6 Hard Real-Time Scheduling - DRS Methode

Mit der Ausführungssteuerung eines Systems ist dafür zu sorgen, dass die zeitlichen Anforderungen erfüllt werden. Dabei unterscheidet man harte und weiche Echtzeitbedingungen.

***Harte Echtzeitbedingungen* (hard real-time conditions).** Das Einhalten der harten Echtzeitbedingungen ist notwendig, damit das System funktionieren kann. Das sind typischerweise Anforderungen an die Reaktionszeiten des Systems, die mittels *Deadlines* spezifiziert werden. Wenn eine harte Zeitbedingung verletzt wird, muss damit gerechnet werden, dass das System die angeforderte Funktionalität nicht mehr erfüllen kann. Das kann z. B. lediglich bedeuten, dass ein automatisiert hergestelltes Produkt nicht brauchbar ist. Es kann aber auch grosser Schaden an der Anlage entstehen, und bei sicherheitskritischem Fehlverhalten muss mit Gefahren für Mensch und Umwelt gerechnet werden. Das Einhalten harter Echtzeitbedingungen sollte deshalb garantiert werden können. Das heisst es müsste bewiesen werden können, dass immer alle harten Echtzeitbedingungen eingehalten werden (Schedulability Verification), was in der industriellen Praxis aber selten möglich ist.

***Weiche Echtzeitbedingungen* (soft real-time conditions).** Zeitbedingungen, die nur im Mittel eingehalten werden müssen, oder deren Einhaltung erwünscht ist, weil sie die Performance des Systems verbessert, heissen weiche Zeitbedingungen. Das betrifft typischerweise Anforderungen wie. z. B. das Erreichen eines bestimmten Durchsatzes, sanfte Regelung gewisser mechanischer Prozesse oder ein angenehmes Verhalten der Benutzerschnittstellen. Werden solche Bedingungen verletzt, funktioniert das System zwar noch richtig, aber es ist unter Umständen schlecht brauchbar.

In diesem Kapitel wird auf das Scheduling bei *harten* Echtzeitbedingungen eingegangen. Für das Lösen weicher Scheduling-Probleme, welche typischerweise die Performance der Systemausführung betreffen, können meistens Scheduling-Verfahren angewendet werden, die man von Desktop-Systemen kennt.

DRS Methode (Domain-oriented Real-time Scheduling)

Die in Kapitel 2 beschriebene domänenorientierte Software-Architektur zeigt bereits, dass die Steuerung der Systemausführung auf zwei verschiedenen Ebenen erfolgen kann. Dementsprechend vermittelt die *DRS Methode* Konzepte, mit welchen die Ausführungssteuerung eines Embedded Systems als zweistufige Scheduling-Hierarchie organisiert wird. Auf der Prozessor-Ebene muss jeder Embedded Unit genügend Prozessorzeit zur Verfügung gestellt werden. Das ist zu erreichen, indem man die Embedded Units preemptiv als sog. *periodische Server* ausführt. Dabei wird in der Regel ein Echtzeit-Betriebssystem verwendet. Für die lokale Ausführung der Interaktionsfunktionen und der Reaktiven Maschine der einzelnen Embedded Units kann so der Anteil der Prozessorzeit pro Unit-Zyklus garantiert werden.

Zudem muss auf Unit-Ebene pro Embedded Unit die lokale Verarbeitung anstehender Ereignismeldungen so organisiert werden, dass vorgegebene Reaktionszeiten eingehalten werden. Dieses lokale non-preemptive Schedulingproblem wird ohne Betriebssystem durch eine geeignete Organisation der *Event Message Queue* gelöst.

Der grosse Vorteil der entstehenden Scheduling-Hierarchie besteht darin, dass die Ausführung der Interaktionsfunktionen und der Reaktiven Maschine innerhalb einer Embedded Unit und das Unit-Scheduling auf Prozessorebene durch nur einen Parameter gekoppelt ist, dem garantierten Prozessorbenutzungsfaktor U der Unit. Wenn das Scheduling auf der Prozessorebene geändert wird, hat das nur einen Einfluss auf diejenigen Units, deren Prozessorbenutzungsfaktor verändert wurde. Wird hingegen das lokale Scheduling in einer Unit verändert, sind die andern Units überhaupt nicht davon betroffen. Das erleichtert das Finden der optimalen Systemausführung beträchtlich, und wird vor allem bei Systemänderungen wichtig. Vergleichbare hierarchische Scheduling-Ansätze sind kürzlich auch für reglerbasierte Systeme und für nicht-embedded Applikationen entwickelt worden [11, 13].

6.1 Zyklusbasierte Ausführung einer Embedded Unit

Bemerkung. In diesem Unterkapitel werden bereits einige Scheduling-Begriffe benutzt, die erst im übernächsten Unterkapitel im Detail behandelt werden.

Dieses Unterkapitel behandelt die Organisation der Ausführung der Interaktionsfunktionen und der Reaktiven Maschine einer einzelnen Embedded Unit. Dabei wird eine generische Ausführungsstruktur für Embedded Units vorgeschlagen, die den domänenorientierten Entwicklungsansatz berücksichtigt und ein verifizierbares Scheduling der Ereignisverarbeitung ermöglicht.

Mit DEC-Modellen sind die Datenpfade der Verbindungssoftware bereits definiert. Für die Aktivierung der Interaktionsfunktionen und der Reaktiven Maschine ist ein zusätzlicher aktiver Programmteil notwendig, der als *Unit Controller* bezeichnet wird, die sog. “Main Function” der Unit-Task. Die Programmstruktur des Unit Controller bestimmt damit die zeitliche Ordnung, in der die Interaktionsfunktionen und die Reaktiven Maschine ausgeführt werden. Damit ist der Unit Controller für das Einhalten der Echtzeitbedingungen verantwortlich. Der Unit Controller wird zwar “von Hand” programmiert, aber die DEC-Methode unterstützt den Design des Unit Controller durch klare Ausführungskonzepte.

6.1.1 Organisationsformen der Software-Ausführung

Die spezifizierten Datenpfade der Unit Software bestimmen bereits einen Teil die Ausführungsreihenfolge durch das natürliche Kausalitätsprinzip: *Input* vor *Function* vor *Output.* Die Input- und Output-Datenpfade sind aber in parallelen *Input* und *Output Channels* organisiert, was einen beträchtlichen Spielraum für die zu realisierende Ausführung ergibt.

In *Abbildung 61* sind zwei extreme Organisationsformen der Ausführung dargestellt, die das ‘*Input* vor *Function* vor *Output’*-Prinzip einhalten.

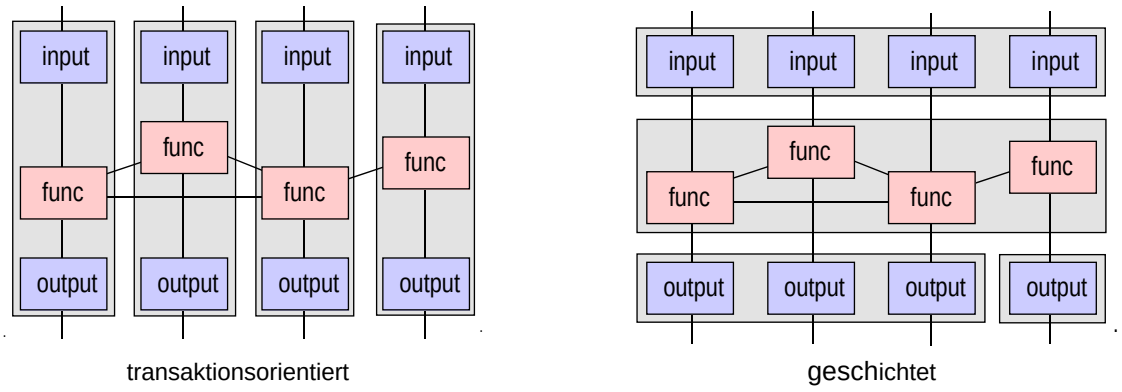


Abb. 61: Zwei extreme Organisationsformen der Software Ausführung

Bei der *transaktionsorientierten* Organisation werden bei der Ausführung diejenigen *Input*-, *Func*- und *Output*-Codeblöcke in einer Prozedur zusammengefasst, die zur selben Transaktion gehören. Diese sog. *vertikale* Strukturierung ist bekannt aus der klassischen Datenverarbeitung. Sie entsteht implizit auch bei objekt-orientierten Entwürfen (Aktivierungsketten von Objekten). Die auf den ersten Blick natürliche Strukturierung wirkt sich dann ungünstig aus, wenn zwischen den einzelnen Transaktionen starke funktionale Abhängigkeiten bestehen, und das ist bei vielen Embedded Systemen der Fall.

Die *geschichtete* Organisation in *Abbildung 61* fasst Codeblöcke derselben Softwareschicht in entsprechenden Prozeduren zusammen, eine *Input*-, eine *Func*- und zwei *Output*-Prozeduren. Der horizontale Strukturierungsansatz macht Sinn, wenn für die Input und Output Funktionen mit horizontal übergreifenden Datenstrukturen gerarbeitet werden muss, wie z. B. das Einlesen eines Ports, dessen Bit-Signale Information für verschiedene Funktionen liefern. Vor allem aber können Ausführungs-

konflikte aufgrund horizontaler funktionaler Abhängigkeiten relativ einfach innerhalb der funktionalen Prozedur gelöst werden.

Mit dem domänenorientierten Entwicklungsansatz ist die *geschichtete* Organisation der embedded Aktivitäten zum Teil bereits vorgegeben. Durch die Aufteilung der embedded Software in *Reaktive Maschine* und *Embedded-Connection* hat man sich auf Unit-Ebene bereits für eine Ausführung der Steuersoftware in einer eigenen Schicht entschieden, als *Reaktive Maschine* mit Run-to-Completion Semantik. Für die Strukturierung der Ausführung der *Interaktionsfunktionen* wird im Folgenden eine generische Ausführungsstruktur für Embedded Units festgelegt. Dabei zeigt sich im Hinblick auf das RT-Scheduling, dass auch bei der Verbindungssoftware die geschichtete Organisation der Ausführung von Vorteil ist. Wenn die Einhaltung harter Echtzeitbedingungen garantiert werden muss (*Schedulability Verification*), ist es entscheidend, dass die Ausführungsstruktur generisch und einfach ist, was nur bei der geschichteten Ausführung der Fall ist. Die transaktionsorientierte Ausführung hingegen kann aufgrund der zum Teil zyklischen funktionalen Abhängigkeiten weder generisch noch einfach sein, was bei beim Garantieren der Einhaltung von Echtzeitbedingungen schnell zu unlösbaren Problemen führt. Weitere Schwierigkeiten treten bei Systemänderungen auf, weil immer damit gerechnet werden muss, dass die applikationsspezifische Ausführungsstruktur zu überarbeiten ist. Bei der geschichteten Ausführung sind lediglich Änderungen beim Event Scheduling der Reaktiven Maschine zu erwarten.

6.1.2 Generische Ausführungsarchitektur einer Embedded Unit

in *Abbildung 62* ist die generische Ausführungsarchitektur *Embedded Unit* dargestellt. Die Darstellung zeigt, wie welche Datenflüsse die auszuführenden Komponenten verbinden.

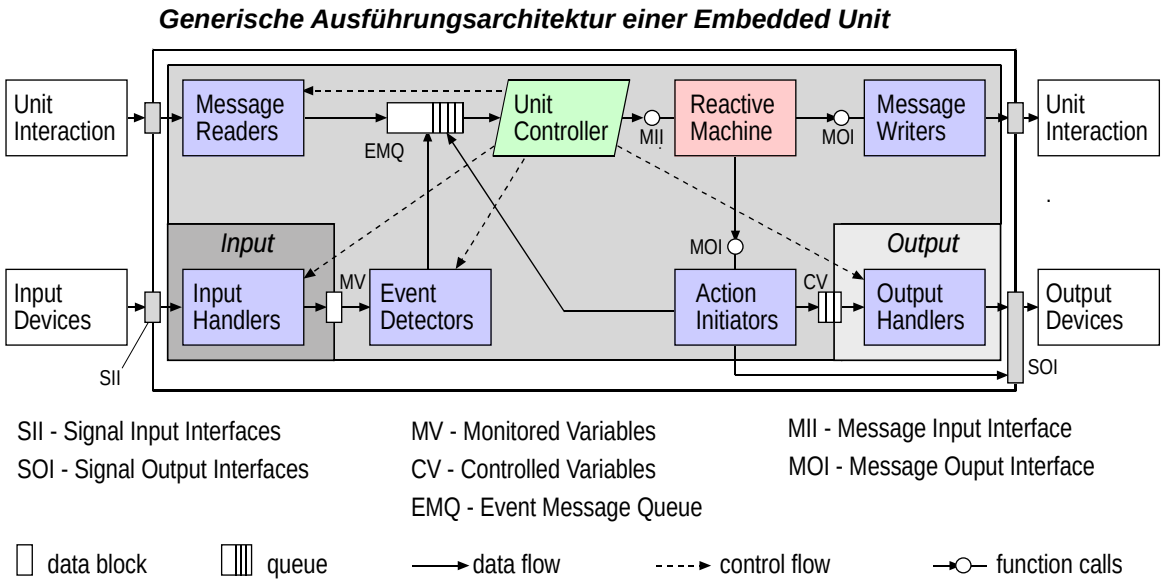


Abb. 62: Komponenten und Datenpfade einer Embedded Unit

Der *Unit Controller* implementiert die Ablaufsteuerung, was durch die entsprechenden Kontrollflüsse dargestellt ist. Auszuführen ist die *Reactive Machine*, indem sie für detektierte Events mit entspechenden Event Messages getriggert wird. Dazu kommen die *Interaktionsfunktionen*, die im DEC-Modell im *Process Link Layer* und *Signal Link Layer* definiert worden sind. Für die Ausführung werden diese elementaren Interaktionsfunktionen als sequentielle *Compound Statements* in den entsprechenden Prozeduren *Input Handlers, Event Detectors, Action Initiatiors* und *Output Handlers* eingebunden.

Falls mehrere Embedded Units im Spiel sind, kommen noch die Prozeduren *Message Readers* und *Message Writers* für die Kommunikation zwischen Embedded Units dazu. Auf diese Interaktions-

funktionen gehen wir hier nicht näher ein. Im DEC-Modell werden diese im Zusammenhang mit spezifizierten Communication Channels erstellt.

Die dargestellte Ausführungsarchitektur ist durch das Konzept der *zyklusbasierten* Unit Ausführung begründet. Eine Embedded Unit wird in periodischen Zyklen ausgeführt, und für jeden Zyklus wird eine bestimmte Ausführungszeit zur Verfügung gestellt. Das Konzept ermöglicht, sowohl auf der Prozessor-Ebene wie auch auf der Unit-Ebene ein Scheduling festzulegen, bei dem verifiziert werden kann, ob die zeitlichen Anforderungen auch im Worst Case erfüllt werden können (zweistufige Scheduling-Hierarchie).

Die periodischen Unit-Zyklen eignen sich gut, um die sich wiederholende Aufgaben mit festen Ausführungszeiten wie das Einlesen von Inputdaten, die Detektion der Ereignisse und das erzeugen des Outputs statisch planbar auszuführen. Das gilt auch für die Verarbeitung der periodischen Ereignisse. Die statisch planbare Ausführung ist jedoch bei sporadisch auftretenden Ereignisse nicht mehr möglich. Das Auftreten solcher Ereignisse ist nicht voraussehbar. Die Verarbeitung sporadischer Ereignisse muss deshalb dynamisch geplant werden (dynamic scheduling). Der Zyklustakt bestimmt damit die Verarbeitung der periodischen Aktivitäten; die dynamisch zu planende Verarbeitung der sporadischen Ereignisse muss hingegen auf die freie Zeit in den periodischen Zyklen verteilt werden können.

Die Ereignisverarbeitungen müssen gegenseitig non-preemptiv ausgeführt werden. Das verlangt die Run-to-Completion Semantik der Reaktiven Maschine. In der Regel werden alle Komponenten einer Embedded Unit non-preemptive durch den *Unit Controller* prozedural gesteuert. Solche Abläufe sind deterministisch und für Interaktionsfunktionen, die meistens eine feste Ausführungszeit haben, einfach machbar.

Ausführungskonzept - Zyklus-basierte Ausführung einer Embedded Unit

Die Ausführung einer Embedded Unit erfolgt in periodischen Zyklen immer nach demselben *Input-Detection-Reaction-Output* Muster:

Input Handler; *Event Detector*; (*Message Reader,)*
Reactive Machine; . . . ; *Reactive Machine*;
Output Handler;

Der Unit-Input wird in der Input-Phase am Anfang eines Zyklus eingelesen. Die anschliessende Detektionsphase dient dem Detektieren neuer sporadischer Ereignisse. Dabei werden für detektierte Ereignisse entsprechende Ereignismeldungen in der *Event Message Queue* zwischengespeichert. Die Organisation der *EMQ* hängt von den zeitlichen Anforderungen ab (FiFo, Ereignisprioritäten, EDF).

In der anschliessenden Reaktionsphase werden zuerst die periodischen Ereignisse verarbeitet und anschliessend sporadische. Wie oft die *Reactive Machine* in einem Zyklus für eine Ereignisverarbeitung aufgerufen werden kann ist dynamisch bestimmt, weil die Ausführungszeiten oft stark vom Systemzustand abhängen. Die Bedingung für eine weitere mögliche Ereignisverarbeitung hängt immer davon ab, wieviel von der zur Verfügung gestellten Prozessorzeit noch übrig ist. Mögliche Varianten der Unit-Ausführung werden im folgenden Unterkapitel im Detail diskutiert. Die *Action Initiators* werden für jede *Action Message* direkt durch die *Reactive Machine* aufgerufen. Unter Umständen können die Steuerdaten direkt an das entsprechende Peripheriegerät ausgegeben werden, in dem man die entsprechenden *Output Handler* Funktionen in den *Action Initiators* integriert (bypassing).

Nach Abschluss der Reaktionsphase folgt in der Output-Phase die Ausgabe des von den Ereignisverarbeitungen initiierten Output.

Spezialfall. Falls sicher ist, dass in jedem Zyklus alle detektierten Ereignisse verarbeitet werden können, ist keine *Event Message Queue* notwendig. Der *Event Detector* kann dann für jedes detektierte Ereignis selber die Reaktive Maschine triggern (siehe *Event Detector ohne Event Message Queue* am Schluss diese Unterkapitels).

Verfeinerung der generischen Ausführungsarchitektur. Die in *Abbildung 62* dargestellte Ausführungsarchitektur zeigt generisch, wie die Verbindungssoftware einer Embedded Unit in Prozeduren oder Objekten organisiert wird. In einer Anwendung kann diese Organisation verfeinert werden. Zum Beispiel kann ein "langsamer" *Input Handler*, der nur alle 4 Zyklen aufgerufen wird, oder ein "schneller" *Input Handler,* den eine Interrupt-Routine aktiviert. Auch bei den Event Detektoren können Verfeinerungen sinnvoll sein. Zum Beispiel mit 3 Event Detektoren, die 3 verschiedene Event Message Queues bedienen, können relativ einfach 3 statische Ereignis-Prioritäten unterstützt werden.

Harte Echtzeitbedingungen. Für die Verarbeitung harter sporadischer Ereignisse sind unterschiedliche *Deadlines* (Zeitschranken) einzuhalten. Welche Ereignismeldung dann jeweils zur Verarbeitung kommt, ist durch die Deadlines der noch nicht verarbeiteten Ereignismeldungen bestimmt. Der optimale Algorithmus für solche Aufgaben ist EDF Scheduling (Earliest Deadline First), d. h. es wird immer die Ereignismeldung mit der kleinsten absoluten Deadline verarbeitet (siehe Unterkapitel *Deadline Scheduling der Ereignisverarbeitung*). Die Ablaufsteuerung einer *Embedded Unit* wird deshalb als einfache Kombination von *periodisch-zyklischem* und *EDF*-Scheduling organisiert.

Die Ausführung in periodischen Zyklen wird wichtig, wenn mehrere Embedded Units auf demselben Prozessor ausgeführt werden. Für die lokale Ausführungssteuerung einer Embedded Unit muss bekannt sein, welcher Bruchteil der Prozessorbenutzungszeit der Unit zusteht, und zwar muss definiert sein, innerhalb welcher Zeitspanne diese Benutzungszeit garantiert ist. Es ist klar, dass die lokale Ausführungssteuerung am einfachsten zu realisieren ist, wenn die Prozessor-Benutzungszeit pro Zyklusperiode garantiert ist.

Bemerkungen zur Wahl der Zyklusperiode

Maximale Ausführungszeiten. Das zyklusbasierte non-preemptive Scheduling der Ereignisverarbeitungen kann nur funktionieren, wenn die Zyklusperiode nicht zu klein ist. Sie sollte *grösser* sein als die maximale Ausführungszeit einer einzelnen Ereignisverarbeitung, damit der Input-Handler in jedem Zyklus aktiviert wird. In der Regel ist die Zyklusperiode ein Vielfaches der maximalen Ereignisverarbeitungszeit, und die erwähnte Bedingung ist kein Problem.

Minimale Deadlines. Falls harte Zeitschranken (Deadlines) für Antwortzeiten bestehen, muss in der Regel die Zyklusperiode *kleiner* als die vorgegebenen Deadline-Intervalle gewählt werden. Weil nur am Anfang eines Zyklus neue Inputdaten eingelesen werden, muss für jedes aufgetretene Ereignis im schlimmsten Fall mit einer Detektionslatenzzeit von der Grösse der Zyklusperiode gerechnet werden. Die Zyklusperiode muss damit kleiner sein als sämtliche relativen Zeitschranken für die Ereignisverarbeitung. Solche Anforderungen sind normalerweise kein Problem, weil Antwort-Zeitschranken bezüglich einer Zeitskala definiert werden, in der auch die physikalischen Vorgänge des betroffenen Prozesses beschrieben werden. Diese Zeiten sind in der Regel ein Vielfaches der in Frage kommenden Zyklusperioden.

Konfliktfall. Es kann vorkommen, dass in einem System die Ausführungszeiten gewisser Ereignisverarbeitungen grösser sind als die kleinsten Deadline-Intervalle, d. h. es ist nicht möglich eine Zyklusperiode finden, die grösser als die grösste Ausführungszeit und kleiner als das kleinste Deadline-Intervall ist. In diesem Fall muss die *Embedded Unit* in zwei, eventuell sogar drei Units aufgeteilt werden, so dass für jede Unit eine brauchbare Zyklusperiode definiert werden kann.

Im Folgenden geht es darum, Wege zur Umsetzung des zyklusbasierten Ausführung einer Embedded Unit aufzuzeigen. Dabei muss garantiert werden, dass der Zyklustakt eingehalten wird. Bei den diskutierten Implementierungsvarianten nehmen wir an, dass die betrachtete *Embedded Unit* als einzige Parallelitätseinheit auf dem Rechner läuft. Entstehende Wartezeiten, wenn in einem Zyklus keine Ereignismeldungen zur Verarbeitung anstehen, können in diesem Fall ohne Betriebssystemhilfe einfach als Leerlaufschlaufen (busy wait) implementiert werden. Wenn während Wartezeiten andere Arbeiten ausgeführt werden müssen, ist die einfache 'busy wait'-Technik jedoch nicht mehr anwendbar. Die Lösung solcher Probleme wird im Zusammenhang mit dem Scheduling mehrerer Embedded Units im letzten Teil dieses Kapitels behandelt.

6.1.3 Streng-periodische Ausführung einer Unit

Bei der *streng-periodischen* Implementierung wird garantiert, das der Input Handler immer am Anfang jedes Zyklus aufgerufen wird. Zyklen mit garantierter Einhaltung des Taktbeginns sind notwendig, wenn für die periodisch zu verarbeitenden Ereignisse von Reglern der *Input-Jitter* (Schwankung der Inputzeiten) stark beschränkt sein muss, d. h. der *Input Handler* muss immer genau am Anfang eines neuen Unit-Zyklus aktiviert werden.

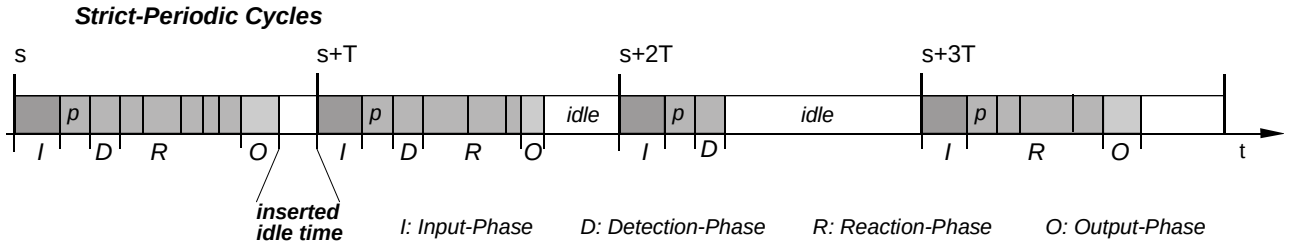


Abb. 63: Periodische Zyklusausführung

Abbildung 63 zeigt die Phasen streng-periodischer Unit-Zyklen. In diesem Beispiel ist in jedem Unit-Zyklus ein periodisches Abtast-Ereignis (*p*) eines Reglers zu verarbeiten. In der Input-Phase (*I*) sind die *Input Handlers* aktiv. Diese Phase dauert immer gleich lang. Anschliessend folgt die periodische Ereignisverarbeitung (*p*), d. h. die Reaktive Maschine wird genau einmal aufgerufen (statisches Scheduling). Danach folgt die Detektionsphase (*D*) durch Aktivierung der *Event Detectors.* Die Ausführungszeit dieser Phase weist geringe, aber beschränkte Schwankungen auf, die nicht ins Gewicht fallen. In der anschliessenden Reaktionsphase (*R*) folgt das dynamische Scheduling der Ereignisverarbeitungen für detektierte sporadische Ereignisse. Die Verarbeitung der anstehenden Ereignismeldungen wird abgebrochen, sobald die verbleibende Zykluszeit (*Slack Time*) nicht mehr reicht, um das nächste Ereignis zu verarbeiten und die Output-Phase auszuführen, oder wenn die *Event Message Queue* leer geworden ist. Die Dauer der Output-Phase (*O*) ist durch die erfolgte Reaktionsphase bestimmt. Die Ausführung kann aber in der aktuellen Reaktionsphase statisch voraus bestimmt werden, weil die Outputs bekannt und die Ausführungszeiten der einzelnen Output-Funktionen immer gleich sind (quasi-statisches Scheduling).

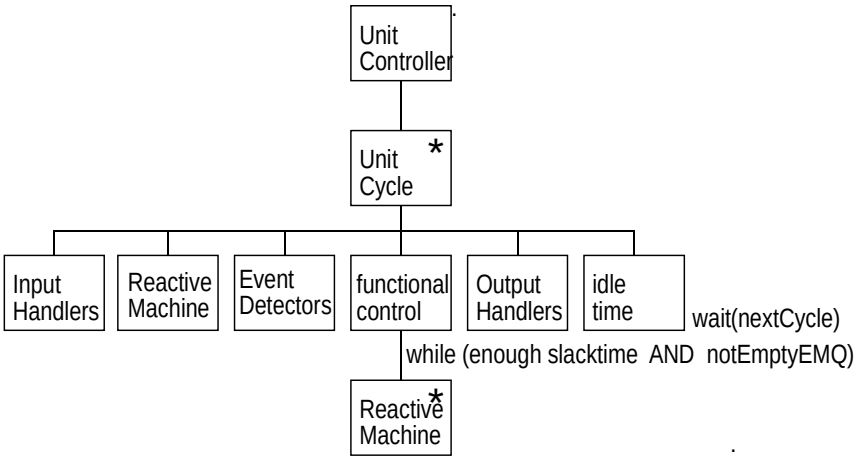


Abb. 64: Struktogramm der Programmstruktur einer streng-periodischen Unit-Task

Die Programmstruktur des ständig aktiven *Unit Controllers* ist in *Abbildung 64* als Struktogramm dargestellt. In jedem Zyklus werden zuerst die *Input Handlers* ausgeführt, gefolgt von einer ersten Aktivierung der *Reactive Machine* für die periodische Ereignisverarbeitung. Nach anschliessender Aktivierung der *Event Detectors* folgt die iterative Aktivierung der *Reactive Machine* für detektierte sporadische Ereignisse. Bei der Berechnung der *Slack Time* wird die Zeit, die für das Schreiben des anstehenden Outputs notwendig ist, berücksichtigt. Ein Zyklus wird nach der Ausführung der *Output*

Handlers mit einer Leerlauf-Phase (busy wait) abgeschlossen. Sie dauert bis es Zeit ist, den neuen Zyklus zu starten.

Diese Art von non-preemptiven Scheduling wird als *Scheduling with inserted idle times* bezeichnet, weil in jedem Unit-Zyklus eine Restzeit (*idle time*) nicht genutzt werden kann. Das heisst nicht, dass der Prozessor in dieser Zeit nichts anderes tun kann. *Idle* bezieht sich hier auf das *Event-Processing*. Diese Idle-Zeit kann immer für unterbrechbare Arbeiten (preemptable tasks) genutzt werden, z. B. von einer langsameren unterbrechbaren *Embedded Unit*.

In der Abbruchbedingung wird jedesmal die *WCET* (*Worst Case Execution Time*) der anstehenden Verarbeitung verwendet. Diese WCET-Werte müssen entweder berechnet werden (durch Addition der Ausführungszeiten der ausgeführten Instruktionen) oder durch Messung des “Worst Cases”. Für CIP-Modelle kann z. B. Code generiert werden, der bei der Ausführung Messwerte liefert, aus denen im CIP Tool anschliessend diese WCET-Werte für alle Ereignisse automatisiert ermittelt werden.

Dieses *Scheduling with inserted idle times* ist etwas aufwendig in der Realisierung, weil ständig die *Slack Time* berechnet werden muss, was vor allem auch bei Änderungen und Erweiterungen der Software eine Erschwernis ist.

6.1.4 Quasi-periodische Ausführung einer Unit

Bei der *quasi-periodischen* Ausführung gibt man zeitlich den Zyklusgrenzen einen gewissen Spielraum, so dass der Takt aber noch eingehalten wird (es geht kein Takt verloren). Diese einfachere Implementierung der Ausführungssteuerung kommt ohne eingefügte Leerlaufzeiten aus, hat aber keine Kontrolle über die genauen Inputzeiten, was bei periodischen Verarbeitungen ein Problem sein kann. Der grosse Vorteil dieser Ausführungsart ist, dass die WCET’s für die einzelnen Ereignisverarbeitungen nicht bekannt sein müssen. Es ist lediglich sicher zu stellen, dass der maximale WCET etwas kleiner als die Taktperiode T ist.

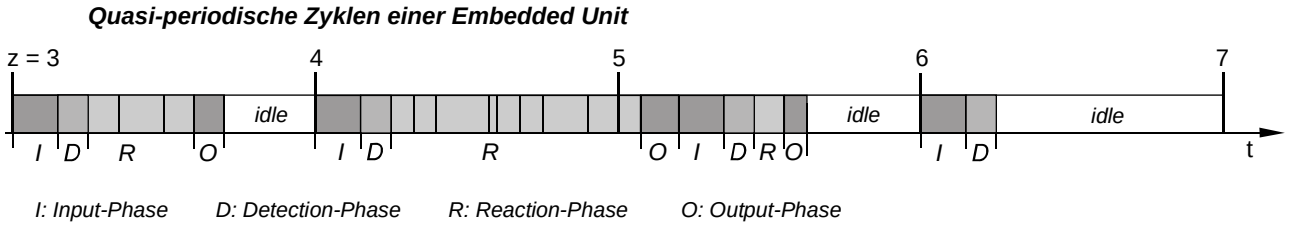


Abb. 65: Quasi-periodische Zyklusausführung

Abbildung 65 zeigt einige Zyklen einer quasi-periodischen Ausführung. Die letzte Verarbeitung eines sporadischen Ereignisses eines Zyklus und die anschliessende Output-Phase dürfen am Anfang des nächsten Taktes abgeschlossen werden. Das ist immer der Fall, wenn in einem Takt nicht alle anstehenden Ereignismeldungen ausgeführt werden können, wie z. B. im zweiten Takt des Ablaufszenarios in *Abbildung 65*.

Damit diese Implementierung mit schwankender Zyklusgrösse im Takt bleibt, muss sichergestellt sein, dass bei jeder überlappenden Ereignisverarbeitung im neuen Takt noch genügend Zeit verbleibt. Die Zeit muss reichen für die Ausführung der *Input-Phase*, der *Detektion-Phase*, der *Output-Phase*, Verarbeitung eines sporadischen Ereignisses und unter Umständen auch für die Verarbeitung periodischer Ereignisse. Eine entsprechende Bedingung kann rein statisch formuliert werden, d. h. aufgrund von Unit-Maximal-Werten, wie z. B.

$$T_{max\_input} + T_{max\_detection} + (T_{max\_periodic\_event}) + T_{max\_sporadic\_event} + T_{max\_output} < T$$

Dabei sind T_{max_input}, $T_{max\_detection}$ und T_{max_output} die Maximalwerte der entsprechenden Zyklusphasen, und $T_{max\_periodic\_event}$ und $T_{max\_sporadic\_event}$ maximale WCET-Werte für Ereignisverarbeitungen.

Oft genügt auch eine vereinfachte Bedingung, wie zum Beispiel $T_{max\_sporadic\_event} < T/2$.

In *Abbildung 66* ist die Programmstruktur der Unit-Task und des interrupt-getriebenen Zyklus-Timers dargestellt. Der *Unit Controller* steuert wie im streng-periodischen Fall die verschiedenen Phasen eines Zyklus. Die Phase *possible busy wait* ist entweder Null, oder sie dauert bis der neue physikalische Takt beginnt. Es werden in diesem Beispiel keine periodischen Ereignisse verarbeitet.

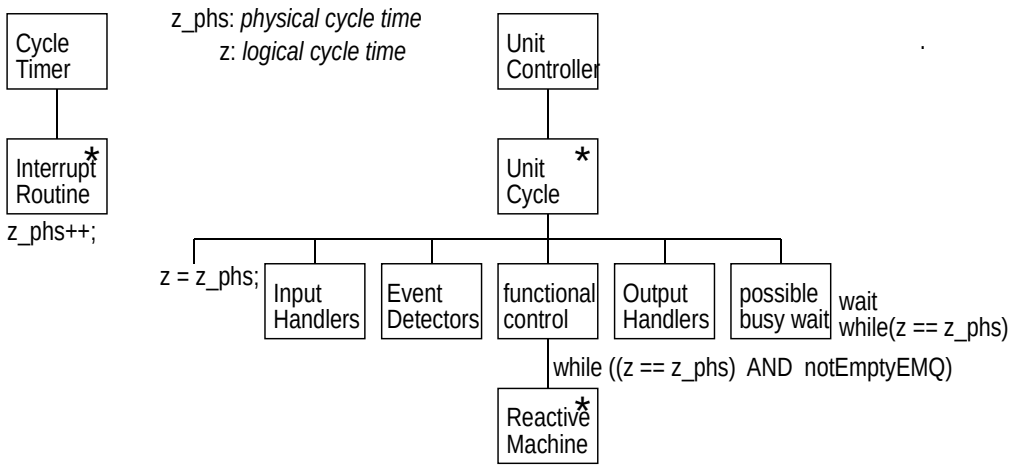


Abb. 66: Struktogramm der Programmstruktur einer quasi-periodischen Unit-Task

Der Verbleib in der iterativen Ereignisverarbeitung im *Unit Controller* ist durch folgende einfache Bedingung bestimmt:

(z == z_phs) AND *notEmptyEMQ*

z_phs ist der *physikalische* Zykluszähler des Timers (*physikalische* diskrete Zykluszeit)
z ist der Zykluszähler im *Unit Controller* (nachgeführte *logische* diskrete Zykluszeit).
*notE*mptyEMQ ist wahr, wenn die Event Message Queue nicht leer ist.

Die Aktualisierung der logischen Zykluszeit z am Anfang eines neuen Zyklus ist robust, denn sie funktioniert auch, wenn eine Ereignisverarbeitung länger als eine Zyklusperiode dauert, und auch der Integer-Overflow von z (wrap around bei 2^16) muss nicht gesondert behandelt werden.

Bei der Überwachung der Deadlines können die Schwankungen der Zyklusgrenzen gut toleriert werden. Wie wir im nächsten Kapitel (Scheduling) sehen werden, können die Deadline-Intervalle in Anzahl Zyklen spezifiziert werden. Die Abweichungen liegen somit im Bereich der zeitlichen Auflösung.

6.1.5 Periodisch/quasi-periodische Zyklus-Ausführung

Schliesslich können die beiden beschriebenen Ausführungskonzepte elegant kombiniert werden, nämlich streng-periodische Input-Phasen kombiniert mit quasi-periodischen Detection-Reaction-Output-Phasen. Auch diese Lösung kommt ohne eingefügte Leerlaufzeiten aus, hat aber die volle Kontrolle über die Inputzeiten.

Für die Implementierung sind zwei Ausführungseinheiten notwendig. Eine getaktete, d. h. streng-periodische *Input-Task*, welche wiederholt die Input-Phase ausführt, indem sie bei jedem Takt den *Unit Controller* unterbricht und den *Input Handler* triggert. Die restlichen Arbeiten werden durch den *Unit Controller* gesteuert, der in einer Task mit tieferer Priorität ausgeführt wird.

Eine einfache Implementation ohne RTOS ist möglich, wenn nur Arbeiten der Embedded Unit zu erledigen sind. Die Input-Task wird als Interrupt-Subroutine des Timers implementiert, der die physikalische Zykluszeit inkrementiert, und der *Unit Controller* als einzig laufendes Hauptprogramm.

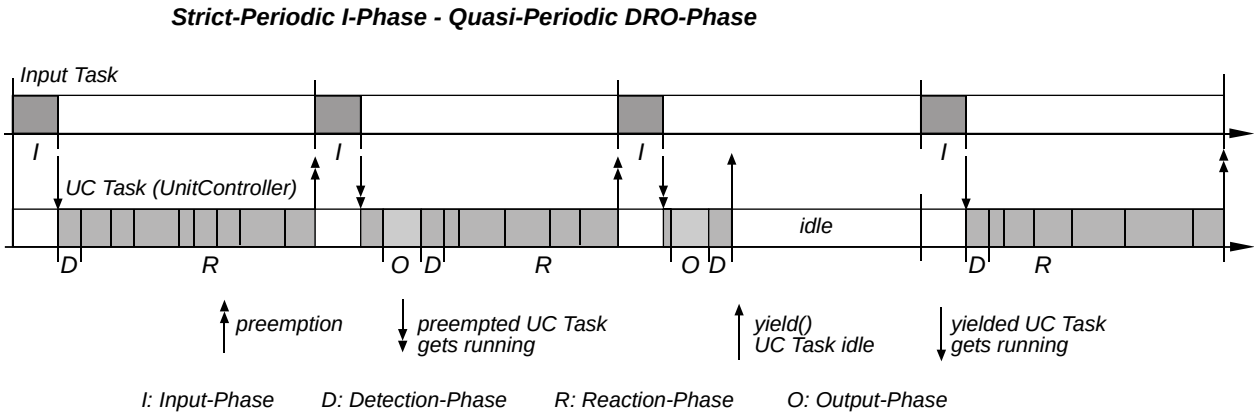


Abb. 67: Streng-periodische Input-Phasen, quasi-periodische Detection-Reaction-Output-Phasen

Die *Abbildung 67* zeigt ein Stück der Ausführung der *Input Task* und der *Unit Controller Task*. Die *Input Task* wird streng-periodisch ausgeführt. Ihr Zyklus beginnt immer, wenn ein neuer Takt beginnt, d. h. wenn die physikalische Zykluszeit z_phs ihren Wert geändert hat.

Ein neuer Unit-Zyklus beginnt mit der Detektion neuer Ereignisse. Falls die *Unit Controller Task* nicht alle anstehenden Ereignismeldungen abgearbeitet hat, wird im folgenden Takt die Ereignisdetektion verzögert ausgeführt, wie z. B. im zweiten Takt des Ablaufszenarios von *Abbildung 67*.

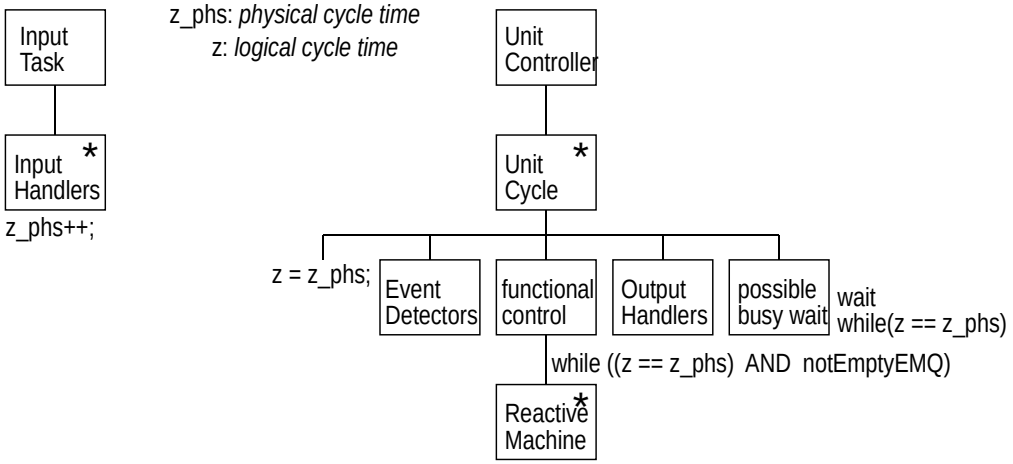


Abb. 68: Struktogramm der Input und der Unit Controller Task

In *Abbildung 68* ist die Programmstruktur der *Input Task* und der *Unit Controller Task* dargestellt.

Der *Unit Controller* steuert wie im streng- oder quasi-periodischen Entwurf die ganze Reaktionsphase. Der Warte-Phase *possible busy wait* ist wieder entweder Null, oder sie dauert bis der neue Takt beginnt.

Die Implementation mit der asynchronen *Input Task* funktioniert nur, weil der *Event Detector* am Anfang des Unit-Zyklus aktiviert wird. Damit ist gesichert, dass er nicht durch den *Input Handler* unterbrochen wird. Das ist wichtig, weil der *Event Detector* Variablen liest, die vom *Input Handler* beschrieben werden.

Kombination verschiedener Input Handler

In derselben Unit können verschiedene *Input Handler* kombiniert werden. In *Abbildung 69* ist eine entsprechende Ausführungsarchitektur einer Embedded Unit dargestellt. In dieser Darstellung sind auch die Kontrollflüsse eingezeichnet (gestrichelt).

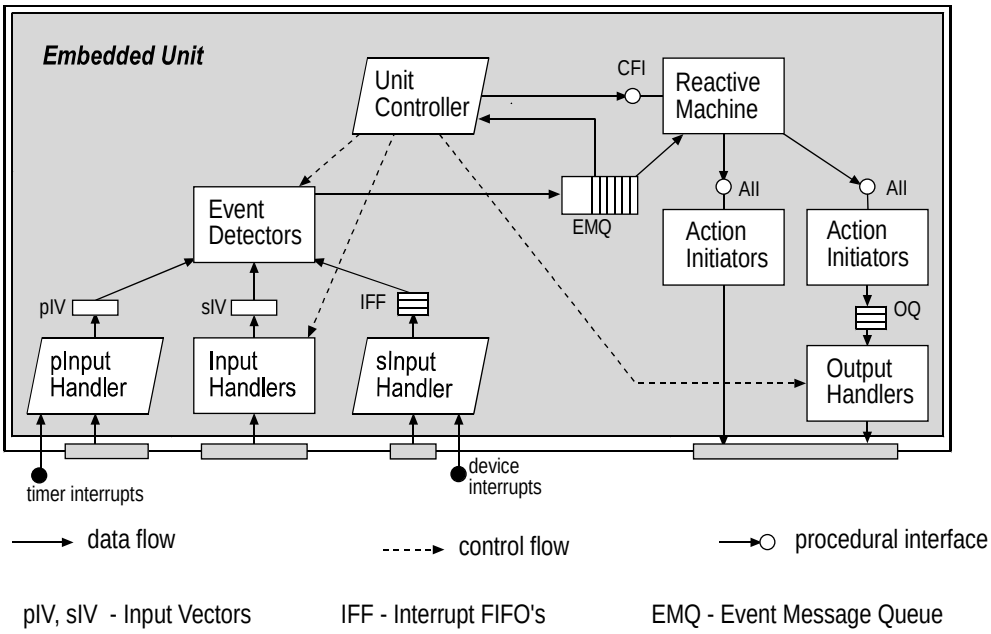


Abb. 69: Unit Architektur mit periodischem, quasi-periodischen und sporadischem Input Handler

Neben dem *Unit Controller* sind auch der periodische *pInput Handler* und der sporadische *sInput Handler* Komponenten mit eigenem Control Thread (aktive Komponenten). Der *pInput Handler* wird durch den Zyklus-timer getriggert, und der *sInput Handler* stellt Inputzugriffe dar, die in Interrupt-Routinen ausgeführt werden. Das ist zum Beispiel notwendig, wenn verteilt installierte Sensoren die Information für aufgetretene Ereignisse über einen Feldbus übertragen. In den Interrupt-Routinen wird jedoch nur der neue Input in einer geeigneten Datenstruktur abgelegt (Interrupt FIFO). Die Ereignisdetektion erfolgt jedoch wie für die anderen Inputs erst am Anfang des neuen Unit-Zyklus.

Es sind aber auch Lösungen möglich, bei welchen Ereignisse bereits während der Interrupt-Verarbeitung detektiert und in eine mit Semaphoren geschützte EMQ geschrieben werden. Diese EMQ wird dann mit höchster Priorität abgearbeitet. Damit wird es möglich, auch während des aktuellen Zyklus neu aufgetretene Ereignisse zu detektieren und mit höchster Priorität zu verarbeiteten. Die maximale Latenzzeit für die Verarbeitung ist so nicht mehr die Zyklusperiode T (Polling Latenz), sondern nur noch die maximale Ausführungszeit der Ereignismeldungen (Run-to-Completion Semantik der Reaktiven Maschine).

6.1.6 Event Detector ohne Event Message Queue (Optimierung)

Ein anderer Ansatz (Optimierung bei knappen Ressourcen) [2] besteht darin, nur so viele Ereignisse zu detektieren, wie gerade verarbeitet werden können, unter Umständen kann das auch nur gerade *eines* sein. Mit dieser Variante müssen keine Ereignismeldungen in einer EMQ zwischengespeichert werden, weil für jedes detektierte Ereignis sofort die Reaktive Maschine getriggert wird. Nicht detektierte Ereignisse werden in einem der folgenden Zyklen detektiert und verarbeitet. Das funktioniert vor allem bei sporadischen Ereignissen gut, weil damit gerechnet werden kann, dass sich der Zustand des betreffenden Sensors während einer gewissen Zeit nicht ändert.

Die Detektion und Verarbeitung aufgetretener Ereignisse erfolgt in jedem Zyklus von Neuem nach einer statisch festgelegten Prioritätenliste, solange bis die *Slack Time* genügend gross ist (streng-peri-

odische Ausführung), oder bis sich die physikalische Zykluszeit erhöht hat (quasi-periodische Ausführung). Werden die relativen Deadlines von *Event Messages* (maximale Wartezeiten) als statische Prioritäten verwendet, entspricht das dem *Deadline Monotonic Scheduling* Verfahren (*DMS*). Die kleinste Deadline hat dabei die höchste Priorität. Im Unterschied zum dynamischen *Earliest Deadline First* Scheduling (EDF) muss aber damit gerechnet werden, dass bei Vollauslastung die *Event Messages* mit grosser relativer Deadline zu spät verarbeitet werden (sog. Verhungern bei statischen Prioritäten).

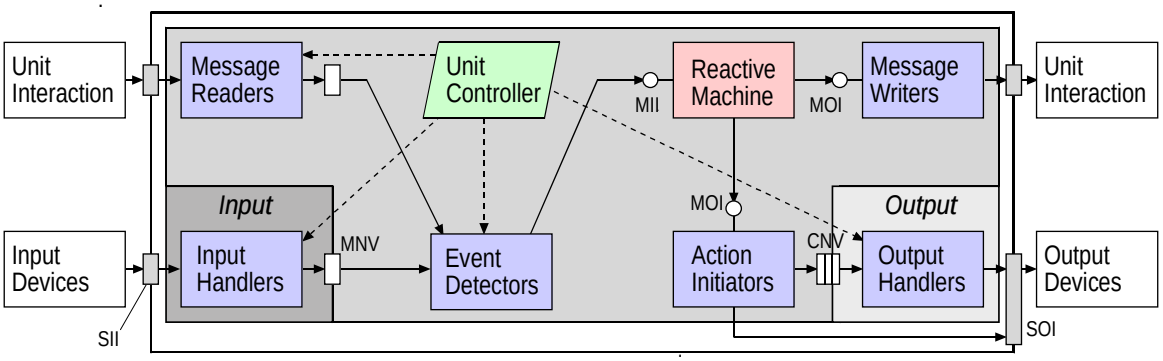


Abb. 70: Optimierte Ausführungsarchitektur

Neben Deadline-Intervallen können hier aber auch Kriterien, welche die Betriebssicherheit (Safety) betreffen, bei der Ereignis-Detektion einen Rolle spielen. Bei Ereignislawinen, wie sie am häufigsten bei systemübergreifenden Pannen auftreten, kann eine solche Variante Vorteile haben, weil diese Situationen gerade diejenigen sind, wo die vorhandene Leistung des Systems dringend zur Behebung des Problems gebraucht wird.

Ablaufsteuerung des Unit Controller

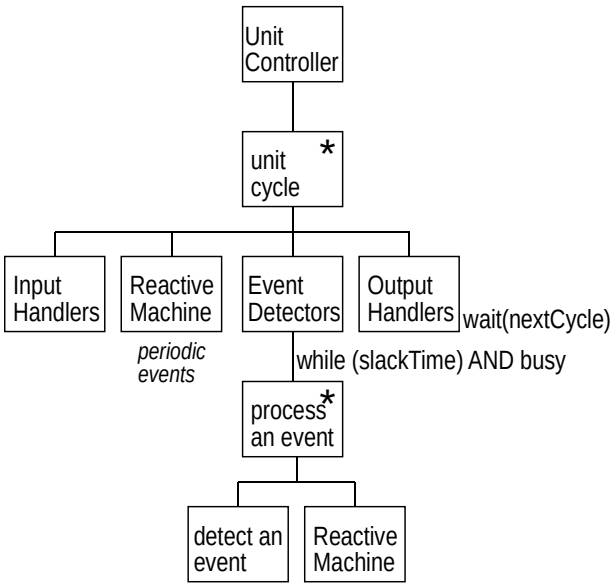


Abb. 71: Struktogramm des Event Detector

Im Unterschied zur Ablaufsteuerung mit dem *Unit Controller* basiert das Deadline-Scheduling des *Event Detector* wie gesagt auf statischen Ereignis-Prioritäten und ist damit weniger optimal als das dynamische EDF-Verfahren. Wie weit das in der industriellen Praxis relevant ist, hängt sicher vom einzelnen System ab. Was mehr ins Gewicht fallen kann, ist dass ein potenzieller Overload vom *Event Detector* nicht frühzeitig erkannt werden kann. Geeignete Massnahmen sind damit erst beim Eintreffen der “Katastrophe” möglich.

6.2 Deadline-Scheduling der Ereignisverarbeitung

Die zeitlich sequentielle Ausführung der Ereignisverarbeitung einer *Embedded Unit* entspricht dem Scheduling non-preemptiver Tasks eines Informationsverarbeitungssystems. Der Zeitpunkt der *Detektion* eines aufgetretenen Ereignisses (Ereignisinstanz eines Ereignistyps) hat in diesem Sinn die gleiche Bedeutung wie die *Ankunft* eines Jobs (*task gets ready*), und die Ausführung der entsprechenden Ereignismeldung entspricht der Ausführung des bereitstehenden Jobs. Wir gehen davon aus, dass alle Ereignisse sporadisch oder periodisch sind, d. h. zwischen zwei Instanzen desselben Ereignistyps vergeht immer einen minimale Zeitspanne (*minimal period*).

6.2.1 Deadline-Intervalle für die Ereignisverarbeitung

Bemerkung zum Begriff Deadline. In der Scheduling-Literatur ist es üblich, für Tasks mit harten Echtzeitbedingungen sogenannte *relative Deadlines* als Scheduling-Parameter festzulegen. Die relative Deadline eines Job spezifiziert, innerhalb welcher Zeitspanne der angekommene Job ausgeführt werden muss. Mit absoluter Deadline wird der Zeitpunkt bezeichnet, bis zu dem der Job abgearbeitet sein muss, d. h.

absolute Deadline = Ankunftszeit + *relative Deadline*

Die Bezeichnung *relative Deadline* ist etwas irreführend, denn sie legt nicht einen Zeitpunkt fest, sondern sie spezifiziert ein Zeitintervall, nämlich das Intervall zwischen dem Zeitpunkt der Task-Ankunft und der oberen Zeitschranke für ihre Ausführung. Anstatt *relative Deadline* findet man deshalb in der neueren Literatur auch die klarere Bezeichnung *Deadline-Intervall*, die wir im folgenden auch verwenden werden.

Ereignis-Deadline-Intervall. Ein *Ereignis-Deadline-Intervall* ist ein spezifiziertes Attribut eines Ereignis-Typs, das eine obere Grenzen für die Verarbeitungszeit der entsprechenden Ereignisinstanzen (Jobs) auf dem Rechner festgelegt. Ereignisse mit spezifizierten *Deadline-Intervallen* bezeichnen wir als *harte* Ereignisse, d. h. die mit *Deadline-Intervallen* ausgedrückten Zeitbedingungen sind *harte* Zeitbedingungen. Das sind Zeitbedingungen, die unbedingt eingehalten werden müssen. Ereignisse ohne spezifiziertes Deadline-Intervall sind sog. *weiche* Ereignisse, für welche hier keine Zeitbedingungen formuliert werden.

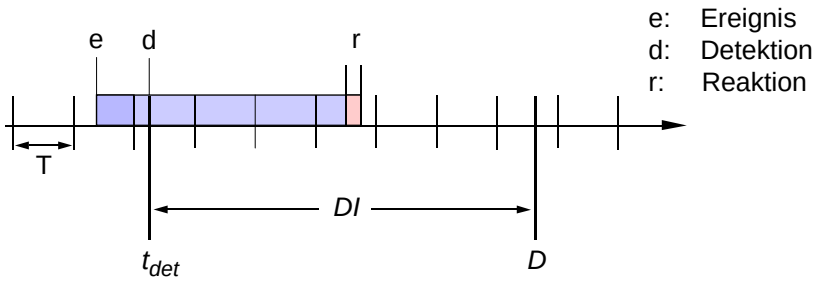


Abb. 72: Deadline-Intervalle

In *Abbildung 72* ist das zeitliche Szenario von der Verarbeitung eines harten Ereignisses dargestellt. T ist die Periode der zyklischen Unit-Ausführung. Zum Zeitpunkt e tritt das Ereignis in der Rechnerumgebung auf, d markiert den Zeitpunkt der Ereignisdetektion auf dem Rechner (*Event Detector*) und r bezeichnet den Zeitbereich der Ereignisverarbeitung (*Reactive Machine*). DI ist das Deadline-Intervall, innerhalb welchem die Ereignisverarbeitung abgeschlossen werden muss.

Die *Deadline-Intervalle* für die Ereignisverarbeitung müssen aus den zeitlichen Anforderungen an das System (Antwortzeiten) und bestimmten Systemeigenschaften ermittelt werden. Bei den Systemeigenschaften spielen Übertragungszeiten von Sensordaten, Latenzzeiten der Ereignisdetektion und unter Umständen Separationszeiten zu sequentiell abhängigen Nachfolge-Ereignissen eine Rolle.

6.2.2 Zyklusbasierte Deadline-Intervalle

Bei der Herleitung der Ereignis-Deadline-Intervalle aus den Zeitanforderungen besteht meistens ein Spielraum für die ermittelten Werte, und auch die Genauigkeit dieser Werte beziehen sich fast immer auf eine Zeitskala, in der auch die physikalischen Vorgänge der betroffenen Prozesse beschrieben werden. Die Zeitskala, in der Prozessor-Ausführungszeiten angegeben werden, ist hingegen wesentlich feiner. Diese Zeitskala ist relevant für die Überwachung der Zyklusperioden, für die Überwachung der Deadlines hingegen genügt eine gröbere Zeitauflösung. Diese Tatsache kann ausgenützt werden, um die Spezifikation, die Verifikation und die Implementation des non-preemptiven Deadline-Scheduling stark zu vereinfachen. Das zyklus-basierte Deadline-Scheduling wird nämlich entscheidend weniger komplex, wenn Deadline-Intervalle nicht bezüglich der Ausführungszeit definiert werden, sondern indem einfach festgelegt wird, innerhalb wieviel Zyklen ein detektiertes Ereignis zu verarbeiten ist.

> ***Schedulingkonzept - Zyklusbasierte Deadline-Intervalle.*** Deadline-Intervalle sporadischer Ereignisse beziehen sich in der Regel auf eine Zeitskala, die um ein Vielfaches gröber ist als die Zeitskala für Ausführungszeiten. Deadline-Intervalle werden deshalb durch eine *Anzahl Zyklen* spezifiziert, d. h. die Reaktive Maschine muss dann für ein detektiertes Ereignis innerhalb dieser Anzahl Zyklen getriggert werden, den Detektionszyklus eingeschlossen.

Mit dem Konzept zyklusbasierter Deadline-Intervalle wird für die Überwachung der Deadline-Intervalle der Zykluszähler z als diskrete Zeit verwendet:

z = t/T (Ganzzahldivision) mit t = Prozessorzeit und T = Zyklusperiode

z numeriert die einzelnen Zyklen der Embedded Unit und wird als diskrete Zykluszeit bezeichnet.

In *Abbildung 73* ist das zeitliche Szenario einer Ereignisverarbeitung in Bezug auf die diskrete Zykluszeit z dargestellt. Das Deadline-Intervall des detektierten Ereignisses ist mit DI = 6 spezifiziert. Die Detektion findet zur Zeit z = 4 statt (d. h. detektiert im Zyklus 4), und die Verarbeitung zur Zeit z = 7 (im Zyklus 7).

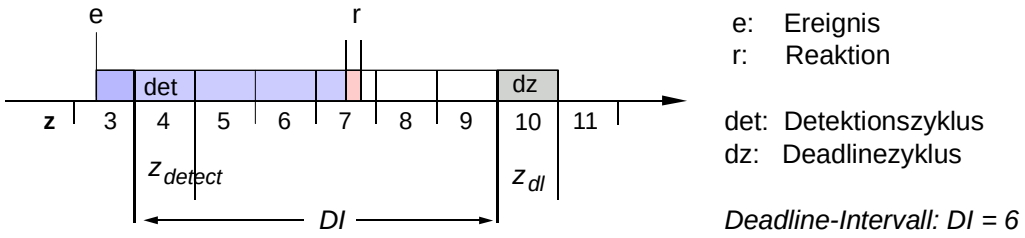


Abb. 73: Zyklusbasiertes Deadline-Intervall

Zyklus 10 ist in diesem Beispiel der sogenannte *Deadline-Zyklus*, d. h. das Ereignis muss vor diesem Zyklus verarbeitet werden. Der *Deadline-Zyklus dz* definiert damit die absolute Deadline für die Ereignisverarbeitung, ausgedrückt in der diskreten Zykluszeit z.

Zykluszeit des Deadline-Zyklus dz

$z_{dl} = z_{detect} + DI$ z_{dl} ist die Zykluszeit des Deadlinezyklus *dz* und z_{detect} die Zykluszeit der Detektionszyklus *det*.

Durch die Projektion der zeitlichen Anforderungen auf die diskrete Zykluszeit wird erreicht, dass alle im selben Zyklus ausgeführten Steuerfunktionen als gleichzeitig ausgeführt gelten. Damit spielt für das Deadline-Scheduling die Ausführungsreihenfolge in einem einzelnen Zyklus keine Rolle mehr. Pendente Ereignismeldungen mit dem selben Deadline-Zyklus (gleiche absolute Deadline) können einfach *FirstInFirstOut* (*FirstComeFirstServed*) ausgeführt werden. Die Implementation des Deadline-Scheduling kann damit stark vereinfacht werden, indem deadline-indexierte FIFO-Queues verwendet werden (Multilevel-Scheduling mit Deadline-Zyklen als Prioritätslevel). Im Besonderen erlaubt diese Implementationstechnik eine einfache und flexible Behandlung von *Overload Conditions*, wie z. B. die frühzeitige Erkennung potenzieller Deadline-Überschreitungen.

Es ist klar, dass auch für die mathematische Verifikation der Machbarkeit des Scheduling die Einführung der Zykluszeit eine grosse Vereinfachung bringt. Für die Verifikation spielen neben den Ausführungszeiten und den Deadline-Intervallen auch die minimalen Separationszeiten der Ereignisse (maximale Auftrittshäufigkeit) der Ereignisse eine zentrale Rolle. Auch diese können in *Anzahl Zyklen* angegeben werden. Für die periodischen Ereignisse wurde bereits vorausgesetzt, dass die Perioden Vielfache der Zyklusperiode sind. Bei den sporadischen Ereignissen wird entsprechend die minimale Separationszeit der Ereignisse in Einheiten der Zykluszeit ausgedrückt.

6.2.3 Ausführung der Ereignismeldungen: Unit Controller

Die zentrale Aufgabe des *Unit Controller* bei harten Echtzeitbedingungen ist die non-preemptive Aktivierung der Reaktiven Maschine nach dem *EDF*-Schedulingverfahren (Earliest Deadline First). Mit EDF wird immer die Task, bzw. die Ereignismeldung mit der kleinsten absoluten Deadline zur Ausführung gebracht (kleinster Wert bedeutet hier höchste Prioritätsstufe). Die absolute Deadline wird damit zur Grösse, welche die Priorität für die Ausführung einer Task, bzw. einer Ereignismeldung bestimmt. Solche Prioritäten werden als *dynamische* Prioritäten bezeichnet, weil sie jedesmal bei der Ankunft einer Task, bzw. der Detektion eines Ereignisses berechnet werden müssen.

In Multitasking-Systemen mit EDF-Scheduling wird die absolute Deadline $d_{absolut}$ der ankommenden Task durch Addition der Ankunftszeit $t_{arrival}$ und des Deadline-Intervalls *di* der Task ermittelt:

$$d_{absolut} = t_{arrival} + di$$

Die Anzahl der möglichen Prioritätsstufen zu einem bestimmten Zeitpunkt ist durch die Auflösung der Ausführungszeit (z. B. Mikrosekunden) bestimmt, und das sind immer sehr viele Werte. Man arbeitet deshalb mit einer *Deadline-Queue*, in welcher die ankommenden Tasks aufsteigend nach absoluter Deadline sortiert abgelegt werden.

Bei unserem zyklus-basierten Deadline-Scheduling der Ereignismeldungen ist die Bildung der absoluten Deadline gleich. Der wesentliche technische Unterschied zum Multitasking ist aber, dass die Detektionszeit z_{detect} eines Ereignisses und das Deadline-Intervall *DI* der entsprechenden Ereignismeldung nicht bezüglich der “kontinuierlichen” Ausführungszeit t ausgedrückt werden, sondern bezüglich der diskreten Zykluszeit z. Der Deadline-Zyklus z_{dl} spielt dabei wie gesagt die Rolle der absoluten Deadline.

$$z_{dl} = z_{detect} + DI \qquad \text{(vgl. mit } \textit{Abbildung 73}\text{)}$$

Die Anzahl der möglichen Prioritätsstufen zu einem bestimmten Zeitpunkt sind damit ganzzahlige Werte aus einem Intervall, dessen Grösse durch das maximale Deadline-Intervall bestimmt ist. Der Bereich möglicher Prioritätsstufen wird damit vergleichbar mit den Prioritätsbereichen konventioneller Multitasking-Systeme (statischen Prioritäten). Für die Realisierung des EDF-Scheduling kann deshalb ähnlich wie in Multitasking-Systemen ein *Multilevel-Scheduling-Mechanismus* implementiert werden (siehe unten), was eine grosse Vereinfachung bedeutet. Das ist wichtig, denn nur bei einfachen Implementierungen ist es machbar, dynamisch den kritischen Work-Load des laufenden Systems zu evaluieren und frühzeitig auf potenzielle Deadline-Überschreitungen zu reagieren.

Multilevel Deadline Scheduling für Ereignismeldungen sporadischer Ereignisse

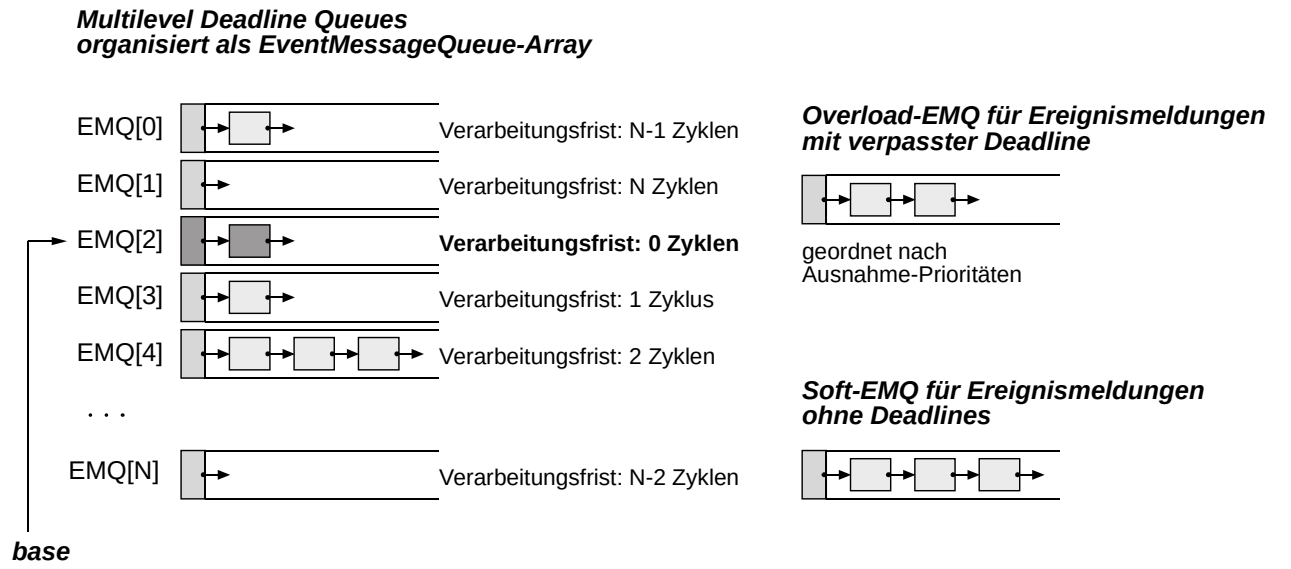
Die zyklusbasierten Deadline-Intervalle der Ereignismeldungen einer *Embedded Unit* sind durch ganzzahlige Werte aus ein einem beschränkten Bereich {1, .., N} gegeben. N ist zum Beispiel 120 (120 Zyklen). Um mit einer entsprechend beschränkten Anzahl Prioritäten arbeiten zu können, ersetzen wir die absoluten Deadlines durch *Verarbeitungsfristen*, d. h. die Deadlines werden auf die aktuelle Zykluszeit bezogen.

$$Verarbeitungsfrist = z_{dl} - z$$

Anstatt die absolute Anzahl Zyklen bis zum Deadline-Zyklus können genau so gut die Differenzen zur aktuellen Zykluszeit z als Prioritäten verwendet werden. Die Ereignismeldung mit der kleinsten *Verarbeitungsfrist* hat jeweils die höchste Priorität.

Die Werte der gültigen *Verarbeitungsfristen* liegen immer im Bereich {1, .., N} und können wie statische Prioritäten behandelt werden, ausser dass alle bei jedem Zykluswechsel um 1 zu dekrementieren sind. Wenn die *Verarbeitungsfrist* einer Ereignismeldung Null wird, ist für die entsprechende Ereignisverarbeitung die Deadline überschritten worden (Deadline Miss).

Die anstehenden Ereignismeldungen können damit wie konventionelle Multitasking-Systeme in einem Array von FIFO-Queues organisiert werden, nämlich eine FIFO pro Prioritätsstufe. Für unser Deadline-Scheduling bedeutet das, dass pro mögliche *Verarbeitungsfrist* eine als FIFO organisierte *Event Message Queue* notwendig ist. Im Unterschied zum klassischen Multilevel-Scheduling erhöht sich aber die Priorität jeder Queue mit jedem Zyklus-Takt, weil die *Verarbeitungsfrist* um 1 abnimmt. Wenn die *Verarbeitungsfrist* Null wird, muss die Queue leer sein, andernfalls liegt eine Deadline-Überschreitung vor.



Die Variable *base* zeigt immer auf die EMQueue mit Verarbeitungsfrist 0

base wird mit jedem Takt zyklisch erhöht: ++*base* % (N+1);

(*base*+k) % (N+1) ist der Index der EMQueue mit k Zyklen Verarbeitungfrist

Abb. 74: Multilevel Deadline Queues

In der *Abbildung 74* ist dargestellt, wie das *Multilevel Deadline Scheduling* mit einem zyklischen Array von *Event Message Queues* (Ereignismeldungswarteschlangen) realisiert werden kann. Jede EMQueue ist als FIFO organisiert. N ist das längste Deadline-Intervall der harten Ereignismeldungen.

Die *base*-Variable zeigt immer auf die EMQueue mit *Verarbeitungsfrist* 0. Diese Variable wird bei jedem Zykluswechsel zyklisch inkrementiert (d. h. modulo N+1). Die *Verarbeitungsfrist* einer EMQueue ist durch ihren relativen base-Index bestimmt:

$$\textit{Verarbeitungsfrist}(\text{EMQ[k]}) = (k - \text{base}) \bmod (N+1)$$

Für jedes detektierte Ereignis legt der *Event Detector* die entsprechende Ereignismeldung im EMQueue-System ab. Eine Ereignismeldung mit Deadline-Intervall DI kommt in die EMQueue mit *Verarbeitungsfrist* DI. Ereignismeldungen ohne spezifizierte Deadline kommen in die *Soft-EMQ*. Falls im System keine Ereignisse mit Deadlines vorkommen, reduziert sich das Multilevel Deadline Scheduling damit auf die klassische FIFO-Event-Technik.

Der *Unit Controller* führt pro Zyklus soviel Ereignismeldungen wie möglich aus. Die Ausführung erfolgt mit der *Verarbeitungsfrist* der abgelegten Ereignismeldungen als Priorität, d. h. es wird, nach aufsteigendem Index eine EMQueue nach der andern geleert. Erst wenn alle Ereignismeldungen mit Deadlines ausgeführt worden sind, kommen diejenigen aus der Soft-EMQ an die Reihe.

Falls in einem neuen Zyklus in der *base*-EMQueue (*Verarbeitungsfrist* 0) Ereignismeldungen vorhanden sind, sind sie nicht innerhalb des spezifizierten Deadline-Intervalles verarbeitet worden und müssen in die *Overload-EMQ* geschoben werden. Diese Warteschlange ist nach Ausnahmekriterien (overload priorities) organisiert und die abgelegten Ereignismeldungen werden im neuen Zyklus vor allen andern mit höchster Priorität verarbeitet.

6.3 RT-Scheduling periodischer und aperiodischer Tasks

Wenn für verschiedene Arbeiten eine gemeinsame Ressource benötigt wird, die nur exklusiv benutzt werden kann, muss die Zuteilung der Ressource zeitlich geplant werden (Scheduling). Das führt vor allem dann zu Problemen, wenn mehrere dringende Arbeiten zur selben Zeit ausgeführt werden sollten.

Beim Software-Scheduling sind die Arbeiten Programmteile paralleler Programme, die ausgeführt werden müssen, und die benötigte Ressource ist der Prozessor. Die parallelen Programme werden als Tasks, und die einzelnen Arbeiten als Jobs bezeichnet,

6.3.1 Verwendete Scheduling Begriffe und Notation

Job

Ein Programmteil, der ausgeführt werden soll.

Task

Eine Ausführungseinheit, die aus sequentiell auszuführenden Jobs besteht.

Taskzustände

running

Ein Job der Task wird ausgeführt.

ready

Ein Job der Task sollte ausgeführt werden.

blocked

Es kann kein Job der Task ausgeführt werden.

Job Zeiten (Zeitpunkte)

Arrival Time a

Ankunftszeit eines Jobs: Die Task wird *ready* (ausgelöst durch ein HW- oder SW-Ereignis).

Start Time s

Für einen bereitstehenden Job wird die Ausführung gestartet, d. h. die Task wird *running*.

Finishing Time f

Die Ausführung eines Jobs ist abgeschlossen, d. h. die Task wird *blocked*.

Deadline d

Die Ausführung des Jobs muss bis zum Zeitpunkt d beendet sein.

Bemerkung. Es wir hier angenommen, dass kein neuer Job ankommt, wenn die Task *ready* oder *running* ist. Die Annahme ist sinnvoll, wenn nur *periodische* oder *sporadische* Tasks vorkommen (siehe unten).

Periodische Task mit Periode T

Zwischen den Ankunftzeiten zwei sich folgender Jobs vergeht immer gleichviel Zeit. Diese Zeitspanne wird als Taskperiode T bezeichnet.

Aperiodische Task

Die Task ist nicht periodisch.

Sporadische Task mit Separationszeit T

Die Task ist aperiodisch, aber zwischen den Ankunftszeiten zweier Jobs vergeht mindestens die *Separationszeit* T. T wird auch als *Minimalperiode* oder *Minimal Interarival Time* der Task bezeichnet.

Statisches Scheduling

Das Scheduling ist bereits zur Kompilationszeit bestimmt (pre-run-Time-Scheduling).

Beispiele: Zeitscheiben-Scheduling, Tabellen-basierte Ausführung (Dispatcher).

Dynamisches Scheduling

Das Scheduling kann erst zur Laufzeit bestimmt werden.

Beispiele: Scheduling mit statischen oder dynamischen Prioritäten. Statische Prioritäten sind Konstanten; der Wert von dynamischen Prioritäten wird auch zur Laufzeit bestimmt.

Preemptives Scheduling

Die Ausführung eines Jobs kann vom Scheduler unterbrochen werden (Task-Switch). Für die Taskumschaltung ist eine Funktion notwendig, die den Task-Kontext auf dem Stack speichert (unterstützt durch RT-OS). Ist ein Job abgearbeitet, gibt er mit einem Betriebssystemaufruf (*System Call*) selber die Kontrolle an den Scheduler ab.

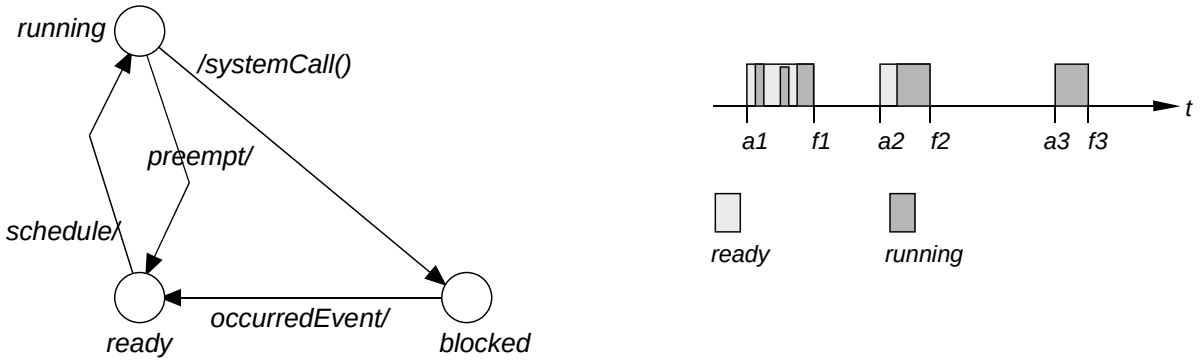


Abb. 75: Preemptives Scheduling: Zustandsdiagramm und Ausführungsszenario einer Task

Non-preemptives Scheduling

Non-preemptives Scheduling bedeutet, dass der laufende Job nie durch den Scheduler unterbrochen wird, d. h. ein Task-Wechsel kann nur stattfinden, wenn der laufende Job mit einem Betriebsystemaufruf selbst die Kontrolle abgibt. Das geschieht immer, wenn ein Job abgearbeitet ist. Das ist aber auch während der Abarbeitung möglich (*yield()*), um die Ausführung einer anderen Task zu ermöglichen.

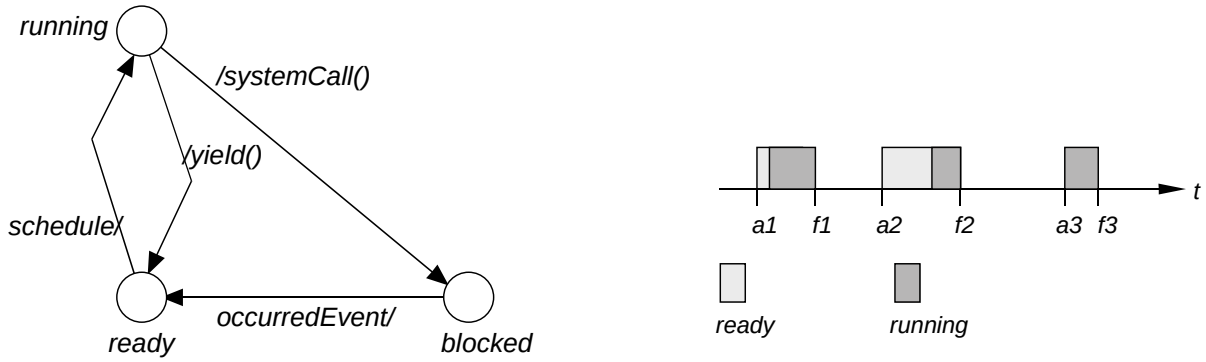


Abb. 76: Non-Preemptives Scheduling: Zustandsdiagramm und Ausführungsszenario einer Task

Non-*preemptives* Scheduling kann gut ohne Betriebsystem realisiert werden, indem die Jobs als Prozeduren implementiert werden, welche die Kontrolle jeweils mit *return* abgeben. Gesteuert wird die Ausführung der Jobs durch eine weitere Prozedur, die als zyklischer Controller aktiv ist.

- *Nachteil*: Neue zeitkritische Arbeiten müssen warten, bis der aktive Job den Prozessor freigibt.
- *Vorteil*: Der Zugriff auf gemeinsame Variablen (Shared Variables) muss nicht geschützt werden.

6.4 Rate Monotonic Scheduling

Für die Ausführungssteuerung periodischer Tasks können verschiedene Schedulingverfahren verwendet werden (Rate Monotonic, Deadline Monotonic, Earliest Deadline First). Wir betrachten hier das einfachste Schedulingverfahren: Rate Monotonic Scheduling.

6.4.1 Rate Monotonic (RM) Scheduling periodischer Tasks

m Periodische Tasks

Tasksystem: { $Task_k$ | k = 1, ..., n }

T_k : Task-Periode

C_k : Maximale Ausführungszeit pro Periode: $C_k < T_k$ (Worst Case Execution Time)

RM Scheduling Algorithmus *[10]*

> Die Frequenzen T_k^{-1} (*Rates)*) der Tasks definieren
> aufsteigend statische preemptive Task-Prioritäten.

Damit läuft unter den Tasks, die *ready* sind, immer die Task mit der kleinsten Periode.

RM Scheduling mit RT-OS. Die Tasks werden als periodische Tasks des RT-OS installiert, d. h. jede Task wird am Anfang ihrer Periode vom RT-OS in den Zustand *ready* versetzt. Hat ein Job in seinem Zyklus seine Arbeit beendet, muss er mit einem entsprechenden Betriebssystemaufruf wie zum Beispiel *waitPeriod()* die Kontrolle abgeben.

RM Scheduling ohne RT-OS. Eine einfache RM Scheduling-Implementierung ist ohne RT-OS möglich, wenn pro Task ein priorisierbarer Hardware-Timer zur Verfügung steht. Die Tasks können dann einfach als Interrupt-Subroutine der Timer ausgeführt werden.

Schedulability Verification (Überprüfung der zeitlichen Planbarkeit)

Kann jeder Task immer in jedem Zyklus die Prozessorzeit C_k zur Verfügung gestellt werden?

Definition

> Ein Task-System ist *schedulable* (zeitlich planbar), wenn garantiert werden kann,
> dass nie ein Job einer Task eine Deadline überschreitet (*deadline miss*).
>
> Bei unseren periodischen Tasks ist das Ende der Periode die Deadline eines Jobs.

Prozessorbenutzungsfaktoren (Processor Utilisation Factors) periodischer Tasks

Für die zeitliche Planbarkeit spielt der *Prozessorbenutzungsfaktor* eine entscheidende Rolle.

> ***Definition***
>
> $U_k = C_k / T_k$ ist Prozessorbenutzungsfaktor von $Task_k$.
>
> Der Prozessorbenutzungsfaktor $U_k < 1$ ist der Bruchteil der Prozessorzeit,
> welche $Task_k$ pro Zyklus im schlimmstem Fall benutzt.

Notwendiges Schedulability-Kriterium

> $U = U_1 + U_2 + ... + U_n <= 1$ “Man kann nicht mehr verteilen als vorhanden ist.”
>
> U wird als Prozessor-Benutzungsfaktor des Tasksystems bezeichnet.

Die Bedingung ist im allgeneinen nicht hinreichend.

Zeitlich Planbares RM Scheduling Beispiel U = 0.85..

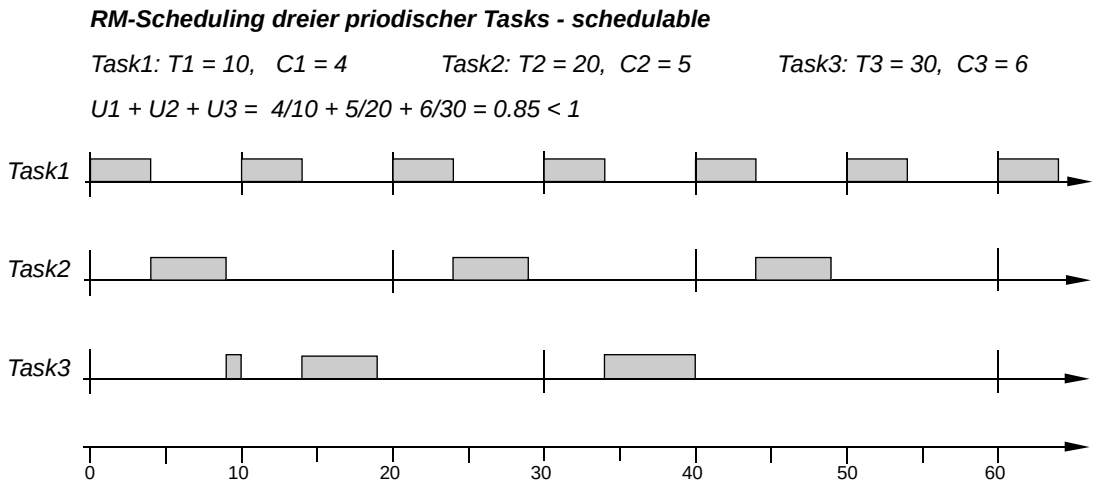


Abb. 77: RM Schedule dreier Tasks

Zeitlich nicht planbares RM Scheduling Beispiel - U = 0.96..

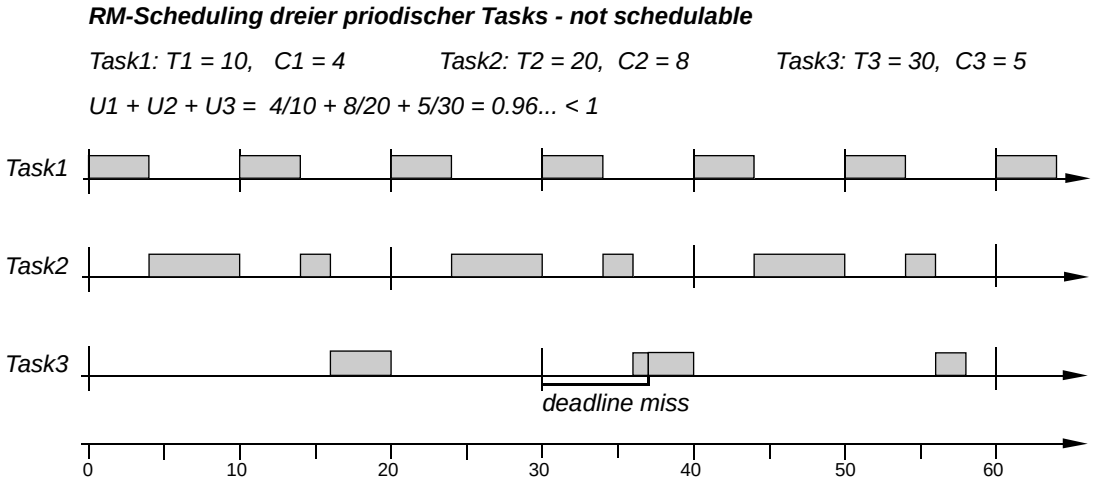


Abb. 78: RM Schedule mit Überschreitung einer Deadline

In der ersten Periode erhält die Task3 nicht genügend Prozessorzeit!

Das gleiche Beispiel ist mit EDF Scheduling zeitlich planbar - U = 0.96..

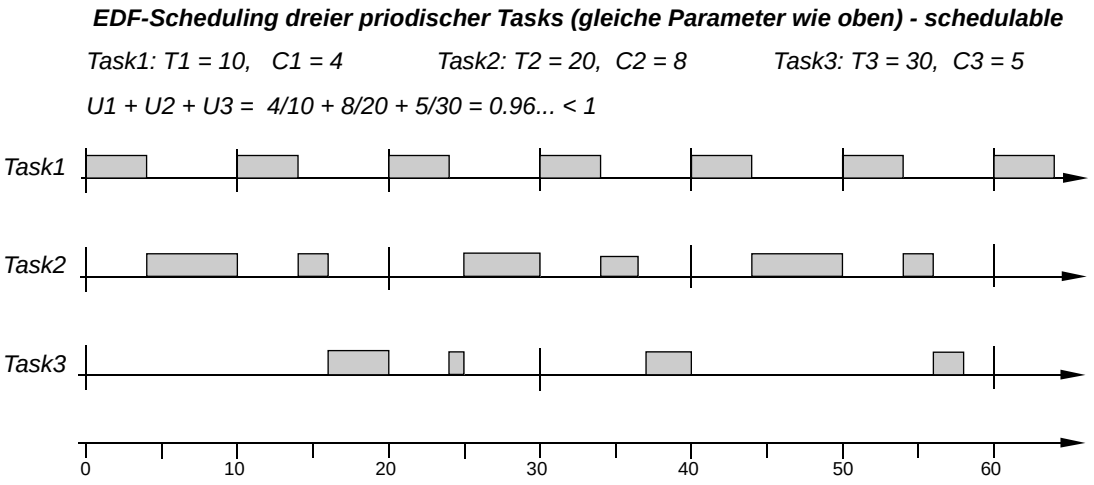


Abb. 79: EDF Schedule ist machbar

Beim *EDF*-Scheduling (*Earliest Deadline First*) hat immer die Task mit der frühesten *Deadline* die höchste Priorität.

Wie gross $U = U_1 + ... + U_m$ sein kann, hängt offenbar vom Schedulingverfahren ab.

Für ein bestimmtes Schedulingverfahren sollte eine obere Schranke für UB angegeben werden können. Ein solche Schranke wird als *Utilisation Bound UB* bezeichnet.

*Beispiele für Utilisation Bound*s:

RM: UB = 0.69	genau: $UB = n * (2^{1/n} - 1)$ (Theorem von 1973 Liu u. Layland [11])
	Grenzwert n -> Unendlich: UB = ln 2 = 0.69 (approximativ)
EDF: UB = 1	EDF ist *optimal*, auch für sporadische Tasks.
	Deadlines sind dynamische Prioritäten.

Harmonische Zyklusperioden T_1, ..., T_n

Am einfachsten wird das RM-Scheduling, wenn die Zyklusperioden harmonisch sind.

Definition

Die Perioden T_1, ..., T_n sind harmonisch,
wenn jede Periode ein ganzzahliges Vielfaches der nächst kleineren ist.

$T_{k+1} = n_k * T_k$ mit n_k ganzzahlig, k = 1, ..., n-1

Schedulability.

Bei harmonischen Zyklusperioden ist die Bedingung $U_1 + ... + U_n <= 1$
auch für das RM-Scheduling hinreichend, es gilt UB = 1. (Beweis als Übung)

RM Scheduling Beispiel mit harmonischen Perioden - U = 0.975

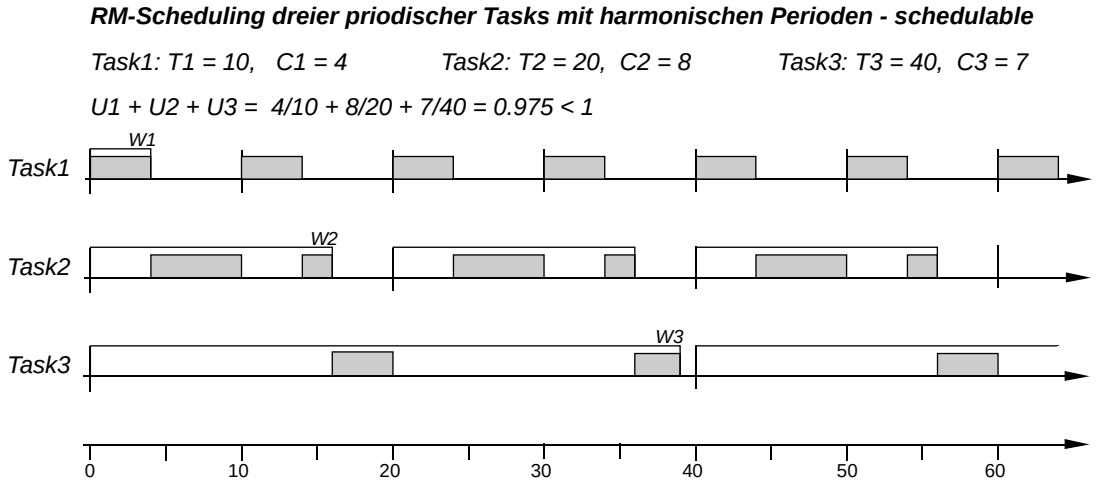


Abb. 80: RM Worst Case Schedule dreier Tasks mit harmonischen Perioden

Die in der Abbildung eingezeichneten Intervalle W_k heissen Work-Intervalle. Sie stellen die Zeitspanne dar, während denen eine Task im Worst-Case in einer Periode ready oder running ist. Man erkennt man gut, dass diese Intervalle bei harmonischen Perioden für den Worst Case Schedule in jedem Zyklus gleich gross sind.

Nicht-harmonische Lösungen mit rationalen Perioden-Verhältnissen (gemeinsamen Grundtakt), sind auch gut machbar, aber etwas aufwendiger in der Realisierung und weniger optimal bei den Konfigurierungsmöglichkeiten für die Prozessor-Benutzungsfaktoren. Die Work-Intervalle einer Task sind nicht mehr in jeder Periode gleich gross, das maximalen Work-Intervalle kann aber auch aufgrund der Schedulingparameter T_k und C_k statisch bestimmt werden. Um die Schedulability zu verifizie-

ren, muss der vollständige Worst-Case Schedule aufgezeichnet und analysiert werden, d. h. für die *Hyperperiode* (KGV der Perioden) des Tasksystems muss gezeigt werden, dass das maximalen Work-Intervalle jeder Task kleiner als die Periode ist, d. h. $W_k <= T_k$.

Irrationale Perioden-Verhältnisse von Zyklusperioden sollten vermieden werden, weil Ausführungs- und Verifikationsprobleme von kaum zu bewältigendem Komplexitätsgrad entstehen.

6.4.2 Periodische Task-Server für aperiodische Tasks in hybriden Tasksystemen

Ein hybrides Tasksystem umfasst periodische und aperiodische Tasks. Ein klassischer Ansatz, solche Systeme auszuführen besteht darin, die Ausführung der aperiodischen Tasks durch eine oder mehrere zusätzliche periodische Tasks zu steuern. Diese Tasks werden als *Task-Server* bezeichnet. Die *Task-Server* können so als periodische Subsysteme implementiert werden, die neben anderen periodischen Tasks laufen. Für die Steuerung der Ausführung der aperiodischen Tasks durch die *Task-Server* können unterschiedliche Schedulingverfahren angewendet werden, z. B. lokal non-preemptives EDF-Scheduling der Ereignis-Verarbeitung einer Embedded Unit (siehe 6.4).

Scheduling-Parameter für periodische Tasks und periodische Task-Server

Periodische Tasks - { $Task_k$ \| k = 1, ..., m }

T_k : Task-Periode

C_k : ***Ausführungszeit*** pro Periode (Worst Case Execution Time)

$U_k = C_k / T_k < 1$: Prozessor-Benutzungsfaktor der Task

Es wird davon ausgegangen, dass die pro Periode die Ausführungszeit C_k nie überschritten wird.

Periodische Task-Server - { $Task\text{-}Server_k$ \| k = m+1, ..., m+n },

T_k : Server-Periode

C_k : ***Server-Kapazität*** (Capacity) - zur Verfügung gestellte Prozessorzeit pro Periode

$U_k = C_k / T_k < 1$: Prozessor-Benutzungsfaktor des Servers

Ein Task-Server, der in einem Zyklus seine Kapazität C_k aufgebraucht hat, wird unterbrochen, auch wen der Task-Server noch aperiodische Tasks auszuführen hätte.

Das hybride Tasksystem besteht somit aus einer Menge normaler periodischer RM-Tasks und einer oder mehrerer periodischer Task-Server, welche die Ausführung der aperiodischen Tasks steuern. Bei den periodischen Tasks ist sicher gestellt, das sie ihre Worst Case Execution Time C_k nie überschreiten. Beim Scheduling der Task-Server hingegen muss der Scheduler oder eine weitere periodische Task dafür sorgen, dass den Task-Servern nur soviel Prozessorzeit zugeteilt wird, wie durch die Kapazitäten C_k vorgegeben ist. Die Task-Server selbst kümmern sich nicht darum. Sie aktivieren ihre aperiodischen Tasks bis keine mehr ready ist.

Schedulability

Ein notwendiges Kriterium für die Schedulability des Tasksystems kann wieder mit den Prozessor-Benutzungsfaktoren formuliert werden:

$$U = U_1 + ... + U_m + U_{m+1} + ... + U_{m+n} < 1$$

Typen periodischer Task-Server

In der Literatur [8] findet man unterschiedliche Typen periodischer Task-Server. Der Unterschied der verschiedenen Server liegt in der Art und Weise, wie die Kapazität verwaltet wird, wenn keine sporadische Task zu bedienen ist. Die verschiedenen Typen haben unterschiedliches Antwortzeitverhalten (Responsiveness), und auch bezüglich der zeitlichen Planbarkeit (Schedulability) des Systems sind unterschiedliche Bedingungen zu erfüllen.

Polling Server - PS
Am Anfang jeder Periode wird die Kapazität neu auf C gesetzt: c == C (c ist eine Variable, die zeitabhängig die verbleibende Kapazität festhält). Der Server kann pro Periode aperiodische Tasks bedienen, bis seine Kapazität ausgeschöpft ist (c== 0), oder bis er idle wird, d. h. keine aperiodischen Tasks mehr anstehen. Neu in der aktuellen Periode ankommende Tasks werden in diesem Fall erst in der anschliessenden Periode bedient.

Die Implementierung ist sehr einfach. Der Server verhält sich wie eine normale periodische RM-Task. Die Ausführung kann damit gleich wie im Fall von nur periodischen Tasks geplant werden. Dafür ist das Antwortzeitverhalten für die bedienten aperiodischen Tasks nicht optimal. Wenn der Server bereits am Anfang seiner Periode idle wird, muss eine kurz darauf ankommende aperiodische Tasks bis zur nächsten Periode warten (Polling Latenzzeit).

Deferrable Server - DS
Wie beim PS wird die Kapazität am Anfang der Periode auf C gesetzt. Im Unterschied zum PS kann aber ein DS, der idle geworden ist, zu einem späteren Zeitpunkt wieder aktiviert werden, falls in der Zwischenzeit eine aperiodische Task angekommen ist (Interrupt). Bedingung ist, dass die Kapazität noch nicht ausgeschöpft ist.

Die Implementierung des DS ist etwas komplexer als diejenige des PS. Das Antwortzeitverhalten ist dafür besser. Der DS verhält sich jedoch nicht wie eine RM-Task und kann deshalb bei nicht-harmonischen Perioden die zeitliche RM Scheduling-Planbarkeit der periodischen Tasks korrumpieren.

Priority Exchange Server - PES
Wie beim DS kann der PES mehrmals pro Periode idle werden und wieder aktiviert werden. Die Ausführung erfolgt aber auf der Prioritätsstufe der Task, welche die Ausführungszeit beim idle werden als Kapazität übernommen hat. Es wird dauernd bei allen Tasks über die vom Server übernommenen Kapazitäten Buch geführt.

Die Implementierung ist komplex, das Antwortzeitverhalten ist vergleichbar mit dem PS. Dafür verhält sich der PES wie eine RM-Task, d. h. der PES hat keinen negativen Einfluss auf die Planbarkeit der periodischen Tasks.

Sporadic Server - SS
Die Kapazitätserneuerung erfolgt nicht periodisch, sondern sporadisch nach dynamischen Regeln in Abhängigkeit der aperiodischen Aktivitäten.

Die Implementierung ist noch aufwendiger als die des PES, dafür ist das Antwortzeitverhalten verbessert, und vor allem verhält sich der SS wie eine RM-Task, d. h. die Bedingungen für die zeitliche Planbarkeit der Systems ist gleich wie bei einem rein periodischen Task-System.

Cycle Stealer - CS
Der Cycle Stealer ist nicht ein periodischer Server. Der CS verteilt voraussehbare Idle-Zeiten der periodischen Tasks zum voraus. Damit wird ein sehr gutes Antwortzeitverhalten erhalten. Nachteil ist aber die sehr aufwendige Implementierung.

Im folgenden Unterkapitel wird gezeigt, wie Embedded Units mit Hilfe eines Real-Time Betriebssystem (RT OS) als *Polling Server* implementiert werden können. *Polling Server* sind genügend, wenn die Ereignisse für eine Embedded Unit immer am Anfang eines neuen Unit-Zyklus detektiert werden (*quasiperiodic event detection*). *Deferrable Server* wären sinnvoll, wenn es möglich sein soll, während des ganzen Unit-Zyklus neu aufgetretene Ereignisse detektieren zu können, zum Beispiel bei interrupt-getriebener Event Detection.

6.5 Embedded Unit Controllers als Polling Server ausgeführt

Embedded Units sind parallele asynchrone Ausführungseinheiten eines Embedded Systems. Jede Embedded Unit hat einen *Unit Controller* als aktiven Control Thread, der sowohl die reaktiven Komponenten der Unit als auch die Verbindungssoftware zu den Peripheriegeräten steuert. Funktionale Abhängigkeiten zwischen Embedded Units werden mittels asynchronem nicht-blockierenden Messing-Passing realisiert. Auf gesendete Messages wird gleich wie auf detektierte externe Ereignisse reagiert.

Das globale Scheduling der *Unit Controller* von *Embedded Units*, die auf demselben Prozessor implementiert sind, muss mindestens zum Teil preemptiv (unterbrechend, verdrängend) erfolgen, weil die Verarbeitungszeiten von Ereignissen einer Unit mit grosser Zyklusperiode grösser sein können als die Zyklusperiode einer Unit mit kurzen Ereignisverarbeitungen. Solche Verhältnisse sind zu erwarten. Sie sind ja ein Hauptgrund dafür, dass auf demselben Prozessor überhaupt mit mehreren Units gearbeitet wird.

Embedded Units könnten durch einfaches preemptives zyklisches Zeitscheiben-Scheduling (Round-Robin Scheduling) ausgeführt werden. Für die einzelnen Units müsste die Grösse der Zeitscheiben konfigurierbar sein, so dass die Zykluszeiten und die Deadlines sicher eingehalten werden können. Die Übertragung von Meldungen zwischen den Reaktiven Maschinen der Units würde mit Intertask-Kommunikationsmitteln gelöst.

Solche statische Zeitscheiben-Lösungen sind jedoch bezüglich Ausführungsressourcen sehr ineffizient, weil aufgrund der sporadischen Ereignisverarbeitung pro Unit die angeforderte Prozessorzeit zeitlich stark schwankt. Damit ginge bei dieser vollständig statischen Scheduling-Lösung viel Prozessorzeit in den Leerlaufphasen der einzelnen Units verloren. Eine gute Lösung muss auf Systemebene dynamisch die Prozessorzeit ausgleichen können, dabei aber gleichzeitig jeder Unit den angeforderten Teil der Prozessorzeit auch bei gemeinsamen Lastspitzen garantieren können.

Lösungen, die diese Anforderungen erfüllen, werden durch das Konzept periodischer *Task-Server* unterstützt, das gut auf Embedded Units angewendet werden kann. Wir werden Task-Server betrachtenten, die mit dem *Rate Monotonic Scheduling* Verfahren ausgeführt werden.

6.5.1 Unit Controllers als Polling Server

Ein *Unit Controller* steuert lokal die Ausführung einer Embedded Unit. Anstatt Jobs von aperiodischen Tasks werden anstehende Ereignismeldungen durch die Reaktive Maschine der Embedded Unit ausgeführt (Ereignisverarbeitungen). Der Unit Controller wirkt gewissermassen als *Reaktive Machine Server.* Ein Unit Controller aktiviert aber auch noch sämtliche Funktionen, welche die Embedded Interaktion betreffen wie I/O Handling, Ereignisdetektion und Aktionsinitiierung. Bei mehreren Prozessoren kommt noch die Implementierung der Inter-Prozessor-Kommunikation dazu.

Ein weiterer Unterschied zu klassischen Task-Servern besteht bezüglich den periodischen Verarbeitungen. Ein klassischer Task-Server bedient nur aperiodische Tasks. Periodische Task werden auf derselben Ebene wie der Task-Server nach derselben RM Scheduling-Policy ausgeführt. Bei Embedded Units hingegen wird die periodische Ereignisverarbeitung in der Regel in die Reaktive Maschine der Embedded Units integriert. Der tiefere Grund dafür ist, dass periodische und aperiodische Ereignisverarbeitungen normalerweise funktionale Abhängigkeiten aufweisen und deshalb in derselben reaktiven Maschine ausgeführt werden sollten, damit nicht mit der Reaktiven Maschine einer asynchron laufenden Embedded Unit kommuniziert werden muss.

Wir betrachten hier der Einfachheit halber Implementierungen als Polling Server, die mit dem *Rate Monotonic Scheduling* Verfahren gesteuert werden. Als Alternative zu RM wäre auch eine Realisierung des Server-Scheduling mit dynamischen *EDF*-Prioritäten denkbar (bessere Auslastung des Systems bei nicht harmonischen Perioden), die Implementierung ist aber erheblich komplexer und kaum mit konventionellen Real-Time-Betriebssystemen machbar.

Wir setzen im folgenden ein 1-Prozessor-System voraus, das mehrere *Polling Embedded Units* (EU_1, , ..., EU_B) und eine *Background Task* für tiefer prioritäre Arbeiten ausführen soll. Dabei wird als Ser-

verperiode T_k der Wert der *Zyklusperiode* der entsprechenden Embedded Unit EU_k genommen. Verschiedene Werte machen kaum einen Sinn. T_k < *Zyklusperiode* führt nicht zu kürzeren Reaktionszeiten, weil die Latenzzeit bei der Ereignisdetektion durch die *Zyklusperiode* der Embedded Unit bestimmt ist. T_k > würde unnötig diese Latenzzeit erhöhen. Bei Implementierungen als *Deferrable Server* kann hingegen mit T_k < *Zyklusperiode* unter Umständen eine Steigerung der Performance erreicht werden.

Ausführungsszenario dreier Polling Embedded Units

Die Ausführung der *Unit Controller* der Embedded Units wird durch eine zusätzliche periodisch aktivierte Task gesteuert, die als *Dispatcher* bezeichnet wird. Die Dispatcher-Periode wird als Dispatch-Takt *u* bezeichnet. *Abbildung 81* zeigt ein Ausführungsszenario dreier Polling Embedded Units. Die Unit-Perioden sind harmonisch. Die Perioden und die Kapazitäten der Units sind Vielfache der Dispatch-Periode u. Das kürzeste Kapazitätsintervall (C1) ist in diesem Beispiel gleich gross wie der Dispatch-Periode.

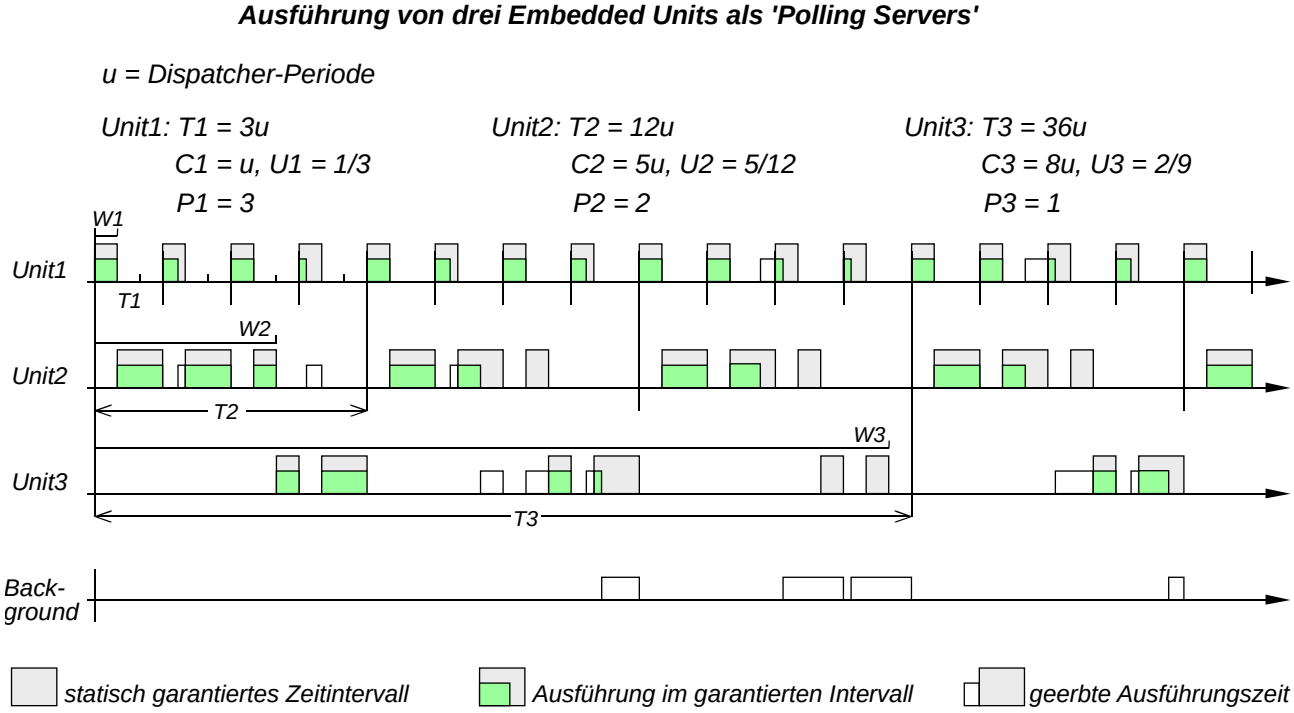


Abb. 81: Ausführung von drei Polling Embedded Units

Dispatch-Zeitpunkte

In *Abbildung 81* sind für jede Embedded Unit EU_k die statisch garantierten Zeitintervalle eingezeichnet. Das sind die Segmente, während denen eine Unit aktiv ist, wenn alle Units ausgelastet sind (*Worst Case Schedule*). Diese Segmente sind zur Entwicklungszeit bestimmbar, und sie werden benutzt, um die Dispatch-Zeitpunkte zu bestimmen, d. h. immer am Schluss eines solchen Segments wird durch den Dispatcher die Unit zur Ausführung gebracht, bei der ein solches Intervall beginnt. Gibt es kein solches Intervall, wird die Background Task aktiviert (kurz vor Ende von T_3).

Pro Unit-Zyklus ergeben diese Segmente zusammen die garantierte Ausführungszeit C_k. Das umfassende Intervall der Segmente eines Zyklus wird als Work-Intervall W_k bezeichnet. Das ist der maximale Zeitabschnitt, während dem die Embedded Unit EU_k in ihrem Zyklus arbeiten kann, wenn alle Units ausgelastet sind. Weil die Perioden harmonisch sind, wiederholt sich in jedem Zyklus das selbe Segment Muster. Aus diesem Grund ist auch die Programmstruktur des Dispatcher zyklisch. Mit D wird die Anzahl Dispatch-Takte des Dispatcher-Zyklus bezeichnet. Er dauert gleich lang wie die grösste Unit-Periode, in unserem Beispiel ist das T_3 (Hyperperiode der RM-Ausführung). Die Ver-

teilung der Ausführungszeiten der Embedded Units im Worst-Case Fall kann damit in einer *Dispatch-Tabelle* mit D Element*en* festgehalten werden.

Im Normalfall gibt es bei jeder Unit viele Zyklen, in denen der Unit Controller *idle* wird, weil alle anstehenden Ereignismeldungen verarbeitet worden sind. Für die Verteilung der nicht benutzten Ausführungszeit ist der Dispatcher nicht zuständig. Diese Zeit erbt eine andere Unit, die noch zu arbeiten hat. Die Aktivierung der erbenden Unit erfolgt automatisch aufgrund von Basisprioritäten der Units. Die Basisprioritäten entsprechen in diesem Beispiel den Frequenzen der Units, was aber nicht so sein muss.

6.5.2 RT-OS-basierte Implementierung mit statischem Dispatcher

Bei der im folgenden beschriebenen Implementierung wird ein RT-OS vorausgesetzt, das einen Betriebssystemaufruf (System Call) zur Verfügung stellt, mit dem die Priorität einer gewählten Task verändert werden kann (z. B. *changePriority(taskId, newPriority)*). Damit wird es möglich, die Ausführung der Unit Controller durch eine zusätzliche periodische Task zu steuern, indem gezielt die Prioritäten der Unit Controller manipuliert werden.

Die Unit Controller der Embedded Units werden als OS-Tasks mit fixen Basisprioritäten installiert (Basisbereich = {1, ..., B}, dabei bezeichnet B die Anzahl Units. Die *Background Task bekommt die Priorität 0.* Der Dispatcher wird als periodische Task mit Priorität B+2 installiert (höchste Priorität). Die Periode u des Dispatchers ist die Basis-Zeiteinheit der Ausführungssteuerung. Sie wird als Teil der kleinsten Unit-Periode definiert.

Der Dispatcher sorgt in seinen Dispatch-Zyklen dafür, dass der auszuführende Unit Controller die *Dispatch-Priorität* B+1 erhält und die restlichen Unit Controller auf ihre Basis-Priorität gesetzt sind. Damit die Ausführungssteuerung nicht die Performance des Systems beeinträchtigt, müssen die Ausführungszeiten des Dispatchers im Vergleich zur Dispatschperiode sehr kurz sein, was in der Regel auch der Fall ist.

Wird eine laufende Unit idle, d. h. es sind keine Ereignismeldungen mehr zu verarbeiten, muss der entsprechende Unit Controller mit einem geeigneten Betriebssystemaufruf (z. B. *semaphoreWait()*) die Kontrolle abgeben (kein *busy wait* !) und sich für den Rest des aktuellen Zyklus blockieren. Anstatt der blockierten Unit läuft dann - durch den Scheduler des Betriebssystem gesteuert - von den nicht blockierten Units diejenige mit der höchsten Basispriorität (vererbte Prozessorzeit). Die Blokkierung ist dann durch den Dispatcher bei der Aktivierung beim nächsten Zyklusstart mit dem entsprechenden Betriebssystemaufruf (z. B. *semaphoreSignal()*) wieder zu lösen.

Der Dispatcher wird als *statisch* bezeichnet, weil die Unit-Perioden und die Unit-Kapazitäten alle Task-Switch-Zeitpunkte bereits vor der Laufzeit bestimmen. Die Ausführungsintervalle der verschiedenen Units können damit schon vor der Systemausführung in einer Dispatch-Tabelle festgehalten werden. Das wäre zum Beispiel bei einer Implementierung der Units als *Deferrable Server* nicht mehr möglich, weil die verbleibenden Ausführungszeiten der Units dynamisch bestimmt sind.

Abbildung 82 zeigt eine Embedded Unit, die sowohl einen periodischen *Input Handler* (eine Aktivierung pro Unit-Zyklus) als auch einen interrupt-getriebenen sporadischen *sInput Handler* besitzt, der die neuen Inputdaten für den *Event Detector* in einer Input-Queue ablegt. Interrupt-getriebene Input-Handler sind notwendig, wenn z. B. die Sensordaten verteilt erfasst und mit einem Feldbus zum Rechner der Embedded Unit übertragen werden. Der *Event Detector* wird in diesem Beispiel nur einmal pro Zyklus aktiviert.

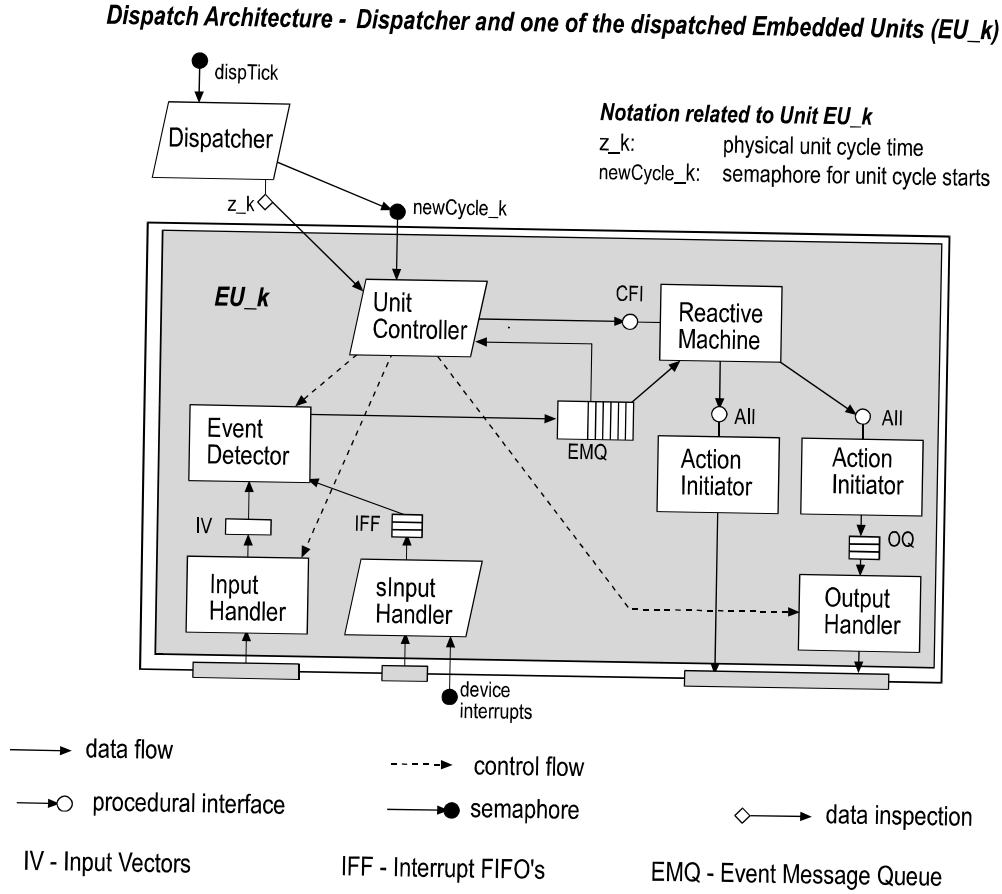


Abb. 82: Architektur - Dispatcher und Embedded Unit

Durch Inspektion der Zeitvariablen *z_k* des Dispatcher (Inspektion einer fremden Variablen ist durch ein Rombus dargestellt) kann der *Unit Controller* nach jeder Ereignisverarbeitung überprüfen, ob physikalisch ein neuer Zyklus begonnen hat. In jedem Unit-Zyklus werden neue Inputdaten erfasst und neue Ereignisse detektiert. Anschliessend erfolgt die Ereignisverarbeitung, indem der *Unit Controller* für Meldungen aus der *Event Message Queue* die *Reactive Machine* triggert. Der *Unit Controller* gibt seine Kontrolle über den Semaphor *newCycle_k* ab, sobald die *Event Message Queue* leer wird. Er wird dann vom *Dispatcher* für das erste Ausführungssegment im nächsten neuen Zyklus deblockiert.

Dispatcher

In *Abbildung 83* ist Programstruktur und die Definition der lokalen Variablen des Dispatcher dargestellt. Zusätzlich ist die Verwendung der RT-OS-Prioritäten erläutert.

Control Variables. Der aktuelle Dispatch-Takt wird in der Dispatch-Time *dt* nachgeführt. Die Variable *ux* hält die Identität der laufenden Unit fest, und die Variable *dispTick* referenziert den Semaphor des Dispatch-Timers.

Dispatch Table. Die Tabelle enthält für jeden *Dispatch-Step* des Dispatcher-Zyklus ein Element (D Elemente). In jedem Tabellenelement ist in der Variablen *uid* eingetragen, welche Unit zu aktivieren ist. Ist kein Task-Switch auszuführen, wird *uid* auf -1 gesetzt. Die zweite Tabellen-Variable *cycleStart* enthält eine 1, wenn von der zu aktivierenden Unit auch die Zykluszeit erhöht und der Unit-Semaphor freigegen werden muss.

UnitControlBlock Array. Die Tabellenelemente enthalten Unit Variablen, auf welche auch der entsprechende Unit Controller zugreifen kann (shared variables). Die Konstante *baseprio* ist die Basispriorität der Unit. Die Variable *z* enthält den aktuellen Wert der diskreten physikalischen Zykluszeit, und über die Semaphor-Variable *newCycle* kann sich eine Unit bei leerer Event Message Queue blockieren. Sie wird vom Dispatcher deblockiert, wenn der neue Unit-Zyklus beginnt.

OS-based Task Switch Function TS(k). Die Funktion TS(k) sorgt durch Manipulation der Unit-Prioritäten dafür, dass der Unit Controller der Unit *EU_k* die Kontrolle bekommt (*task switch*).

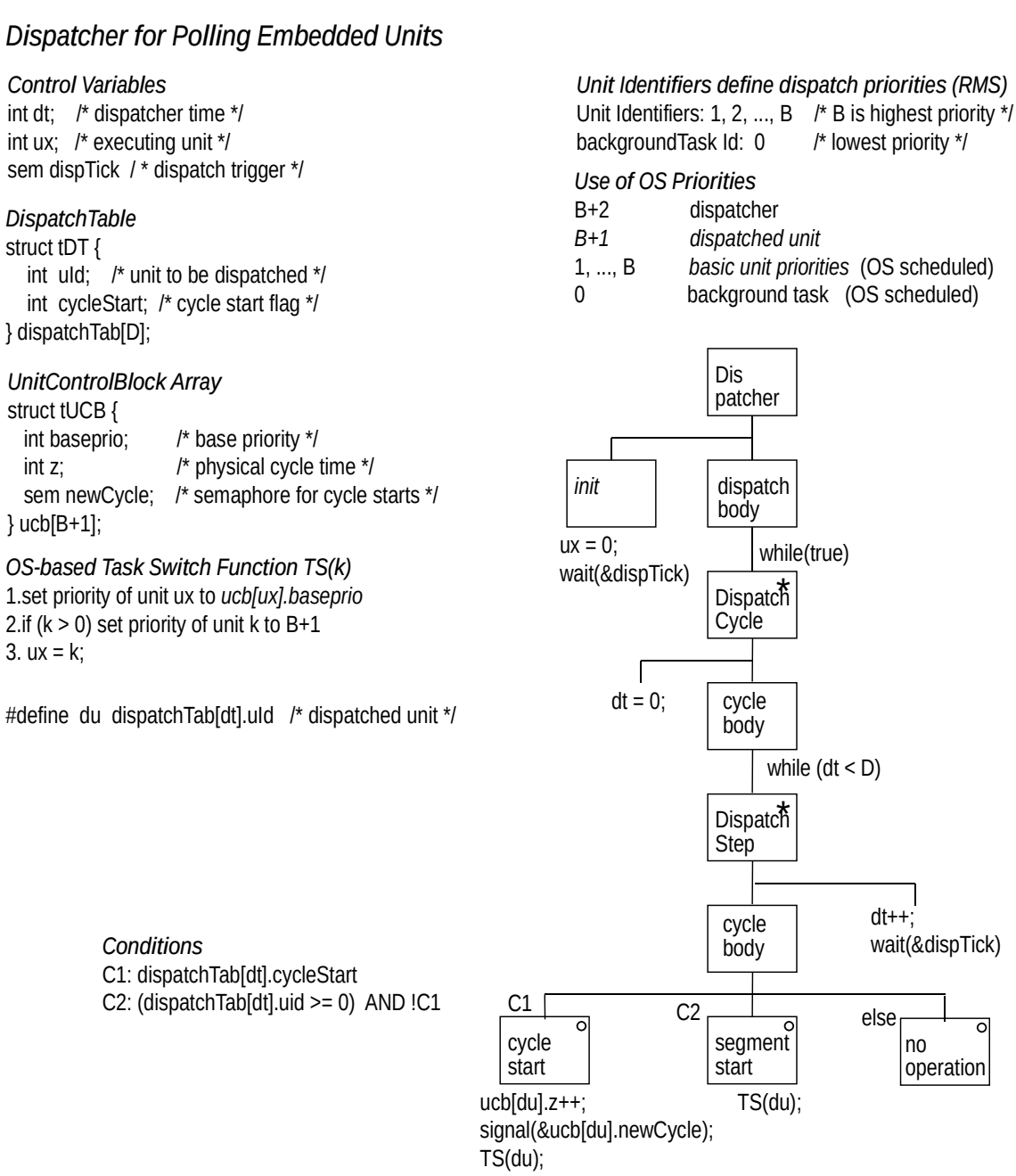


Abb. 83: Struktogramm: Dispatcher für Polling Embedded Units

Die Programmstruktur ist eine Iteration von *Dispatch Cycles*, die aus D *Dispatch Steps b*estehen. In jedem *Dispatch Step* wird entweder ein neuer Zyklus einer Unit gestartet (*cycle start*), oder nur ein weiteres Segment in einem Unit-Zyklus gestartet (*segment start*), oder es muss nichts gemacht werden (*no operation*). Der Wert der Bedingungen (*Conditions C1, C2*) für diese alternativen Strukturblöcke sind durch die Einträge in der Dispatch Tabelle bestimmt. Die auszuführenden Operationen sind jeweils bei den entsprechenden Struktur-Boxen des Struktogramms angehängt.

Unit Controller

Abbildung 84 zeigt die Programmstruktur des *Unit Controller* der Embedded Unit EU_2 (quasi-periodische Version). Nach der Initialisierung folgt die iterative Ausführung der Unit-Zyklen. Pro Zyklus wird zuerst der *Input Handler* und der *Event Detector* aufgerufen.

Anschliessend wird im '*busy period*'-Loop für Event Messages aus der Event Message Queue wiederholt die *Reactive Machine* getriggert. Können alle anstehenden Event Messages verarbeitet werden (*EMQ.empty() == TRUE*), wird der '*busy period*'-Loop verlassen. Nach dem Aufruf des *Output Handler* wartet dann der Unit Controller mit einem *wait()*-Aufruf bei seinem *newCycle-Semaphor* auf den nächsten Zyklusstart.

Hat der Zyklusstart bereits während einer Ereignisverarbeitung statt gefunden, wird der '*busy period*'-Loop aufgrund der erhöhten physikalischen Zykluszeit *ucb[2].z* verlassen, sobald die *Reactive Machine* die Kontrolle zurückgibt. Der anschliessende *wait()*-Aufruf führt in diesem Fall nicht zu einer Blockierung, weil der Semaphor bereits vom Dispatcher freigegeben worden ist.

Unit Controller der Embedded Unit EU_2 - Implementierung als Polling Server

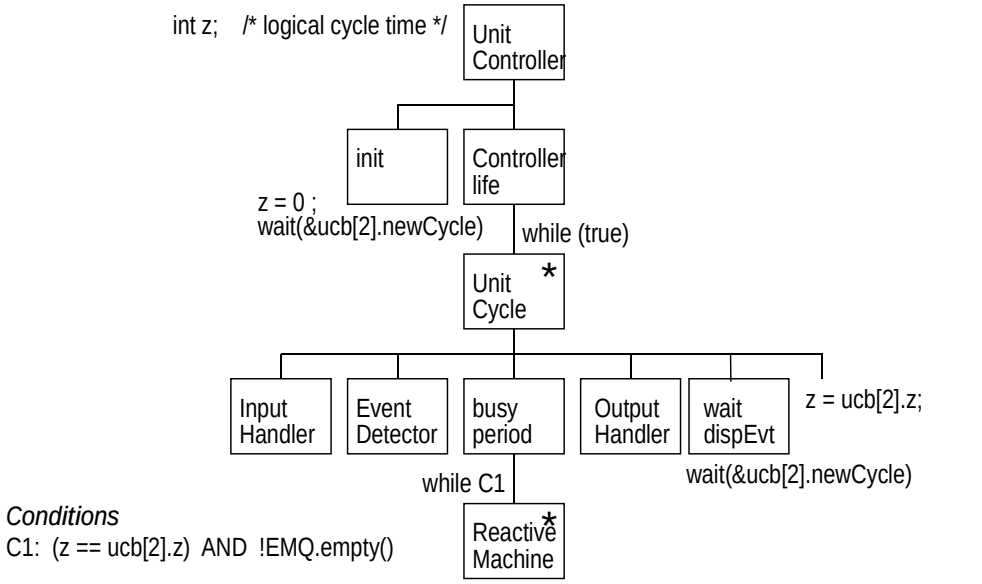


Abb. 84: Struktogramm: Unit Controller der Embedded Unit EU_2 (periodischer Input-Handler)

Bemerkung

Einfachere kooperative Lösung. Eine einfachere Ausführungssteuerung der Unit Controller ist möglich, wenn die Unit Controller selber die Verantwortung übernehmen ihre Kapazität pro Periode nicht zu überschreiten. Sie können dann gleich wie periodische Tasks mit einfachem RM-Scheduling ausgeführt werden. Falls aber doch ein Unit Controller die Kapazität überschreitet, kann das zu unvorhersehbaren Problemen bei der lokalen Ereignisverarbeitung der Unit Controller führen, weil für diese nicht die garantierte Prozessorzeit verfügbar ist.

6.6 Schedulability Verification

Hard Real-Time-Systeme verlangen, dass immer alle Deadlines einzuhalten sind. Das heisst, dass die Machbarkeit des Scheduling vor der Laufzeit gezeigt werden muss (*Scheduling feasability verification, schedulability verification*).

Ein Task-System ist mit einem bestimmten Scheduling Verfahren schedulable (zeitlich planbar), wenn garantiert werden kann, dass nie eine Deadline überschritten wird. Die Garantie sollte aufgrund der Taskparameter bewiesen werden können. Die möglichen Taskparameter sind in der Regel durch ein bestimmtes Taskmodell bestimmt, z. B. *'periodische non-preemptive Tasks mit Deadlines gleich oder kleiner als die Periode'*.

Zu Beginn dieses Kapitels ist bereits die Schedulability periodischer Task-Systeme diskutiert worden. Für solche einfachen Task-System kann die Garantie der Machbarkeit des Scheduling gezeigt werden, indem verifiziert wird, dass das die Summe der Prozessorbenutzungsfaktoren der Tasks durch einen entsprechenden *Utilisation Bound* beschränkt ist. Bei aperiodischen Ereignisverarbeitungen ist das im allgemeinen nicht mehr möglich, weil das Auftreten der Ereignisses zeitlich nicht voraussagbar (non-predictable) ist. Damit die Scheduling-Aufgabe überhaupt lösbar ist, muss zumindest bekannt sein, wie häufig solche Ereignisse im Worst Case auftreten. Bei *sporadischen* Ereignissen muss der minimale zeitliche Abstand zweier Instanzen desselben Ereignistyps bekannt sein. Bei nicht-sporadischen aperiodischen Ereignissen (*Burst-Ereignisse*) ist der minimale zeitliche Abstand Null. Für solche Ereignisse, die typischerweise in kommunizierenden Systemen auftreten (Bursts), muss zumindest die zeitliche Dichte der Ereignisse für ein vorgegebenes Burst-Intervall bekannt sein.

Bei Embedded Units bestimmt der Prozessorbenutzungsfaktor einer Unit, ob bei der Ereignisverarbeitung immer alle Deadlines eingehalten werden können. Um sicher zu stellen, dass die Ereignisverarbeitung in diesem Sinne zeitlich planbar ist, muss aber eine umfassende Analyse der implementierten Komponenten erfolgen, das heisst es sind die für den schlimmsten Fall (Worst Case) die notwendigen Prozessorbenutzungszeiten zu ermitteln. Das ist das Hauptthema diese Unterkapitels. Es wird im folgenden gezeigt, wie eine solche Analyse mit dem *'Processor Demand'*-Verfahrendurchgeführt werden kann.

Pragmatische Ansätze bei der Festlegung der Prozessorbenutzungsfaktoren

Ein Hauptproblem bei der Festlegung von Benutzungsfaktoren für nicht voraussagbare Komponenten-Aktivierungen ist der notwendige pessimistische Worst Case Ansatz. Erstens ist es nicht immer einfach, diese Worst Case Häufigkeiten zu bestimmen, und zweitens - was noch viel problematischer ist - kann es sein, dass man aufgrund der Worst Case Annahmen zum Schluss kommt, dass das Scheduling mit den verwendeten Prozessoren nicht machbar ist (not feasable), obwohl die Wahrscheinlichkeit für das Auftreten des Worst Case verschwindend klein ist.

In der industriellen Praxis werden deshalb pragmatischere Ansätze vorgezogen, zum Beispiel indem Prozessorbenutzungsfaktoren aufgrund experimenteller Erfahrungswerte festgelegt werden und das System so gebaut wird, dass es potenzielle Worst Case Situationen frühzeitig erkennen und geeignete Massnahmen auslösen kann. Das ist aber nur möglich, wenn das Scheduling möglichst einfach ist, indem soviel wie möglich statisch geplant wird und bei funktionalen Abhängigkeiten preemptive Unterbrechungen auf das notwendige Minimum beschränkt werden.

Das im vorgehenden Unterkapitel diskutierte Multilevel Deadline Scheduling unterstützt solche Ansätze. Der aktuelle Work Load für die sporadisch auftretenden non-preemptiven Ereignisverarbeitungen kann für jede Verarbeitungsfrist sehr einfach ermittelt und überprüft werden. Eine mögliche geeignete Massnahme bei einem detektierten kritischen Work Load wäre zum Beispiel, unkritische Funktionen temporär auszuschalten, indem eine entsprechende Ereignismeldung für die reaktive Maschine erzeugt wird, die geeignete Modus-Umschaltungen bewirkt. Bei sicherheitskritischen Anlagen muss das System unter Umständen vorsichtig heruntergefahren werden. Das muss so erfolgen, dass keine Schäden entstehen (Gracefull Degradation).

6.6.1 Das Processor Demand Verfahren für hybride Task Systeme

Das Processor-Demand-Verfahren ist eine Technik, mit der für hybride Tasksysteme (periodische und sporadische Tasks) die Schedulability verifiziert werden kann [8, 14].

Die 'Processor Demand'-Funktion PD()

Für eine zeitliche Sequenz $\sigma = (J_k(a_1), J_k(a_2), ...,)$ von Jobs einer Task *Task*$_k$ wird die sogenannte *Processor-Demand-Funktion* $PD_\sigma(t_1, t_2)$ definiert. Dabei sind $a_1, a_2, ...,$ die Ankunftszeiten der Jobs.

DEFINITION **$PD_\sigma(t_1, t_2)$**

Der Processor Demand $PD_\sigma(t_1, t_2)$ ist die Prozessorzeit, die bei der Jobsequenz σ für die Ausführung derjenigen Jobs benötigt wird, die im Intervall $[t_1, t_2]$ ankommen und auch in diesem Intervall verarbeitet werden müssen.

Formal ausgedrückt:

$PD_\sigma(t_1, t_2) = n * C_k$

Dabei ist n die Anzahl Jobs $J_k(a_j)$ mit $a_j \in [t_1, t_2]$ und $a_j + D_k \in [t_1, t_2]$

(C_k = Worst Case Execution Time, D_k = Deadline-Intervall)

Processor Demand im Intervall $[t_1, t_2]$ für eine Job-Sequenz der Task Task$_k$

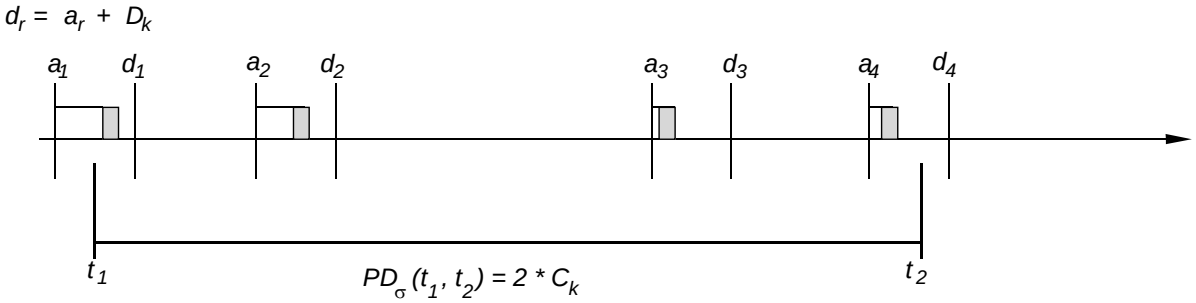


Abb. 85: Processor Demand im Intervall $[t_1, t_2]$: $PD_\sigma(t_1, t_2) = 2 * C_k$

In *Abbildung 86* ist eine Sequenz von 4 ankommenden Jobs der Task T_k dargestellt. Nur der zweite und dritte Job tragen zum Processor Demand für das Intervall $[t_1, t_2]$ bei. Der erste Job kommt vor diesen Intervall an, und der vierte Job hat seine Deadline ausserhalb des Intervalls.

'Processor Demand'-Funktion für mehrere Tasks

Ein allgemeines Szenario ankommender Jobs verschiedener Tasks ist durch m verflechtete Sequenzen $\sigma 1, ..., \sigma m$ gegeben, notiert mit $V(\sigma 1, ..., \sigma m)$. Dabei ist σk eine Sequenz von Jobs der Task $Task_k$.

Für den Processor Demand des Szenario $V(\sigma 1, ..., \sigma m)$ ist der Processor Demand die Summe der Processor Demands für die einzelnen Meldungssequenzen:

$$PD_{V(\sigma 1, ..., \sigma m)}(t_1, t_2) = PD_{\sigma 1}(t_1, t_2) + PD_{\sigma 2}(t_1, t_2) + ... + PD_{\sigma m}(t_1, t_2)$$

'Processor Demand Bound'-Funktion - PDB(L)

DEFINITION $PDB_k(L)$ ist der maximale Processor Demand, der in einem Intervall der Grösse L für eine Sequenz von Jobs der Task *Task*$_k$ aufgebracht werden muss.

Für eine sporadische Task *Task*$_k$ mit $D_k = T_k$ ist $PDB_k(L)$ einfach zu bestimmen:

$PDB_k(L) = L / T_k * C_k$ L / T_k ausgeführte Jobs im Worst Case (Ganzzahldivision)

Für eine sporadische Task mit Deadline-Intervallen verschieden von der Minimalperiode ist $PDB_k(L)$ schwieriger zu bestimmen (siehe unten).

Processor Demand Bound eines Task-Systems

Für ein Task-System *TS* mit n Tasks ist der maximale Processor Demand Bound in einem Intervall L durch die entsprechende Summe Processor Demand Bounds der einzelnen Tasks gegeben:

$$PDB_{TS}(L) = PDB_1(L) + ... + PDB_n(L)$$

Das 'Processor Demand'-Kriterium

Nach einem Theorem von S. K. Baruah [15] kann die Schedulability (zeitliche Planbarkeit) eines hybriden Task-Systems (sporadische und periodische Tasks) mit dem 'Processor Demand'-Kriterium überprüft werden. Dabei wird vorausgesetzt, dass das EDF-Schedulingverfahren angewendet wird.

THEOREM

Ein hybrides Tasksystem auf einem Prozessor ist mit preemptivem EDF Scheduling dann und nur dann *schedulable* wenn für alle Intervalle $L > 0$ gilt $PDB_{TS}(L) <= L$

Die Notwendigkeit des Kriteriums ist fast trivial. Im Worst Case dürfen keine Deadlines überschritten werden, d. h. $PDB_{TS}(L) <= L$ muss immer gelten. Dass das Kriterium hinreichend ist, ist hingegen viel schwieriger zu zeigen. Für das zyklusbasierte Deadline-Scheduling ist der Beweis einfacher. Er wird weiter unten im entsprechenden Abschnitt gegeben.

Das Verifikationsprinzip, auf welchem das Kriterium basiert, hat einen sehr generischen Charakter. Es wurde in der Scheduling-Literatur erfolgreich auf die unterschiedlichsten Taskmodelle angewendet. Voraussetzung war aber immer die Anwendung des EDF-Schedulingverfahrens.

6.6.2 Das Processor Demand Verfahren für Reaktive Maschinen

Die Architektur einer Embedded Unit wurde so konzipiert, dass die zeitlich nicht voraussagbaren Arbeiten in bestimmten Arbeitsphasen innerhalb eines Unit-Zyklus ausgeführt werden. Es sind dies die durch die *Reaktive Maschine* ausgeführten Reaktionen auf Ereignismeldungen. In diesem Abschnitt wird das Processor Demand Verfahren anstatt auf Jobs asynchroner Tasks auf die Verarbeitung von Ereignismeldungen durch eine Reaktiven Maschine angewendet. Der Typ einer Ereignismeldung entspricht jetzt einer sporadischen Task, und die Instanzen auftretender Ereignismeldungen entsprechen ankommenden Jobs. Im Unterschied zu asynchronen Tasks werden die Ereignismeldungen non-preemptiv verarbeitet (Run-to-Completion Semantik der Reaktiven Maschine).

Notation für eine bestimmte Embedded Unit

T : Zyklusperiode der Embedded Unit

C_{RM} : Kapazität der Reaktiven Maschine (garantierte Ausführungszeit pro Periode)

$E_1, E_2, ..., E_m$: Typen von Ereignismeldungen, welche die Reaktive Maschine verarbeitet

C_k : Worst Case Execution Time für die Verarbeitung einer Ereignismeldung vom Typ E_k

T_k : minimale Separationszeit (Minimale Periode) zweier Instanzen der Ereignismeldung E_k

D_k : Deadline-Intervall einer Ereignismeldung vom Typ E_k, wobei $D_k <= T_k$

T, C_{RM} und C_k ist in Einheiten der Prozessorzeit angegeben.

T_k und D_k sind als Vielfache der Unit-Periode T spezifiziert (Anzahl Zyklen).

C_{RM} ist garantierte Ausführungszeit der Embedded Unit pro Periode nach Abzug der statisch bestimmbaren Anteile für das I/O-Handling und die Event-Detektion.

Die 'Processor Demand'-Funktion PD()

Wir wenden jetzt das 'Processor Demand'-Verfahren sinngemäss auf die zyklusbasierte Ereignisverarbeitung in einer Embedded Unit an. Es wird der Einfachheit halber vorausgesetzt, dass die Embedded Unit streng-periodisch ausgeführt wird, d. h. eine Ereignismeldung wird in einem Zyklus nur noch verarbeitet, wenn im aktuellen Zyklus genügend Prozessorzeit zur Verfügung steht. Das 'Processor Demand'-Verfahren kann vergleichbar auch bei quasi-periodischer Ausführung einer Embedded Unit angewendet werden, es müssen aber aufgrund der möglichen Unschärfen beim Einhalten der Zykluszeit bei den Ungleichungen kleine Toleranzen eingeführt werden.

Für ein im Zyklus a detektiertes Ereignis wird eine entsprechende Ereignismeldung $E_k(a)$ erzeugt. Das Deadline-Intervall für die Verarbeitung ist mit D_k in Anzahl Zyklen spezifiziert. Auch die minimale Separationszeit T_k zweier Instanzen der Ereignismeldung E_k ist in Anzahl Zyklen gegeben, und es wird vorausgesetzt, dass $D_k <= T_k$. Zudem ist die Worst Case Execution Time C_k einer Verarbeitung kleiner als die Zyklusperiode T, und sie ist in Einheiten der Prozessorzeit ausgedrückt.

Für zeitliche Sequenzen $\sigma = (E_k(a_1), E_k(a_2), ...,)$ von Ereignismeldungen vom Typ E_k wird zuerst wieder die *Processor-Demand-Funktion* $PD_\sigma(z_1, z_2)$ definiert. Dabei sind z_1 und z_2 zwei gewählte Zykluszeiten (nicht-negative Zahlen) und a_1, a_k, ..., die Zykluszeiten der Detektion der entsprechenden Ereignisinstanzen.

DEFINITION **$PD_\sigma(z_1, z_2)$**

$PD_\sigma(z_1, z_2)$ ist die Prozessorzeit, die für die Verarbeitung derjenigen Meldungen benötigt wird, die im Intervall $[z_1, z_2[$ auftreten und auch in diesem Intervall verarbeitet werden müssen.

Bemerkung: Der Zyklus z_2 gehört nicht mehr zum halboffenen Intervall $[z_1, z_2[$.

oder formal ausgedrückt:

$$PD_\sigma(z_1, z_2) = n * C_k$$

Dabei ist n die Anzahl Ereignismeldengen $E_k(a_j)$ mit $a_j \in [z_1, z_2[$ und $a_j+D_k \in [z_1, z_2[$

(C_k = Worst Case Execution Time, D_k = Deadline-Intervall)

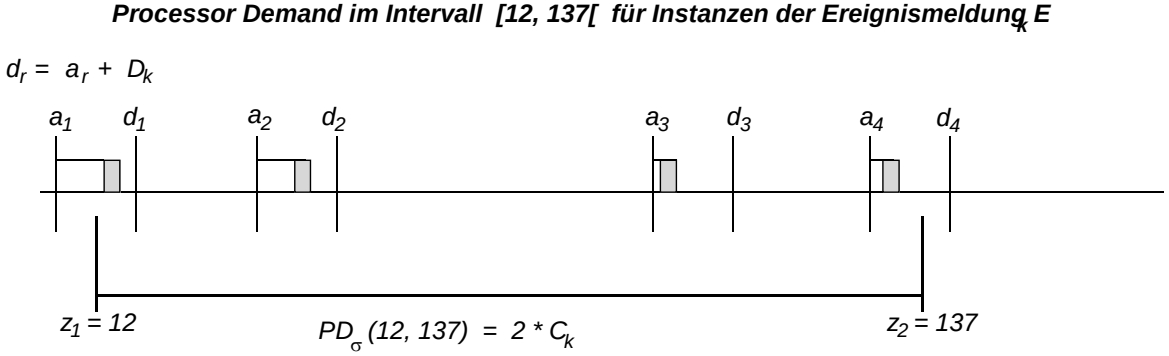


Abb. 86: Processor Demand im Intervall $[z_1, z_2[$: $PD_\sigma(z_1, z_2) = 2 * C_k$

In *Abbildung 86* ist eine Sequenz von 4 Ereignismeldungen vom Typ E_k dargestellt. Nur die zweite und dritte Meldung tragen zum Processor Demand für das Intervall $[z_1, z_2[$ bei. Die erste Meldung wird vor diesen Intervall erzeugt, und die vierte hat ihre Deadline ausserhalb des Intervalls.

'Processor-Demand'-Funktion für beliebige Szenarien

Ein allgemeines Szenario von Ereignismeldungen ist durch m verflechtete Meldungssequenzen $V(\sigma 1, ..., \sigma m)$ gegeben. Dabei ist σk eine Sequenz von Meldungsinstanzen des Typs E_k.

Für den Processor Demand des Szenario $V(\sigma 1, ..., \sigma m)$ ist der Processor Demand die Summe der Processor Demands für die einzelnen Meldungssequenzen:

$$PD_{V(\sigma 1, ..., \sigma m)}(z_1, z_2) = PD_{\sigma 1}(z_1, z_2) + PD_{\sigma 2}(z_1, z_2) + ... + PD_{\sigma m}(z_1, z_2)$$

'Processor Demand Bound'-Funktion - PDB(L)

DEFINITION $PDB_k(L)$ ist der maximale Processor Demand, der in einem Intervall von L Zyklen für Meldungen vom Typ E_k aufgebracht werden muss.

Bemerkung.

$PDB_k(z_1, z_2)$ ist translationsinvariant: Für jedes z gilt $PDB_k(z, z + L) = PDB_k(0, L)$

Damit genügt es den *Processor Demand Bound* als Funktion der Intervallgrösse L zu definieren.

Bestimmung des Processor Demand Bound für sporadische Ereignisse vom Typ E_k

Der *Worst-Case* $PDB_k(L) = PD_{\mathbf{V}(\sigma 1, ..., \sigma m)}(0, L)$ tritt ein, wenn alle Ereignisse am Anfang des Intervalls L auftreten und sich mit minimaler Separationszeit T_k wiederholen.

Behauptung

$PDB_k(L) = (L + T_k - D_k) / T_k * C_k$ ("/" bedeutet Ganzzahl-Division)

Bei der Bestimmung des *Processor Demand Bound* müssen die in *Abbildung 87* gezeigten zwei Fälle für L unterschieden werden. Im ersten Fall enthält L das letzte Deadline-Intervall vollständig. Im zweiten Fall ist das letzte Deadline-Intervall nicht mehr ganz im Intervall L enthalten.

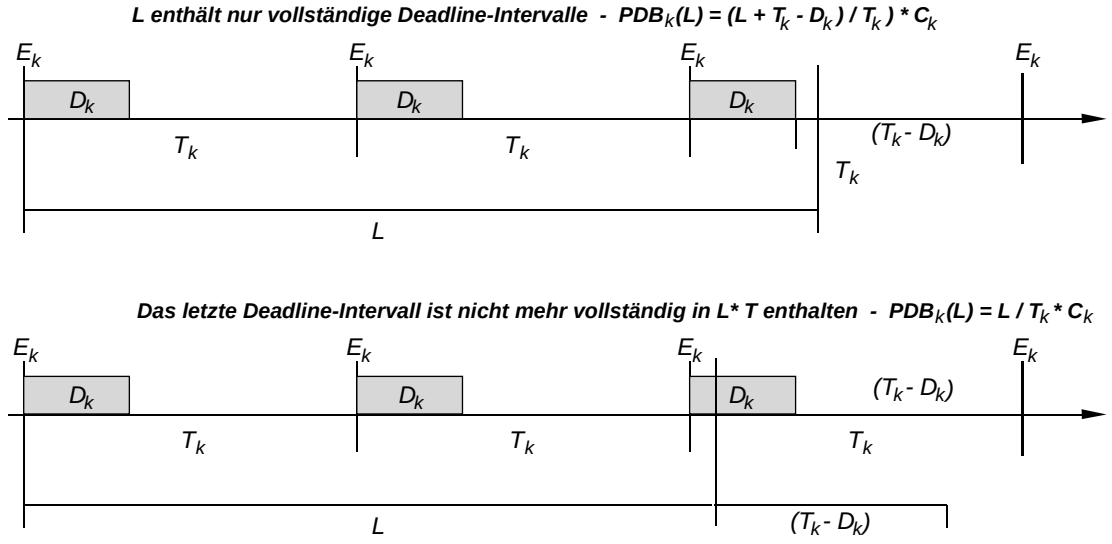


Abb. 87: Processor Demand Bound für Ereignismeldungen vom Typ E_k

Herleitung

Im ersten Fall ist das letzte Deadline-Intervall D_k ganz im Intervall L enthalten, d. h.

$PDB_k(L) = (L + (T_k - D_k)) / T_k) * C_k$

Damit die Ganzzahl-Division die korrekte Anzahl von Ereignisverarbeitungen liefert, ist zu L das Segment $T_k - D_k$ zu addieren.

Im zweiten Fall ist das letzte Deadline-Intervall D_k nicht mehr ganz im Intervall L enthalten.

$PDB_k(L) = L/T_k * C_k$ liefert damit bereits den korrekten Wert.

Wenn nun bei der Division anstatt der Dividend L der Ausdruck $L + (T_k - D_k)$ verwendet wird, ändert sich das Resultat nicht (siehe *Abbildung 87*), d. h. wir können auch hier schreiben

$PDB_k(L) = (L + T_k - D_k)/T_k * C_k$ ***q.e.d.***

Processor Demand Bound der Reaktiven Maschine

Für die gesamte Reaktive Maschine *RM* ist der maximale Processor Demand in einem Intervall von L Zyklen durch die entsprechende Summe über alle Ereignisverarbeitungen gegeben:

$$PDB_{RM}(L) = PDB_1(L) + ... + PDB_m(L)$$
$$= (L + T_1 - D_1) / T_1 * C_1 + ... + (L + T_m - D_m) / T_m * C_m$$

Das 'Processor Demand'-Kriterium

Auf die zyklusbasierte Verarbeitung sporadischer Ereignisse durch eine Embedded Unit zugeschnitten lautet das 'Processor Demand'-Kriterium wie folgt:

THEOREM

Eine Reaktive Maschine ist mit EDF schedulable <=> für L > 0 gilt $PDB_{RM}(L) <= L * C_{RM}$

Dabei ist der Processor Demand Bound $PDB_{RM}(L)$ die maximale kumulierte Ausführungszeit, die in L Zyklen für die Ereignisverarbeitungen benötigt wird.

$L * C_{RM}$ ist die im Intervall L zur Verfügung gestellte Prozessorzeit.

Zulässige Einschränkung des Testbereichs für L

Als Testbereich für L genügen Werte, die kleiner sind als das kleinste gemeinsame Vielfache (Hyperperiode) der Separationszeiten $T_1, ..., T_M$, weil sich die Worst Case Szenarien mit dieser Hyperperiode wiederholen.

Der Testbereich kann noch weiter eingeschränkt werden: Es genügen Werte für L, die im Worst-Case Szenario eine absolute Deadline (Deadlinezyklus) sind, denn zwischen zwei sich folgenden absoluten Deadlines ändert sich der Wert des *Processor Demand Bound* beim Worst-Case Szenario nicht.

Beweis des Processor Demand Kriteriums für die zyklusbasierte Ereignisverarbeitung

Wir zeigen

RM ist *nicht* schedulable <=> Es gibt ein L mit $PDB_{RM}(L) > L * C_{RM}$

1. RM ist nicht schedulable <= Es gibt ein L mit $PDB_{RM}(L) > L * C_{RM}$

Beim Worst Case Szenario, in dem alle Ereignisse am Anfang des Intervalls L auftreten und sich mit minimaler Separationszeit wiederholen ist der Processor Demand gleich $PDB_{RM}(L)$. Das bedeutet, dass nicht alle im Intervall L detektierten Ereignisse, die auch ihre Deadline im Intervall L haben, im Intervall L verarbeitet werden können. Damit wird mindestens eine Deadline überschritten.

2. RM ist nicht schedulable => Es gibt ein L mit $PDB_{RM}(L) > L * C_{RM}$

Wir zeigen:

> Wenn es eine Sequenz detektierter Ereignisse σ gibt, bei der bei der Verarbeitung eine Deadline nicht eingehalten werden kann, dann gibt es ein Intervall B von Zyklen mit $PDB_\sigma(B) > B * C_{RM}$. Damit ist aber auch $PDB_{RM}(B) > B * C_{RM}$

Das Intervall B heisst *Deadline-Busy-Period* und wird im folgenden aufgrund einer angenommenen Deadline-Überschreitung konstruiert.

Sei a_x der Detektionszyklus der Ereignismeldung E_x mit Deadline-Miss und D_x das verpasste Deadline-Intervall. Das Szenario ist in *Abbildung 88* dargestellt.

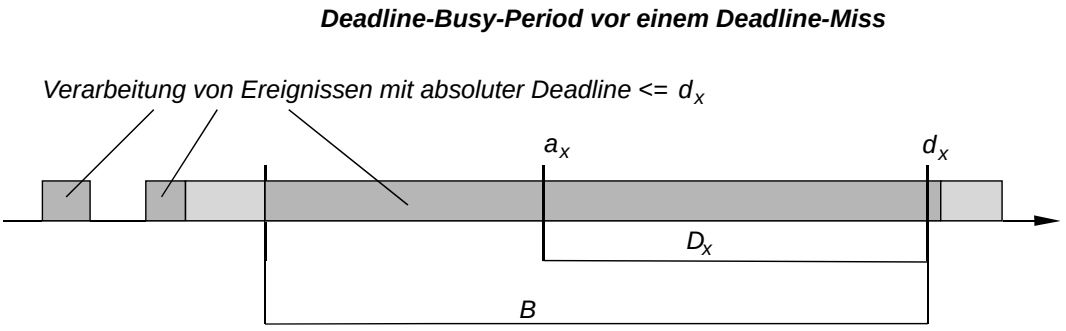


Abb. 88: Verpasste Deadline d_x und zugehörige Deadline-Busy-Period B

Wird für die Verarbeitung einer Ereignismeldung die Deadline überschritten, so gibt es direkt vor der verpassten Deadline immer ein maximales Intervall B von Zyklen, in denen andauernd Ereignismeldungen verarbeitet worden sind, deren Deadline kleiner oder gleich der verpassten Deadline ist (EDF-Schedulingverfahren). Dieses Intervall wird als *Deadline-Busy-Period* bezeichnet.

Es gilt sicher $B >= D_x$

weil alle Ereignisse E_k, die in $[a_x, d_x[$ verarbeitet werden, eine Deadline $d_k <= d_x$ haben.

Zudem sind all diese Ereignisse auch in B detektiert worden, denn im ersten Zyklus vor B gab es entweder eine Idle-Phase, in der keine Ereignisse zu verarbeiten waren, oder es war mindestens ein Ereignis E_j mit einer absoluten Deadline $d_j > d_x$ verarbeitet worden. Das zeigt aber, dass keines der Ereignisse, die in B verarbeitet worden sind, vor B detektiert worden ist. Wegen des EDF-Schedulingverfahren wäre es nämlich vor dem Ereignis E_j verarbeitet worden. Damit geben alle in B detektierten Ereignisse einen Beitrag zum Processor Demand $PD_\sigma(B)$.

Weil aber das Ereignis E_x nicht im Intervall B verarbeitet werden kann, gilt $PD_\sigma(B) > B * C_{RM}$

q.e.d.

6.6.3 Vereinfachung des 'ProcessorDemand'-Kriteriums für sequentiell abhängige Ereignisse

Die oben erläuterte Herleitung für den Processor Demand einer Reaktiven Maschine geht davon aus, dass alle Ereignisse gleichzeitig auftreten können, was in realen Systemen aufgrund sequentieller Abhängigkeiten nicht zutrifft. Es ist klar, dass Ereignisse verschiedener Prozesse als parallele Ereignisse zu behandeln sind, die auch im selben Zyklus detektiert werden können. Die Ereignisse eines bestimmten Prozesses hingegen können in der Regel nicht gleichzeitig auftreten. Das ist z. B. bei den Ereignissen *Closed, NotClosed, Opened, NotOpened* der früher betrachteten Türsteuerung der Fall. Werden solche Ereignisse bei der Bestimmung des Processor Demands aber als parallele Ereignisse behandelt, entstehen viel zu grosse Worst Case-Werte.

Wir nehmen im Folgenden an, dass alle Ereignisse eines Prozesses zeitlich separiert auftreten. Die entsprechenden Ereignismeldungen werden als Meldungen eines sequentiellen Ereignismeldungskanal erzeugt. Falls es Ereignismeldungen eines Kanals gibt, die gleichzeitig auftreten können, müsste ein solcher Kanal für die Bestimmung des Processor Demand in parallele Teilkanäle zerlegt werden.

Damit wird eine grosse Vereinfachung möglich, indem man pro Ereignismeldungskanal alle Meldungen als sequentielle Instanzen eines gemeinsamen Kanal-Meldungstyps behandeln kann.

Die Ereignismeldungskanäle bezeichnen wir mit CHr, dabei hat r die Werte 1, ... , R. Für die Meldungen eines Kanals CHr, die nach Voraussetzung zeitlich separiert auftreten, verwendet man den gemeinsamen Meldungstyp E_{CHr} mit den folgenden Scheduling-Attributen:

T_{CHr} ist das Minimum der Separationszeiten der Ereignismeldungen von Kanal CHr

D_{CHr} ist das Minimum der Deadline-Intervalle der Ereignismeldungen von Kanal CHr

C_{CHr} ist das Maximum der Ausführungszeiten der Ereignismeldungen von Kanal CHr

Alle Meldungen eines Kanals können jetzt als Instanzen des gemeinsamen Meldungstyps E_{CHr} betrachtet werden. Damit erhalten wie ein vereinfachtes Schedulability-Kriterium für die Ereignisverarbeitungen einer Embedded Unit, das nur die Processor Demand Bound-Funktionen $PDB_{CHr}()$ für Instanzen der Kanal-Meldungen verwendet:

$$PDB_{CHr}(L) = (L + T_{CHr} - D_{CHr}) / T_{CH} * C_{CHr}$$

Schedulability-Kriterium

Für alle Intervalle L mit $0 < L < KGV(T_{CH1}, ..., T_{CHR})$ muss gelten

$$PDB_{CH1}(L) + ... + PDB_{CRr}(L) <= L * C_{RM}$$

($KGV(N_1, ..., N_R)$ ist das *Kleinste Gemeinsame Vielfache* der Faktoren $(N_1, ..., N_R)$)

Das vereinfachte Kriterium ist hinreichend für die Schedulability der Reaktiven Maschine einer Embedded Unit. Wird das Kriterium nicht erfüllt, ist unter Umständen die Vereinfachung zu stark. In solchen Fällen müsste die Verwendung der extremalen Scheduling-Attribute D_{CHr}, T_{CHr} und C_{CHr} analysiert werden und die Vereinfachung verfeinert werden.

Reduziertes 'Processor Demand'-Kriterium

Das vollständige Prozessor-Demand-Kriterium basiert auf verschiedenen Worst Case Annahmen:

- als Ausführungszeiten werden Worst Case Execution Times verwendet
- alle parallelen Ereignisse können gleichzeitig auftreten
- sporadische Ereignisse treten so häufig wie möglich auf (minimale Separationszeit als Periode)

Dass viele parallele Ereignisse quasi gleichzeitig auftreten (Ereignislawinen) muss zum Beispiel erwartet werden, wenn bei der gesteuerten Anlage ein technisches Problem entstanden ist. Hingegen ist kaum zu erwarten, dass diese Ereignisse in den im Kriterium betrachteten Intervallen wiederholt mit der minimalen Separationszeit auftreten.

Als entsprechend reduzierte Anwendung des Processor Demand Kriteriums (weniger pessimistisch, dafür nicht hinreichend) kann der Fall betrachtet werden, in dem alle parallelen Ereignisse gleichzeitig auftreten, aber nur genau ein Mal in den im Kriterium betrachteten Intervallen. Wenn wir auch annehmen, dass in einem Kanal nur sequentiell separierte Ereignismeldungen auftreten, können wir zudem die oben definierten Extremalattribute für Kanalmeldungen verwenden. Wenn das Kriterium erfüllt ist, muss nur noch in unwahrscheinlichen Fällen damit gerechnet werden, dass Deadline-Überschreitungen möglich sind.