbst versteht, daß die betreffenden Untersuchungen für jeden Mikrocontrollertyp jeweils neu anzustellen sind).

Prinzip	Wirkung	Auswirkung auf Programmablauf	Programmfortsetzung	unterdrückbar	Eignung für
Rücksetzen	Programmstart von fester Adresse (0) ; Rücksetzen der E-A-Schaltkreise	völliger Neubeginn erforderlich	nein	nein*)	Initialisierung, Fehlersignalisierung
Nichtmaskierbarer Interrupt (NMI)	Rettung des aktuellen Befehlszählers in den Stack, Programmstart von fester Adresse	beliebig programmierbar, Rückkehr zum unterbrochenen Programm möglich	möglich, wenn entsprechend programmiert	nein*)	Signalisierung von Fehlern und anderen außergewöhnlichen Zuständen
Interrupt	Rettung des aktuellen Befehlszählers in den Stack, Start einer spezifischen Behandlungsroutine	beliebig programmierbar, Rückkehr zum unterbrochenen Programm möglich	möglich, wenn entsprechend programmiert	ja	Signalisierung von Aufträgen
Verändern von Speicherinhalten	Verzweigung im Rahmen der Softwarekonventionen des Systems	beliebig programmierbar, die Behandlung obliegt ausschließlich der Software	möglich, wenn entsprechend programmiert	ja	Signalisierung von Aufträgen
WAIT-Leitung erregen	der aktuelle Maschinenzyklus der CPU wird so lange verlängert, bis WAIT inaktiv wird	nur Verlangsamung	ja	mit entsprechender Hardware möglich	Anpassung an Dauer der Zugriffe
BUS-REQUEST-Leitung erregen	der folgende Zyklus wird nicht ausgeführt, der interne Bus wird von der CPU freigegeben	nur Verlangsamung	ja	mit entsprechender Hardware möglich	„Freischalten" des internen Busses (z. B. für DMA)
CPU-Takte stoppen bzw. verlangsamen	der aktuelle Taktzyklus dauert entsprechend länger	nur Verlangsamung	ja	mit entsprechender Hardware möglich	Anpassung an Dauer der Zugriffe

*) Zusatzhardware zum Unterdrücken ist aufwendig und nicht besonders sinnvoll

Tabelle 7.1 Mikrocontroller von außen beeinflussen – typische Alternativen

Ablauf	Befehlsfolge	Bemerkungen	Zeitbedarf (Zykluszeit 400 ns)
Hardware zurücksetzen	Befehl von Adresse 0 ist der erste der gewünschten Steuerroutine	Es darf keine Initialisierung o. ä. erforderlich sein	1,6 µs
NMI	Signal wird am Anfang eines langen Befehls [z. B. SET 3, (IY+8)] wirksam; am Ende wird NMI ausgelöst (PC retten, zu Adresse 66H verzweigen)	Es gibt kein Retten von Registerinhalten. Rückkehrbefehl wird mitgezählt	19 µs
Interrupt	HL: HALT JR HL-#	Es gibt kein Retten von Registerinhalten.	
	Interrupt-Mode 2, Rückkehrbefehl (RETI) wird mitgezählt	2 Fälle	
		1. Signal wirkt direkt (z. B. über spezielle Interrupthardware)	≈ 12 µs
		2. Signal wirkt über STB eines PIO-Ports	≈ 12 µs + Dauer von STB
BUS REQUEST	Die CPU ist solange vom Bus getrennt, bis das Signal aktiv wird	Der Bus wird z. B. am Ende eines speziellen OUT-Befehls abgeschaltet (der auf die Steuerhardware wirkt). Der Folgebefehl ist dann der erste der Steuerroutine	> 800 ns
WAIT	Die CPU verharrt im WAIT-Zustand, bis das Signal aktiv wird	WAIT wird durch Dekodieren der Speicheradressen gebildet, z. B. LD HL, Adresse LD A,(HL); dieser Befehl führt zu WAIT	> 400 ns
Abfrage über Speicheradressen	LD HL, Adresse BA: BIT n,(HL) JRZ BA-#	Schleife dauert 7,6 µs	> 14 µs
	bzw. LD HL, Adresse BA: BIT n,(HL) JPZ BA	Schleife dauert 8,8 µs	> 13 µs
Abfrage über E-A-Adressen (z. B. PIO-Port im Bitmodus)	BA: IN Port BIT n,A JRZ BA-#	Schleife dauert 10 µs	> 12 µs

Tabelle 7.2 Auf externe Signale reagieren – Zeitverhältnisse am Beispiel des Mikroprozessors Z80 (Taktzyklus 400 ns)

8. Anwendungsbeispiel: Steuerung eines Staubsaugers

– Nach Freescale Semiconductor –

Der Mikrocontroller übernimmt die Abfrage der Bedienelemente und die Drehzahlsteuerung des Motors. Das Beispiel betrifft die einfachste Ausführung (Abb. 8.1). Es geht lediglich darum, ein Potentiometer abzufragen und ein Triac so anzusteuern, daß sich die gewünschte Drehzahl ergibt (nur Steuerung, keine Regelung). Abb. 8.2 veranschaulicht die Grundlagen der Phasenanschnittsteuerung.

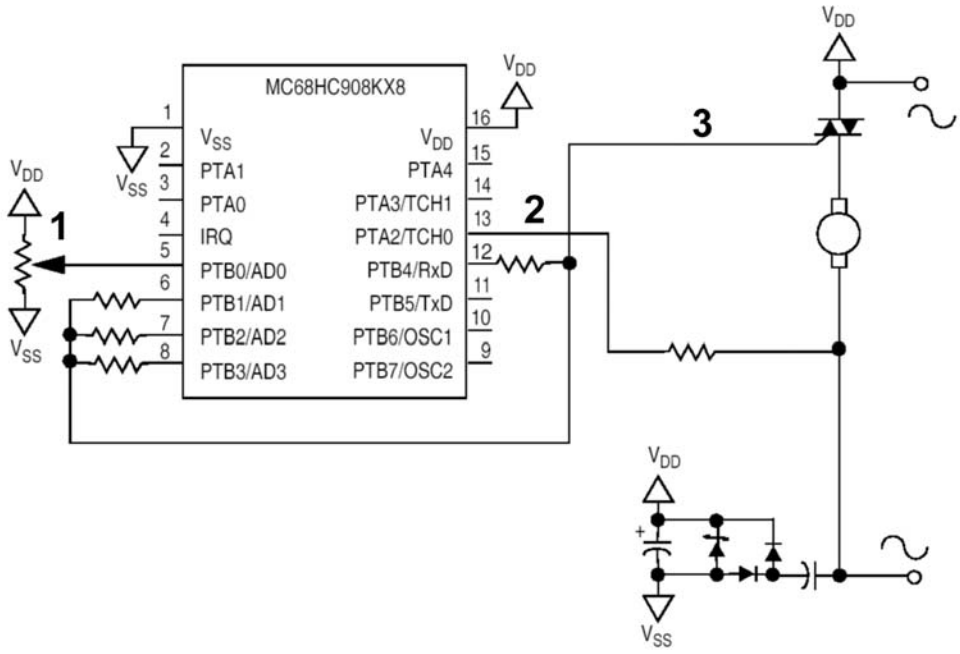


Abb. 8.1 Die Hardware im Blockschaltbild. 1 - Potentiometer zur Drehzahleinstellung; 2 - Nulldurchgangserkennung; 3 - Triac-Ansteuerung

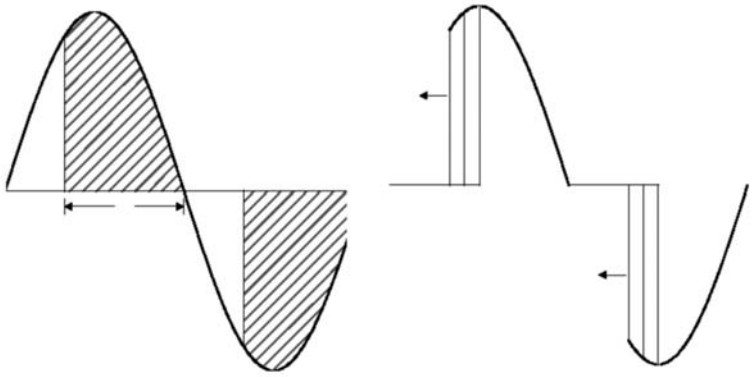


Abb. 8.2 Grundlagen der Phasenanschnittsteuerung. Zündimpuls veranlaßt, daß Triac zündet. Triac bleibt leitend bis zum nächsten Nulldurchgang des Stromes

Der Mikrocontroller erkennt den Nulldurchgang der Spannung. Der Nulldurchgang des Stroms eilt aber dem der Spannung nach (induktive Last). Bei großem Stromflußwinkel kann es vorkommen, daß der Zündimpuls zu früh (= vor dem Nulldurchgang des Stroms) abgegeben wird. Geht dann der Strom durch Null, ist kein Zündimpuls mehr da. Lösung: bei größeren Stromflußwinkeln (135...180°) werden breitere Zündimpulse abgegeben (Abb. 8.3, Tabelle 8.1). (Alternativen: (1) Stromfluß = 0 erkennen, (2) Wellenpaketsteuerung.)

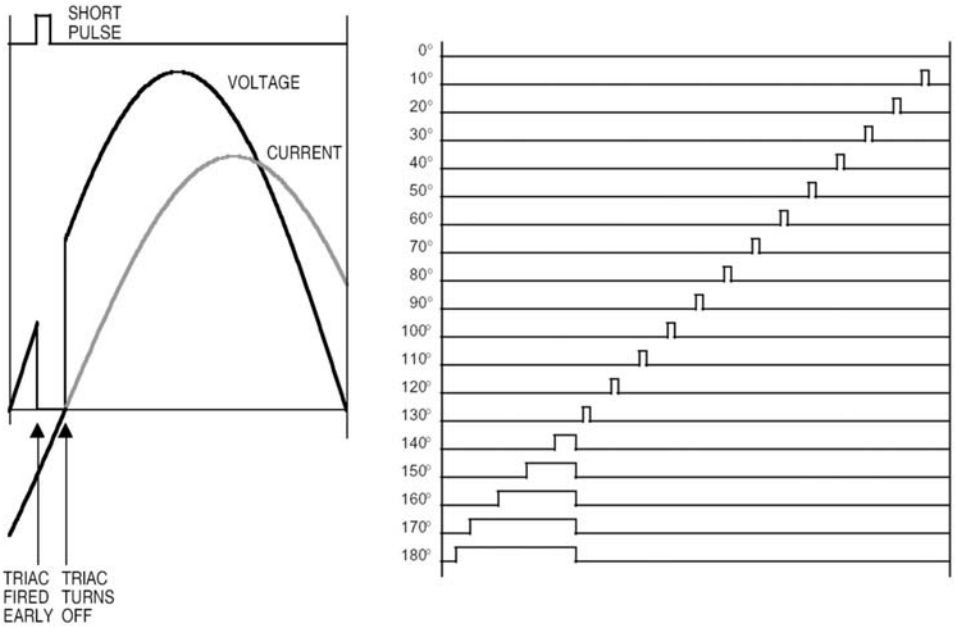


Abb. 8.3 Zündimpulsbreite in Abhängigkeit vom Stromflußwinkel

Stromflußwinkel φ	Aktion
0° < φ < 5°	– (Triac wird nicht angesteuert)
5°< φ < 135°	kurzer Impuls zum Zeitpunkt φ
135° < φ < 175°	Impuls ein zum Zeitpunkt φ, Impuls aus bei 135°
175° < φ <180°	Impuls ein sobald als möglich (sofort nach Nulldurchgang), Impuls aus bei 135°

Tabelle 8.3 Zündimpulserzeugung in Abhängigkeit vom Stromflußwinkel

Es sind die Nulldurchgänge des Netzsinus zu erfassen. Diese bilden die Grundlage für die Ermittlung der Periodendauer und für die Auslösung der Zündimpulse. Die Zeit zwischen zwei Nulldurchgängen enstpricht einem Phasenwinkel von 180°. Die Zeit vom Nulldurchgang bis zum Auslösen des Zündimpulses ergibt sich durch entsprechende Dreisatzrechnung. Der Zündwinkel ergibt sich aus der Potentiometerstellung. Die gewählte Programmorganisation entspricht einer Kombination aus Abfrageschleife für die Bedienvorgänge (hier auf das Abfragen des Potentiometers beschränkt) und Unterbrechungauslösung für die zeitabhängigen Abläufe (Abb. 8.4). Hierzu wird der Zeitgeber (Counter/Timer) des Mikrocontrollers ausgenutzt.

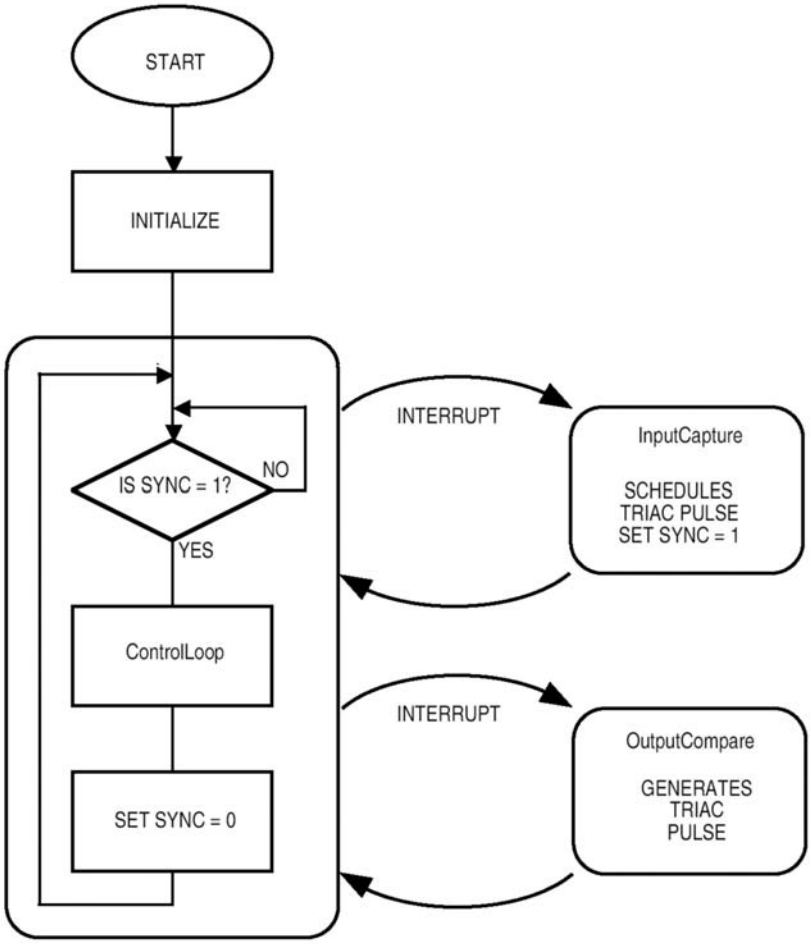


Abb. 8.4 Programmorganisation

Interrupt InputCapture
Bei Nulldurchgang der Netzspannung wird ein Interrupt ausgelöst. Der Zeitgeberstand wird gelesen. Aus den Werten von zwei Zyklen wird der Durchschnittswert der Periodendauer errechnet. Die Interruptroutine setzt das SYNC-Flag. Bei gesetztem SYNC-Flag wird die Steuerschleife einmal durchlaufen.

Interrupt OutputCompare
Diese Interruptroutine erzeugt die Zündimpulse fürs Triac. Sie wird über einen Zeitgeberkanal ausgelöst. Die Zündimpulse werden mit Hilfe zweier Zeitintervalle gebildet (Abb. 8.5):

- nach dem ersten Intervall wird der Impuls eingeschaltet, und der Zeitgeber wird auf das zweite Intervall eingestellt.
- nach dem zweiten Intervall wird der Impuls ausgeschaltet, und der Zeitgeber wird auf das erste Intervall eingestellt.

Steuerschleife (Control Loop)
Liest die aktuelle Potentiometerstellung über A-D-Wandler. Wertebereich des Wandlers: 0...255. Ausrechnung des Stromflußwinkels: (gelesener Wert • 180) : 255.

Weitere Funktion nach dem Einschalten: langsames Hochfahren des Motors auf die jeweils gewünschte Drehzahl (Soft Start). Hierzu wird ein Phasenwinkel gemäß einer Rampenfunktion erzeugt (Akkumulation in konstanten Intervallen gemäß einer Integrationsrate). Erfahrungswert: Hochlauf über ca. 3 s.

Übergabe der errechneten Intervallangaben: bei gesetzter SYNC-Flag. Nach Durchlauf der Steuerschleife wird die SYNC-Flag wieder gelöscht.

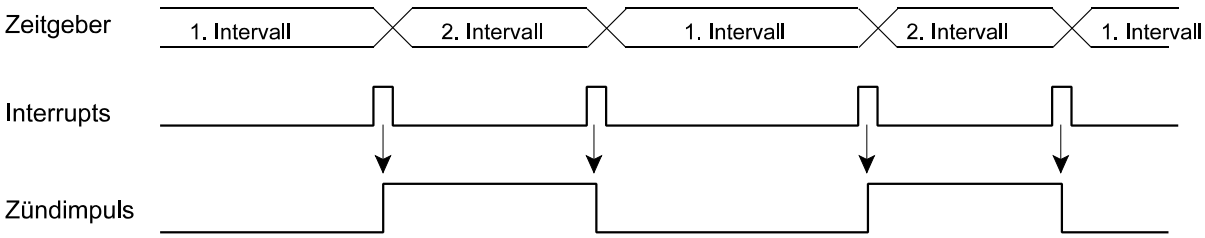


Abb. 8.5 Zündimpulsbildung mittels Zeitgeber und Interruptroutinen

Der Mikrocontroller ist die meiste Zeit nicht beschäftigt (Abb. 8.6). Der InputCapture-Interrupt tritt bei jedem Nulldurchgang auf. Da sollte der Mikrocontroller normalerweise unbeschäftigt sein. Bei kleinen Stromflußwinkeln tritt OutputCompare spät auf, und zwar zweimal kurz hinereinander (kurzer Zündimpuls). Wenn der Stromflußwinkel aber 180° erreicht, kann es vorkommen, daß OutputCompare die Steuerschleife unterbricht.

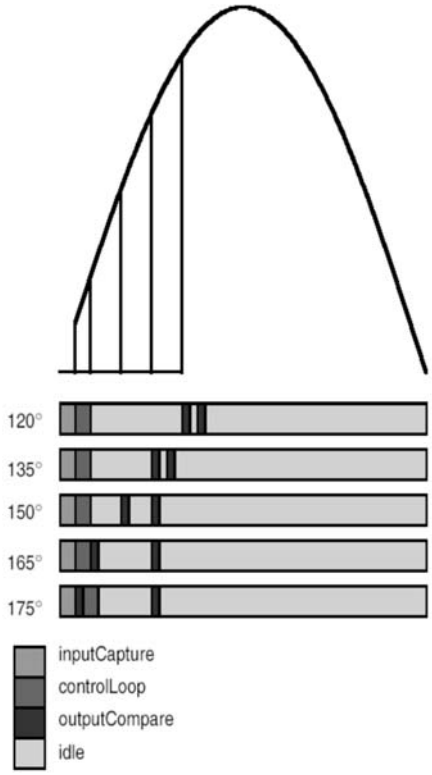


Abb. 8.6 Die Auslastung des Mikrocontrollers in Abhängigkeit vom Stromflußwinkel. Die meiste Zeit hat er nichts zu tun ...