Ein objektorientierter Mikrokernel für den i960 in Oberon-2

D I P L O M A R B E I T

zur Erlangung des akademischen Grades

Diplom-Ingenieur

in der Studienrichtung
Informatik

Eingereicht von

Christoph Steindl

am Forschungsinstitut für Mikroprozessortechnik
der Johannes Kepler Universität Linz,
bei **o. Univ.-Prof. Dr. Jörg Mühlbacher**

Linz, im März 1995

Übersicht und Ziele

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, im März 1995

Christoph Steindl

meinen Eltern

Margareta und Alois

Danksagung

Mein Dank gilt in erster Linie meiner Familie, die mir das Studium in Linz ermöglicht hat und mich in jeder Hinsicht unterstützte.

Gedankt sei an dieser Stelle aber auch meinem Betreuer, Herrn Dipl.-Ing. Gerhard Eschelbeck, und Herrn o. Univ.-Prof. Dr. Jörg Mühlbacher, die mir einerseits bei Problemen halfen, mich andererseits aber auch frei arbeiten ließen, ebenso Herrn o. Univ.-Prof. Dr. Hanspeter Mössenböck, der mich einige Schritte weiter in Richtung eines klaren und einfachen Programmierstils brachte.

Danken möchte ich auch Richard Fanta, der mir seine C++ Implementierung des Mikrokernels „Objix“ zur Verfügung stellte und Professor Frantisek Plasil.

Schlußendlich möchte ich noch der Professorenschaft des BG Zwettl danken, die mich auf meine Zeit an der Johannes Kepler Universität Linz bestens vorbereitet hat.

Zusammenfassung - Abstract

Zusammenfassung

Das Ziel dieser Diplomarbeit war ein objektorienterter, echtzeitfähiger Betriebssystemkern für den 32-Bit Mikrocontroller 80960 von Intel.

Durchgeführt wurde die Arbeit in Oberon-2 mit dem von Richard Hable entwickelten Crosscompiler für den i960. Dieser Crosscompiler sollte im Rahmen dieser Diplomarbeit auf seine Zuverlässigkeit und Effizienz getestet werden.

Als Grundlage dient ein Microkernel für den 8086, der von Richard Fanta in C++ geschrieben worden war.

Nach der Portierung der bestehenden C++ Version auf Oberon-2 und den i960 wurde das Problem des Prozeßwechsels gelöst, zusätzlich wurde ein echtzeitfähiges Schedulingverfahren (präemptiv und prioritätengesteuert) implementiert.

Unter der Verwendung dieses Mikrokernels ist es möglich, Echtzeitanwendungen in einer Hochsprache zu programmieren und die Vorteile von Parallelität in konventionelle Programme einzubauen. Über eine klare Programmierschnittstelle hat man Zugriff auf die Funktionalität des Kernels.

Abstract

The goal of this diploma thesis is an object oriented real time microkernel for Intel´s 32-Bit microcontroller 80960.

The code is written in Oberon-2 using the crosscompiler for the i960 developed by Richard Hable at the FIM. This crosscompiler should be tested for its reliability and efficiency while programming the microkernel in question.

As a basis a microkernel for the 8086 was used that had been written by Richard Fanta in C++.

After having ported the given C++ code to Oberon-2 and onto the i960 the problem of switching between processes was solved and a real time scheduling mechanism was implemented (preemptive and priority driven).

Using this microkernel now it is possible to develop real time applications in a high level language and bring in the advantages of parallelism into conventional programs. A clear programming interface provides access to the functionality of the kernel.

Übersicht und Ziele

Übersicht

Im Kapitel „Grundlagen“ werden Begriffe wie Mikrocomputer, Mikrocontroller, Echtzeit-Betriebssystem und Mikrokernel besprochen. Ebenso erfolgt eine kurze Einführung in Oberon-2. Abschließend wird der Mikroprozessor i960 etwas näher erklärt.

Im Kapitel „Implementierung im Überblick“ werden der Programmentwicklungsprozeß auf dem i960 sowie der Entwicklungsprozeß dieser Diplomarbeit beschrieben.

Im Kapitel „Scheduling und Prozeßwechsel“ werden kurz die Grundbegriffe des Schedulings wiederholt. Danach erfolgt in etwas breiterem Ausmaß eine Erklärung des Prozeßwechsels auf dem i960.

Im Kapitel „Implementierung im Detail“ werden wichtige Prozeduren im Detail beschrieben oder direkt im Quellcode angegeben. Zusätzlich findet man dort eine Aufrufhierarchie, die Aufschluß gibt, welche (system-internen) Prozeduren wann von wem aufgerufen werden.

Im Kapitel „Anwendungen“ werden einige Beispielanwendungen im Quellcode mit Erklärungen abgedruckt.

Im Kapitel „Bewertung und Ausblick“ erfolgt eine Bewertung der Arbeiten mit Bezugnahme auf die gesteckten Ziele, weiters werden mögliche Fortsetzungsarbeiten angesprochen.

Im Kapitel „User Manual“ findet man eine prägnante Beschreibung der Programmierschnittstelle (besonders interessant für den Programmierer).

Im Anhang finden sich dann noch eine komplette Programmierschnittstelle des Mikrokernels, die Syntax von Oberon-2, das Literaturverzeichnis, ein Index, das Abbildungs- und das Tabellenverzeichnis.

Ziele

Für die Implementierung des Mikrokernels habe ich mir folgende Ziele gesetzt:

- Objektorientiertheit, Erweiterbarkeit: der Kernel sollte im Sinne der objektorientierten Programmierung erweiterbar sein, bestehender Code soll in Erweiterungen wiederverwendet werden können.
- Echtzeitfähigkeit: die mit dem Kernel erstellten Anwendungsprogramme sollen zum Beispiel für Robotersteuerungen einsetzbar sein.
- Lesbarkeit, Verständlichkeit: der Kernel soll im Quellcode lesbar und leicht verständlich sein.
- Einfachheit: die verwendeten Algorithmen und Konzepte sollen einfach sein (wobei natürlich trotzdem auf Effizienz geachtet werden muß).
- Portabilität: der Kernel soll mit wenig Aufwand an andere Prozessorumgebungen angepaßt werden können.

Inwiefern ich die Ziele erreicht habe, werde ich im Kapitel „Bewertung und Ausblick“ näher beschreiben.

Grundlagen

Mikrocomputer / Mikrocontroller

Die heuzutage angebotenen PC-Systeme haben alle den gleichen schematischen Aufbau. Der wichtigste Teil innerhalb eines PC-Systems ist der Mikroprozessor (central processing unit, CPU), der neben den Rechenoperationen auch die zeitlichen Abläufe (timing and control) innerhalb des Rechnersystems festlegt. Ein weiterer wesentlicher Baustein eines Rechnersystems ist der Speicherbereich, den man in internen und externen Speicher unterteilen kann. Der interne Speicher kann von der CPU aus direkt angesprochen werden und ist in Festwertspeicher (read only memory, ROM) und Arbeitsspeicher (random access memory, RAM) unterteilt. Im Festwertspeicher befindet sich das Betriebssystem (oder zumindest Teile davon). Im Arbeitsspeicher können Informationen abgespeichert (Schreib-Vorgang) und auch wieder abgerufen (Lese-Vorgang) werden. Die Verbindung zwischen der Zentraleinheit und der Außenwelt wird durch Schnittstellen, Interface-Schaltungen, hergestellt, an welche verschiedenste Geräte (Monitor, Tastatur, Drucker, Maus, Festplatte, Diskettenlaufwerk, CD-ROM,...) angeschlossen werden. Die Interface-Schaltungen übernehmen auch die elektrische Anpassung zwischen Zentraleinheit und Peripherie. Der externe Speicher (auch Massenspeicher genannt, z.B. Festplatte, Diskette, Streamer) wird zur dauerhaften Abspeicherung von Daten und Programmen verwendet.

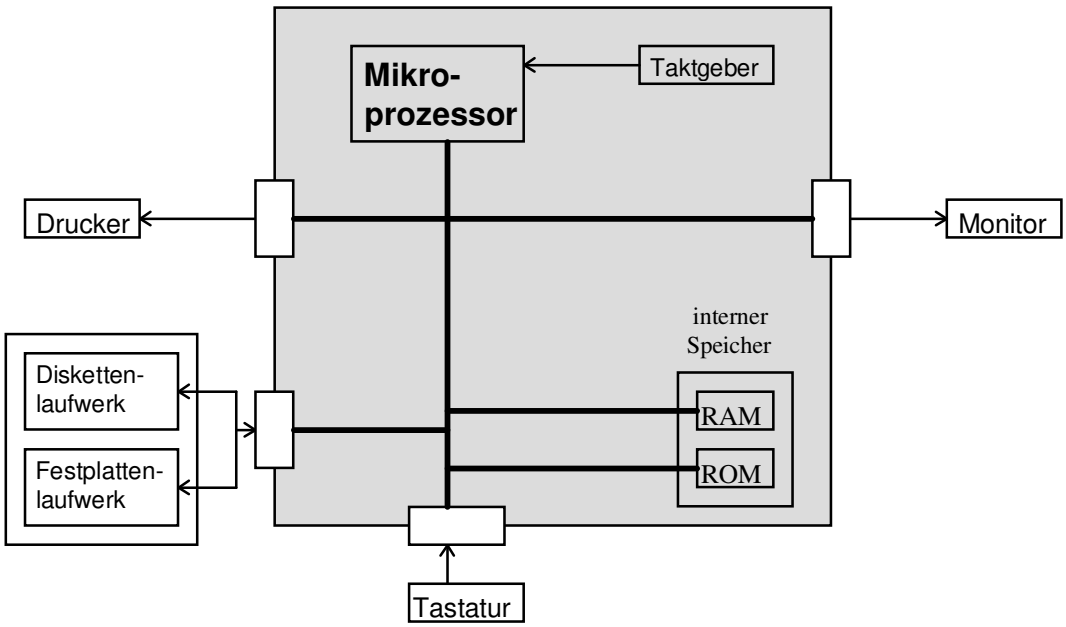

Abbildung 1, beispielhafter Aufbau eines Mikrocomputer-Systems

Mikrocontroller sind spezielle Mikroprozessoren und werden in der Meß- und Steuerungstechnik, der Automatisierungstechnik und der Automobil-Elektronik stark eingesetzt. Im Unterschied zu Mikroprozessoren, die durch zusätzliche externe Bausteine erst zu einem funktionsfähigen System wachsen, sind Mikrocontroller zusätzlich zur CPU mit internem RAM und manchmal auch internem ROM ausgestattet. Ebenfalls auf dem Chip

integriert sind z.B. Timer, serielle Schnittstellen oder Analog-Digital-Wandler, die als on-chip-devices bezeichnet werden. Als Single-Chip-Lösungen können sie sehr individuell an die geforderten Aufgaben angepaßt werden. Moderne Mikrocontroller zeichnen sich durch sehr flexible Interrupt-Möglichkeiten aus, um auf externe Signale - unabhängig vom laufenden Programm - in möglichst kurzer Zeit zu reagieren. Die Ausstattung eines Controller-Typs mit internem ROM ermöglicht, ein Programm fest einzubrennen; die Applikation des Controllers ist damit eindeutig festgelegt. Diese Form wird als “Embedded-Controller” bezeichnet.

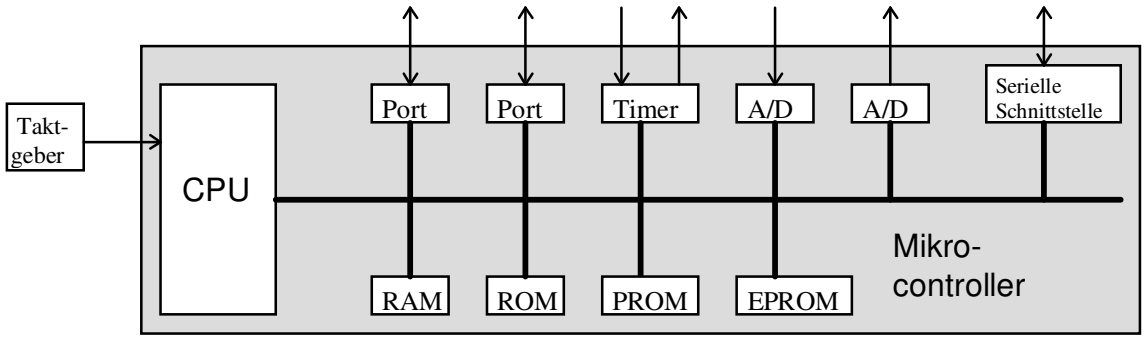

Abbildung 2, beispielhafter Aufbau eines Mikrocontroller-Systems

Echtzeit-Betriebssystem

Im Bereich der Mikrocontrolleranwendungen findet man oft folgende Anforderungen: unter harten zeitlichen Bedingungen bringen Sensoren Daten zum Prozessor, der diese analysieren und gegebenenfalls über Aktoren geeignet reagieren muß. Beispielanwendungen sind Steuerungen für wissenschaftliche Experimente, medizinische Visualisierungsgeräte, industrielle Robotersteuerungen, Brennstoff-Einspritz-Anlagen in Autos etc. Für diese Anwendungen existieren oft fixe, zeitlich wohldefinierte Schranken, innerhalb deren die Abarbeitung geschehen muß.

Es gibt zwei Arten von Echtzeit-Anforderungen: harte und leichte. Bei harten Echtzeit-Anforderungen wird garantiert, daß kritische Aufgaben rechtzeitig (innerhalb einer festgelegten Zeitdauer) abgearbeitet werden. Dazu müssen alle Verzögerungen im System begrenzt sein, angefangen beim Zugriff auf gespeicherte Daten. Diese Anforderungen schränken die Dienste ein, die ein solches Betriebssystem zur Verfügung stellen kann: meist ist der Hintergrundspeicher limitiert oder fehlt ganz, virtuellen Speicher gibt es nicht, und Daten werden meist im ROM bzw. RAM abgelegt. Bei leichten Echtzeit-Anforderungen erhalten zeitkritische Aufgaben Priorität gegenüber anderen. Auch hier müssen die Verzögerungen im Betriebssystem selbst beschränkt sein, die Einhaltung einer Deadline, bis zu der eine Aufgabe abgearbeitet sein muß, wird aber nicht garantiert.

Mikrokernel

Ein Mikrokernel ist ein Basisbetriebssystem, das die notwendige Funktionalität für darauf aufbauende Bibliotheken liefert, um parallele Prozesse, deren Synchronisation und

Kommunikation zu ermöglichen. Weiters verwaltet der Mikrokernel alle Systemressourcen, auf die man nur über die geeigneten Zugriffsfunktionen des Kernels zugreifen kann.

Objektorientierte Programmierung (OOP) im Bereich von Echtzeit-Betriebssystemen

Im folgenden werde ich einige Punkte für und gegen den Einsatz von OOP im Bereich von Echtzeit-Betriebssystemen geben, die teilweise auf einen Workshop auf der OOPSLA 90 zurückgehen [OOP90].

Vorteile sind:

- bessere Modellierung des Problembereichs
- verbesserte Produktivität (durch Prototyping)
- verläßlicherer und wartbarerer Code (weil verständlicher)
- weniger Fehler bei Speicherverwaltung durch Garbage Collector (wenn man einen solchen trotz Echtzeitanforderungen einsetzen kann. Fehler bei der Speicherverwaltung sind bei bisherigen Programmen die größte Fehlerquelle.)
- Wiederverwendung bestehenden Codes in Erweiterungen („Delta-Programmierung“)
- leichte Anpaßbarkeit bestehender Lösungen an neue Anforderungen

Nachteile sind:

- Effizienzprobleme („dynamic dispatch“): Effizienzprobleme sind meist auf schlechte Algorithmen zurückzuführen und nicht auf die verwendete generelle Programmiertechnik (OOP oder Nicht-OOP). Allerdings veranlaßt OOP den Programmierer im allgemeinen dazu, generische Algorithmen zu schreiben, weil ein hoher Wiederverwendungsgrad angestrebt wird. Da diese Algorithmen aber meist nicht auf die spezifischen Probleme zugeschnitten sind, sind sie oft nicht zeitoptimal.
- Probleme mit harten Echtzeit-Anforderungen
- Garbage Collection dauert nicht-vorhersagbar lange.

Einführung in Oberon-2

Im folgenden werden einige der Merkmale von Oberon-2 näher vorgestellt, eine detaillierte und vollständige Beschreibung der Sprache findet man in [MOW92].

Oberon-2 ist eine universelle Programmiersprache in der Tradition von Oberon, Modula-2 und Pascal. Sie bietet Blockstruktur, Modularität, getrennte Kompilation, strenge statische Typprüfung (auch über Modulgrenzen hinweg), Typerweiterung, typgebundene Prozeduren sowie „Infomation Hiding“.

Durch die Typerweiterung wurde Oberon-2 eine objektorientierte Sprache. Ein Objekt ist in Oberon-2 ein Exemplar eines abstrakten Datentyps, der aus eigenen Daten (dem Zustand) und Prozeduren, die mit diesen Daten arbeiten, besteht.

Deklaration von Bezeichnern

Bezeichner, die in einem Modul deklariert sind, können mit einer Exportmarkierung versehen werden ("*" oder "-"). Ein Bezeichner x, der von dem Modul M exportiert wird, kann von anderen Modulen aus verwendet werden, wenn diese das Modul M importieren. Der Bezeichner kann über qualifizierten Import mit M.x angesprochen werden. Bezeichner, die mit einem "-" markiert sind, können von importierenden Modulen aus nur gelesen werden.

Arrays

Gültige Indizes in einem Array sind wie in C im Bereich von 0 bis LEN(array) - 1.

Records

Ein Record ist eine Datenstruktur, die aus einer festen Anzahl von Elementen (auch Feldern genannt) beliebigen Typs besteht.

Wenn ein Record exportiert wird, müssen Elemente, die außerhalb des deklarierenden Moduls sichtbar sein sollen, ebenfalls exportiert werden. Unmarkierte Elemente nennt man private Datenfelder.

Recordtypen sind erweiterbar, d.h. ein Recordtyp kann als Erweiterung eines anderen Recordtyps deklariert werden. Ein Beispiel dafür ist:

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

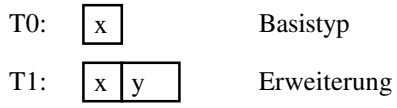

Abbildung 3, Typerweiterung

T1 ist eine (direkte) Erweiterung von T0, T0 ist ein (direkter) Basistyp von T1. Ein erweiterter Typ wie T1 besteht aus den Feldern des Basistyps (hier T0) und den neu deklarierten Feldern.

Zeiger

Zeiger müssen in Oberon-2 entweder auf Records oder Arrays zeigen. Zeiger erben die Typerweiterungen der Objekte, auf die sie zeigen: wenn T1 eine Erweiterung von T0 ist und P1 ein POINTER TO T1 ist, dann ist P1 auch eine Erweiterung von POINTER TO T0.

Die Speicherreservierung für Variablen eines Zeigertyps erfolgt mit NEW(p). Die Speicherreservierung für n-dimensionale offene Arrays (Arrays, deren Dimensionsgrenzen

nicht zur Compilierungszeit bekannt sind) erfolgt mit NEW(p, e0, ..., en-1), wobei die Ausdrücke e0 bis en-1 die Anzahl der Elemente der jeweiligen Dimensionen sind.

Der Zugriff auf die referenzierte Variable erfolgt durch Dereferenzierung: “p^”. Die Bezeichner p^.f und p^[e] können als p.f und p[e] abgekürzt werden (“implizite Dereferenzierung”).

Ein Typ-Guard v(T) stellt sicher, daß der dynamische Typ von v T ist (oder eine Erweiterung von T), d.h. die Programmausführung wird unterbrochen, wenn der dynamische Typ von v nicht T (oder eine Erweiterung von T) ist. Typ-Guards kann man anwenden auf VAR-Record-Parameter und auf Zeigervariablen.

Der Ausdruck v IS T steht für “der dynamische Typ von v ist T” (oder eine Erweiterung von T).

Prozedurtyp

Variablen von diesem Typ haben als Wert eine Prozedur (oder NIL). Bei der Zuweisung einer Prozedur P an eine Prozedurvariable des Typs T müssen die formalen Parameter von P und T übereinstimmen. Die zugewiesene Prozedur darf keine vordefinierte, typgebundene oder lokale Prozedur sein.

Dynamischer und statischer Typ

Record- und Zeigervariablen haben sowohl einen statischen Typ (aufgrund ihrer Deklaration) als auch einen dynamischen Typ (der Typ, den sie zur Laufzeit annehmen). Für Zeigervariablen und VAR-Record-Parameter kann der dynamische Typ eine Erweiterung des statischen Typs sein. Der statische Typ bestimmt, auf welche Elemente man zugreifen kann. Der dynamische Typ bestimmt, welche typgebundene Prozedur aufgerufen wird („dynamic dispatch“).

Mengenvariablen und Mengenoperatoren

Mengenoperatoren sind auf Operanden des Typs SET anwendbar und liefern selber wieder ein Ergebnis vom Typ SET. Mengenkonstruktoren sind eine Liste von Elementen innerhalb geschwungener Klammern. Die einzelnen Elemente können Bereiche sein (z.B. "a..b”) und müssen im Bereich 0..MAX(SET) sein. Als Operatoren sind definiert: „+“ für Vereinigung, „-“ für Differenz (x - y = x * (-y)), „*“ für Schnittmenge, „/“ für symmetrische Differenz (x / y = (x - y) + (y - x)). Der Ausdruck x IN s steht für “x ist ein Element von s”.

Statements

IfStatement = IF Expression THEN StatementSequence {ELSIF Expression THEN
  StatementSequence} [ELSE StatementSequence] END.
CaseStatement = CASE Expression OF Case {"|" Case} [ELSE StatementSequence] END.
Case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE Expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL Expression.

ForStatement = FOR ident ":=" Expression TO Expression [BY ConstExpression] DO
    StatementSequence END.
LoopStatement = LOOP StatementSequence END.

Loop Schleifen sind besonders nützlich, wenn man mehrere Austrittspunkte hat, oder, wenn die Austrittsbedingung in der Mitte einer Befehlssequenz geprüft werden muß. Der Sprung aus einer Loop Schleife heraus erfolgt mit EXIT. Eine Loop Schleife ohne EXIT ist eine Endlosschleife.

Der Rücksprung aus einer Prozedur erfolgt mit RETURN, gefolgt von einem Ausdruck, wenn die Prozedur eine Funktionsprozedur ist.

WithStatement = WITH Guard DO StatementSequence {"|" Guard DO StatementSequence}
    [ELSE StatementSequence] END.
Guard = Qualident ":" Qualident.

Mit einem With Statement kann man abhängig vom Ergebnis des Typ-Tests verschiedene Befehlssequenzen ausführen, in denen auf die getestete Variable ein entsprechender Typ-Guard angewendet wird.

Wenn v ein VAR-Record-Parameter oder eine Zeigervariable mit statischem Typ T0 ist, hat das Statement WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END die folgende Bedeutung: Wenn der dynamische Typ von v T1 ist, wird die Befehlssequenz S1 ausgeführt, wobei v als Variable vom statischen Typ T1 betrachtet wird; wenn der dynamische Typ von v T2 ist, wird die Befehlssequenz S2 ausgeführt, wobei v als Variable vom statischen Typ T2 betrachtet wird; ansonsten wird S3 ausgeführt. T1 und T2 müssen hierbei Erweiterungen von T0 sein. Wenn kein Typ-Test erfüllt ist und kein Else spezifiziert ist, wird das Programm abgebrochen.

Prozeduren

Typgebundene Prozeduren haben einen zusätzlichen Parameter, den Empfänger (receiver). Der Empfänger kann entweder ein Referenzparameter eines Recordtyps T sein oder ein Werteparameter eines Typs POINTER TO T (wobei T ein Recordtyp ist). Prozeduren, die an den Typ T0 gebunden sind, sind implizit auch an beliebige Typen T1, die Erweiterungen von T0 sind, gebunden. Eine Prozedur kann allerdings für erweiterte Typen überschrieben werden, wobei die Liste der formalen Parameter nicht geändert werden kann. Typgebundene Prozeduren werden immer entsprechend dem dynamischen Typ des Empfängers aufgerufen. Wenn eine Prozedur im erweiterten Typ überschrieben wurde, kann man mit “r.P^”, einem sogenannten Super-Call, die Prozedur des Basistyps aufrufen.

Die Parameter einer Prozedur können Werte- oder Referenzparameter (VAR-Parameter) sein. Werteparameter sind lokale Variablen, denen der entsprechende Wert des aktuellen Parameters zugewiesen wird. Referenzparameter entsprechen den aktuellen Parametern.

Module

Module sind Sammlungen von Deklarationen von Konstanten, Typen, Variablen und Prozeduren gemeinsam mit einer Folge von Statements zur Initialisierung von Variablen. Module sind die Kompilationseinheiten.

In der Import-Liste gibt man die Namen der importierten Module an. Wenn ein Modul A von einem Modul M importiert wird und A den Bezeichner x exportiert, kann man x innerhalb von M als A.x ansprechen. Wenn A als B := A importiert wurde, kann man x innerhalb von M als B.x ansprechen. Mit diesem Mechanismus kann man für importierte Module kürzere Bezeichner vergeben.

Einige Merkmale von C++

C++ ist wie Oberon-2 eine universelle Programmiersprache, allerdings basierend auf C. Zusätzlich zu den Merkmalen von C bietet C++ Klassen, Inline-Funktionen, Überladen von Operatoren, Überladen von Funktionsnamen, Konstantentyp, Referenzen, Operatoren für die Freispeicherverwaltung, Prüfung von Funktionsargumenten und Typkonversion.

Deklarationen und Headerdateien

Bevor ein Bezeichner in einem C++ Programm verwendet werden kann, muß er deklariert worden sein; das bedeutet, daß für ihn ein Typ angegeben werden muß, der den Compiler über die Art der von ihm bezeichneten Entität informiert. Die meisten Deklarationen sind auch Definitionen: sie definieren zusätzlich die Entität, auf die sich der Name beziehen soll. In einem C++ Programm muß für jeden Namen genau eine Definition vorhanden sein, aber es können mehrere Deklarationen vorkommen, wobei alle Deklarationen bezüglich des Typs der bezeichneten Entität übereinstimmen müssen. Jede Deklaration, die einen Wert spezifiziert, ist gleichzeitig eine Definition.

Um einen Bezeichner zu deklarieren, ohne ihn zu definieren, kann man ihn mit dem Schlüsselwort „extern“ versehen. Auf diese Weise kann er an mehreren Stellen deklariert werden, darf aber weiterhin nur an einer Stelle definiert werden. Die Deklaration ist eine Information für den Compiler, welchen Typ der Bezeichner hat, die Definition spezifiziert das Objekt selbst, das mit dem Bezeichner verbunden ist.

Um einen Bezeichner nur lokal in einer Datei zu deklarieren, kann man ihn mit dem Schlüsselwort „static“ versehen. Der Compiler kann für den Zugriff auf solche Objekte effizienteren Code generieren, da sie intern gebunden werden

Die Typen von Bezeichnern müssen in allen Deklarationen und in der Definition konsistent sein. Konsequenterweise muß der Quellcode, den der Compiler übersetzen und der Linker binden soll, konsistent sein. In C++ verwendet man zur Deklaration von Typen, Klassen, Prozeduren etc. üblicherweise Headerdateien, die die Schnittstellen-Informationen enthalten und von C++ Quelldateien textuell eingebunden werden. Headerdateien sollten nur Typ-, Funktions-, Daten-, Konstanten- und Namendeklarationen, sowie Include-Direktiven, Makro-Definitionen und Kommentare enthalten, nicht aber Funktions-Definitionen, Datendefinitionen

oder ähnliches. Es soll nur die Schnittstellen-Information in Headerdateien enthalten sein, nicht die Implementierung, die in den C++ Quelldateien vorgenommen werden soll.

Klassen

Klassen sind in C++ Typen mit beschränktem Zugriff auf ihre Elemente und einer Menge von Zugriffsfunktionen. Elemente einer solchen Klasse werden oft mittels spezieller Element-Funktionen, sogenannten Konstruktoren, erzeugt. Weiters wird, bevor ein Objekt gelöscht wird, eine spezielle Funktion, der Destruktor, aufgerufen. Eine Klasse bezeichnet einen Typ und kein Datenobjekt. Jedes Objekt der Klasse besitzt eigene Kopien der Datenelemente der Klasse. Will man einige Datenelemente von allen Objekten der Klasse gemeinsam nutzen, muß man sie mit dem Schlüsselwort „static“ deklarieren, die Objekte bzw. Funktionen müssen allerdings noch irgendwo anders definiert sein.

Jeder Element-Funktion (member function) wird ein Zeiger auf das Objekt, für das die Element-Funktion aufgerufen wurde, als verborgenes Funktionsargument übergeben. Auf dieses implizite Argument kann mit „this“ explizit zugegriffen werden.

Man kann Nicht-Element-Funktionen definieren, die auf private Elemente einer Klasse Zugriff haben, indem man bei der Deklaration das Schlüsselwort „friend“ voranstellt.

Element-Funktionen können in C++ statisch oder dynamisch gebunden werden. Will man dynamische Bindung haben, muß man die Funktion mit dem Schlüsselwort „virtual“ deklarieren.

Rein virtuelle Funktionen (auch abstrakte Funktionen) werden mit „= 0“ initialisiert. Klassen, die abstrakte Element-Funktionen enthalten, sind abstrakte Klassen und können nicht instanziiert werden. Solche abstrakte Element-Funktionen müssen erst in abgeleiteten Klassen redefiniert werden, bevor man Elemente dieser Klasse erzeugen kann.

Der Zugriff auf Elemente einer Klasse ist wie folgt festgelegt: *„Ein Element einer Klasse kann „private“, „protected“ oder „public“ sein: Ist ein Element „private“, so kann sein Name nur in Element-Funktionen und „friends“ der Klasse verwendet werden, in der es deklariert ist. Ist ein Element „protected“, so kann sein Name nur in Element-Funktionen und „friends“ der Klasse verwendet werden, in der es deklariert ist, oder in Element-Funktionen und „friends“ einer von dieser Klasse abgeleiteten Klasse. Ist ein Element „public“, so kann sein Name in beliebigen Funktionen verwendet werden.“* [Str92, p. 226]

Der Zugriff auf Basisklassen wirft weitere Spitzfindigkeiten auf: Ist X eine öffentliche Basis von X1 (class X1: public X {};), kann jede Funktion (implizit) einen Zeiger auf X1 auf einen Zeiger auf ein X konvertieren und auf die „public“-Elemente von X zugreifen. Ist X eine „protected“-Basis von X2 (class X2: protected X {};), können ausschließlich Element-Funktionen und „friends“ von X2 sowie Element-Funktionen und „friends“ von Klassen, die von X2 abgeleitet wurden (implizit) einen Zeiger auf X2 in einen Zeiger auf X konvertieren und auf die „public“- und „protected“-Elemente von X zugreifen. Ist X eine „private“-Basis von X3 (class X3: private X {};), können ausschließlich Element-Funktionen und „friends“ von

X3 (implizit) einen Zeiger auf X3 in einen Zeiger auf X konvertieren, sowie auf die „public“- und „protected“-Elemente von X zugreifen. Die „private“-Elemente einer Basisklasse sind in abgeleiteten Klassen nie zugreifbar.

Mehrfachvererbung

Eine Klasse kann mehr als eine direkte Basisklasse haben.

Arrays

Der Name eines Arrays kann gleichzeitig als Zeiger auf das erste Array-Element verwendet werden. Man kann über Zeiger-Arithmetik auf beliebige Elemente des Arrays zugreifen. Wenn arr wie folgt definiert ist: int *arr, dann ist arr[0] das erste Element des Arrays, auf das arr zeigt, dieselbe Speicherzelle wird durch *arr referenziert. *(arr + 1) ist gleichbedeutend mit arr[1] usw.

C++ führt keine Bereichsüberprüfung durch.

Der i960

Intel hat mit der Serie 80960 eine ganze Familie von ähnlichen Prozessoren für Mikrocontrollerapplikationen herausgebracht. Der in dieser Diplomarbeit eingesetzte Prozessor heißt 80960CA. Eine detaillierte Beschreibung findet sich in [80960].

Details über die Hardware

Der i960 ist ein *superskalarer 32-Bit Prozessor*, der mit bis zu 33MHz getaktet wird. Superskalar bedeutet, daß pro Taktzyklus mehr als ein Befehl abgearbeitet werden kann. Dazu ist es notwendig, daß der Prozessor mehrere Befehle parallel dekodiert und diese Befehle an mehrere Ausführungseinheiten weiterleitet. Der Befehlscache im Prozessor stellt dem Befehlsscheduler des Prozessors zu jedem Taktzyklus die nächsten 4 Befehle zur Verfügung, der diese in einem Takt analysiert und einen, zwei oder drei Befehle ausführen läßt.

Dem Compiler und dem Assembler-Programmierer stehen 32 32-Bit breite, allgemeine Register zur Verfügung, die jeweils 6 Zugriffsports haben. Somit können 6 unabhängige Funktionseinheiten gleichzeitig auf die Register zugreifen.

Um die logische Richtigkeit und Reihenfolge der jetzt parallel ausgeführten Instruktionen beizubehalten, implementiert der Prozessor *„register scoreboarding“*. Interne Flags an den Registern zeigen an, ob diese gerade verwendet werden. Der Prozessor setzt bei Beginn der Befehlsausführung das „scoreboard flag“ des Zielregisters. Wenn folgende Instruktionen keine markierten Register verwenden, kann der Prozessor die Befehle gleichzeitig abarbeiten.

Eine weitere Beschleunigung der Abarbeitung bringt *„register bypassing“* (auch *„result forwarding“* genannt). Dabei wird das Ergebnis eines Befehls, das ein Operand eines zweiten Befehls ist, sofort an die zweite Befehlsausführungseinheit weitergeleitet (ohne die Verzögerung in Kauf zu nehmen, das Ergebnis zuerst in das Zielregister zurückzuschreiben).

Durch statische Sprungvorhersage *(„static branch prediction“)* kann der Programmierer angeben, in welche Richtung ein bedingter Sprung gehen wird. Bei falscher Vorhersage muß die Befehlspipeline invalidiert und die versehentlich begonnenen Befehle müssen mit hoher Wahrscheinlichkeit zurückgesetzt werden.

Auf dem Prozessor selber befinden sich ein 1 KB großer Befehlscache *(two-way set associative instruction cache)* und ein 1 KB großer Datenspeicher.

Je nach Konfiguration steht mehr oder weniger Hauptspeicher zur Verfügung (auf dem verwendeten Entwicklungsboard war es 1 MB RAM).

Prozeduren können entweder über ein *call-and-return* oder über ein *branch-and-link* aufgerufen werden. Bei einem Call werden automatisch alle lokalen Register im Stack-Frame des Aufrufers gesichert. Diese lokalen Register, das sind 16 der insgesamt 32 Register, können innerhalb der aufgerufenen Prozedur dann für lokale Variablen und Zwischenergebnisse in Berechnungen eingesetzt werden. Bei einem Return werden die ursprünglichen Werte der lokalen Register automatisch mit denen aus dem Stack-Frame des Aufrufers wieder hergestellt. Um den Prozeduraufruf effizient zu machen und nicht immer wirklich 16 Register auf dem Stack sichern zu müssen, werden standardmäßig 5 Registersätze im Register-Cache gehalten. Die Größe des Register-Caches ist allerdings frei programmierbar. Bei einem Branch wird kein eigener Stack-Frame für die aufgerufene Prozedur angelegt, die aufgerufene Prozedur benutzt dieselben lokalen Register wie die aufrufende Prozedur (diese Art des Aufrufs verwendet man vor allem für den Aufruf sogenannter „leaf procedures“, also Prozeduren, die keine weiteren Prozeduren aufrufen).

Der *RISC-Befehlssatz* umfaßt die gängigen Operationen wie: load, store, move, arithmetische und boolesche Operationen, Shifts, Operationen auf Bitfeldern, Vergleiche und Sprünge. Auf den Speicher kann allerdings nur (wie bei RISC-Prozessoren üblich) über load und store zugegriffen werden.

Der i960 bietet eine *prioritätengesteuerte Abarbeitung von Interrupts*. Der Interruptkontroller verwaltet 32 Prioritätsstufen bei bis zu 248 verschiedene Interruptquellen, das können externe und interne Hardware, aber auch Software sein. Auf dem eingesetzten Entwicklungsboard gibt es zwei Interruptquellen: der Timerbaustein kann 4 verschiedene Interrupts auslösen und ein Ethernet-Chip kann ebenfalls Interrupts liefern. Der Mechanismus zur Erkennung von Interrupts ist in Hardware implementiert und arbeitet parallel zur Programmausführung, was die Verzögerung bis zur Behandlung minimiert. Weiters kann die aufzurufende Behandlungsroutine („interrupt service routine“ oder kurz ISR) im Cache gehalten werden.

Der Buskontroller ist frei programmierbar. Somit ist es möglich, verschiedenste Speichertypen oder andere Peripherie über den Bus anzusprechen (RAM, ROM, EPROM mit jeweils unterschiedlichen Zugriffszeiten, Wait-states und Busbreiten). Maximal ergibt sich eine Übertragungsrate von 132 Megabytes pro Sekunde (4 Bytes mit 33MHz)

Der DMA-Kontroller hat 4 Kanäle und kann maximal 59 MB pro Sekunde (bei 33MHz) übertragen. Quelle und Ziel der zu übertragenden Daten können entweder im internen RAM, im externen RAM oder im Bereich der Peripherie liegen.

Details über die Software

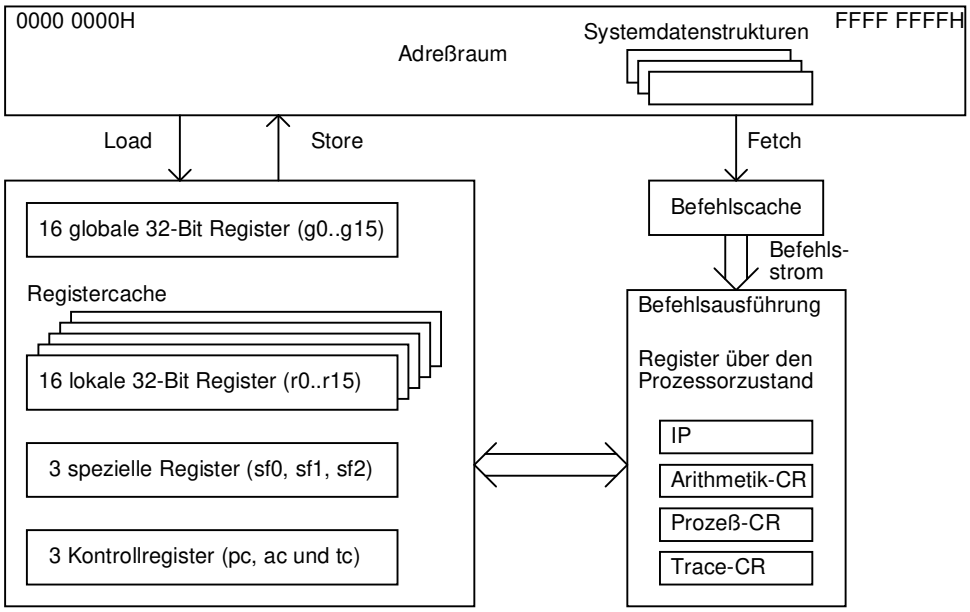

Abbildung 4, Programmierumgebung auf i960

Die globalen Register g0 bis g15 sind 32 Bit breit und universell verwendbar. Die Werte der globalen Register bleiben über Prozeduraufrufe hinweg gleich, weshalb sich die globalen Register für eine effiziente Parameterübergabe eignen. Das Register g15 ist für den aktuellen *frame pointer (fp)* reserviert. Der frame pointer zeigt auf das erste Byte im aktuellen Stackframe.

Die lokalen Register r0 bis 15 sind ebenfalls 32 Bit breit und universell verwendbar. Die lokalen Register werden zur Speicherung von prozedur-lokalen Variablen und temporären Ergebnissen eingesetzt. Bei jedem Prozeduraufruf wird automatisch ein neuer Satz lokaler Register angelegt. Die Register r0, r1 und r2 sind reserviert. r0 enthält den *previous-frame pointer* (pfp, zeigt auf das erste Byte des vorhergehenden Stackframes). r1 enthält den *stack pointer* (sp, zeigt auf das letzte Byte im aktuellen Stackframe). r2 enthält den *return-instruction pointer* (rip, zeigt auf die erste nach dem Prozeduraufruf auszuführende Instruktion).

Das *Arithmetik-Kontrollregister ac* enthält Bedingungsflags (z.B. Ergebnis eines Vergleichs), das *Prozeß-Kontrollregister pc* enthält Informationen über den aktuellen Zustand des Prozessors und das *Trace-Kontrollregister tc* steuert die Möglichkeiten des Tracens beim Debuggen.

Neben den Registern definiert die Prozessorarchitektur einige Datenstrukturen, die für alle Prozessoren der 80960-Familie gleich sind: die Interrupt-Tabelle enthält die Vektoren der Interrupt-Behandlungsroutinen, die Fault-Tabelle enthält die Vektoren der Fault-

Behandlungsroutinen, die Control-Tabelle enthält Werte für die speziellen Register sf0 bis sf31 und der Interrupt-Stack ist der Stack, der während der Abarbeitung von Interrupts verwendet wird.

Implementierung im Überblick

Programmentwicklung für den i960

Entwicklungsumgebung

Das Oberon-System[1] (Version 4.2) läuft als ein Prozeß unter vielen auf einer DECstation 5000/240. Das Oberon-System selbst und das darunterliegende UNIX waren während der gesamten Entwicklung sehr stabil.

Der Standard-Editor unter dem Oberon-System ist voll objektorientiert implementiert. Er behandelt Zeichen sowie beliebige Textelemente auf die gleiche Weise. Beliebige Erweiterungen solcher Textelemente fließen wie normale Zeichen im Text mit (Graphiken, Popup-Menüs, Uhren, Tabellen, Links zu anderen Stellen etc.). Besonders nützlich sind hier die Faltelemente, die es erlauben, größere Textpassagen zusammenschrumpfen zu lassen. Durch einfaches Klicken auf diese Elemente öffnen sie sich und zeigen den zusammengefalteten Text. Wenn man Programme entsprechend mit Faltelementen strukturiert, erhält man leicht eine gute Übersicht und findet schnell an die richtige Stelle.

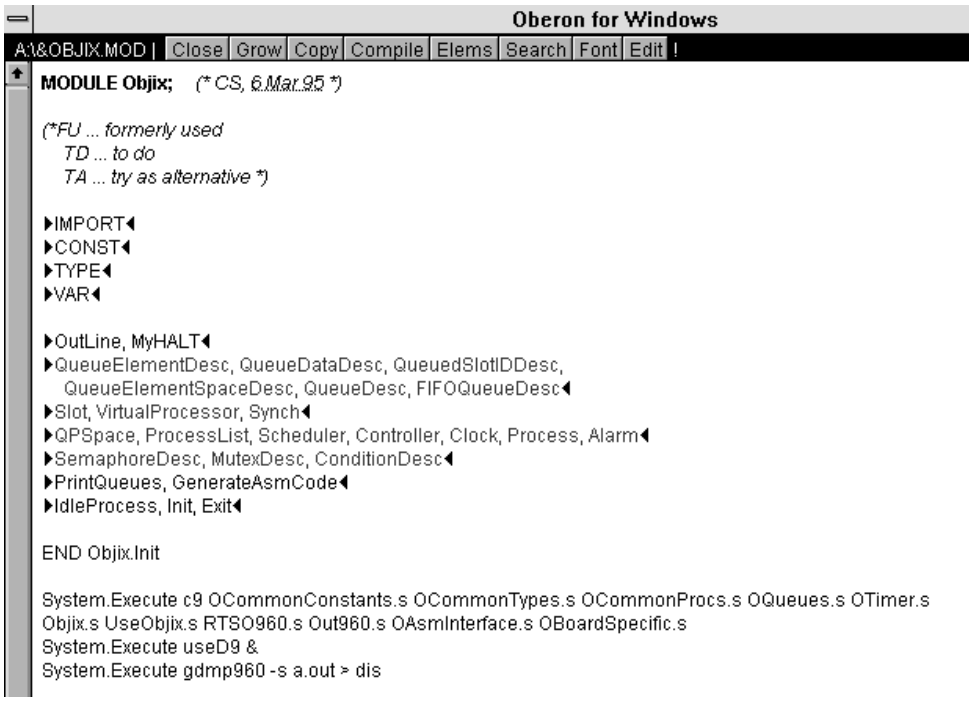

Abbildung 5, Hauptmodul mit zusammengeklappten Faltelementen

[1] Das Oberon-System ist für verschiedenste Rechnersysteme über das Internet kostenlos zu beziehen, z.B. via ftp von neptune.ethz.ch (user: anonymous, password: own email address) oder oberon.ssw.uni-linz.ac.at (user: ftp, password: ftp).

Abbildung 6, Hauptmodul mit geöffneten Faltelementen

Das Oberon-System bietet Portabilität auf Source-Code-Ebene. Besonders in der Anfangsphase meiner Diplomarbeit war es hilfreich, daß die erstellten Texte unter jedem Oberon-System dieselbe Formatierung (Schriftart, Schriftgröße, Farbe, Einrückung etc.) beibehalten. So war es mir möglich, die Portierung von C++ auf Oberon-2 größtenteils zu Hause auf meinem i386, auf einem PowerMac oder auf der UNIX-Workstation zu machen.

Der in der Diplomarbeit eingesetzte Oberon-2 Compiler wurde von Richard Hable (auch im Rahmen einer Diplomarbeit, [HAB95]) implementiert. Dieser Compiler ist eigentlich ein Crosscompiler: er generiert Code, der nicht auf der UNIX-Maschine selbst ausführbar ist, sondern Assembler-Code für den i960. Dieser Assembler-Code wird mit dem Standard-Assembler in Objektcode übersetzt und mit anderen Modulen zu einem ausführbaren Programm gelinkt. Das ausführbare Programm muß anschließend von der UNIX-Maschine auf das Entwicklungsboard übertragen werden. Diese Übertragung kann entweder über die serielle Schnittstelle (mit 9600 Baud) oder über eine Ethernet-Verbindung erfolgen. Unauffindbare Instabilitäten dieser Verbindungen machen allerdings das Auffinden von Fehlern teilweise sehr schwer.

Die Intel-Entwicklungstools umfassen neben einem C-Compiler einen Assembler, Linker, Debugger, sowie einen Disassembler. Alle Tools haben ein rein textuelles Interface (UNIX-like).

Von der Source zum Executable

Der Oberon-2-Compiler erzeugt aus den Oberon-Modulen Assembler-Source-Files für den i960 sowie Symbol-Files. Die Symbol-Files verwendet der Oberon-2-Compiler wieder, um

Abhängigkeiten bei IMPORTs zu überprüfen. Der folgende Befehl übersetzt alle Module mit dem Crosscompiler:

Folds.Compile OAsmInterface.Mod /s OBoardSpecific.Mod /s OCommonConstants.Mod /s OCommonTypes.Mod /s OTimer.Mod /s Objix.Mod /s UseObjix.Mod /s ~

Wie bisher (= mit den Intel-Entwicklungstools) können C- und Header-Dateien mit dem Standard-C-Compiler zu Objektcode-Dateien compiliert werden.

Über eine kleine Erweiterung des Oberon-2-Compilers[2] ist es möglich, ausgehend von einem Modul nur ein Symbol-File zu erzeugen. Das kann man verwenden, um die Schnittstelle einer C-Bibliothek in einem Oberon-Modul nachzubilden. Das generierte Symbol-File kann man zum Zugriff auf die C-Bibliothek verwenden. Die Datei OAsmInterface.Mod ist ein ähnliches Beispiel: Es wird nur ein Symbol-File erzeugt; die darin deklarierten Prozeduren werden in der handgeschriebenen Datei OAsmInterface.s definiert.

Schließlich werden die benötigten Assembler-Source-Files mit dem Standard-Assembler in Objekt-Files übersetzt, und mit dem Standard-Linker werden die Objekt-Files zu einem ausführbaren Programm gelinkt. Ein zusätzliches Linker-Definition-File (im unten angegebenen Beispiel ist das „/mnt/users/ste/dipl/mevca“) beschreibt, welche Arten von Speicher auf dem Entwicklungsboard zur Verfügung stehen und welche Segmente (Text, Daten, Code) in welche Speicherbereiche auf dem Entwicklungsboard geladen werden sollen. Der folgende Befehl assembliert und linkt die Module:

gcc960 -Fcoff -g -ACA -T/mnt/users/ste/dipl/mevca OAsmInterface.s OCommonConstants.s OCommonTypes.s OTimer.s Objix.s UseObjix.s RTSO960.s Out960.s OBoardSpecific.s Fout960.s

Abschließend kann das gelinkte, ausführbare Programm (a.out) mit dem Debugger auf das Entwicklungsboard übertragen werden. Hierzu startet man den gdb960 und gibt danach die folgenden Kommandos ein:

target mon960 tty 00
exec a.out
file a.out
load a.out

[2] Zusätzlich zur Exportmarkierung mit “*” und “-” (read-only) gibt es die Markierung mit einem Plus “+” (nur erlaubt, wenn das Modul SYSTEM importiert wird, da es sich um eine nicht-portable Compilererweiterung handelt). Normalen Symbolen wird der Modulname vorangestellt, um sie global eindeutig zu machen. Variablen und Prozeduren, die mit einem “+” markiert sind, erhalten C-kompatible Namen, die man von C-Modulen bzw. von Assembler-Modulen aus verwenden kann. Zusätzlich kann man Module mit einem “+” markieren (z.B. MODULE Test+; ...). Der Crosscompiler erzeugt dann für dieses Modul zwar eine Symboldatei, aber keine Assembler-Zieldatei.

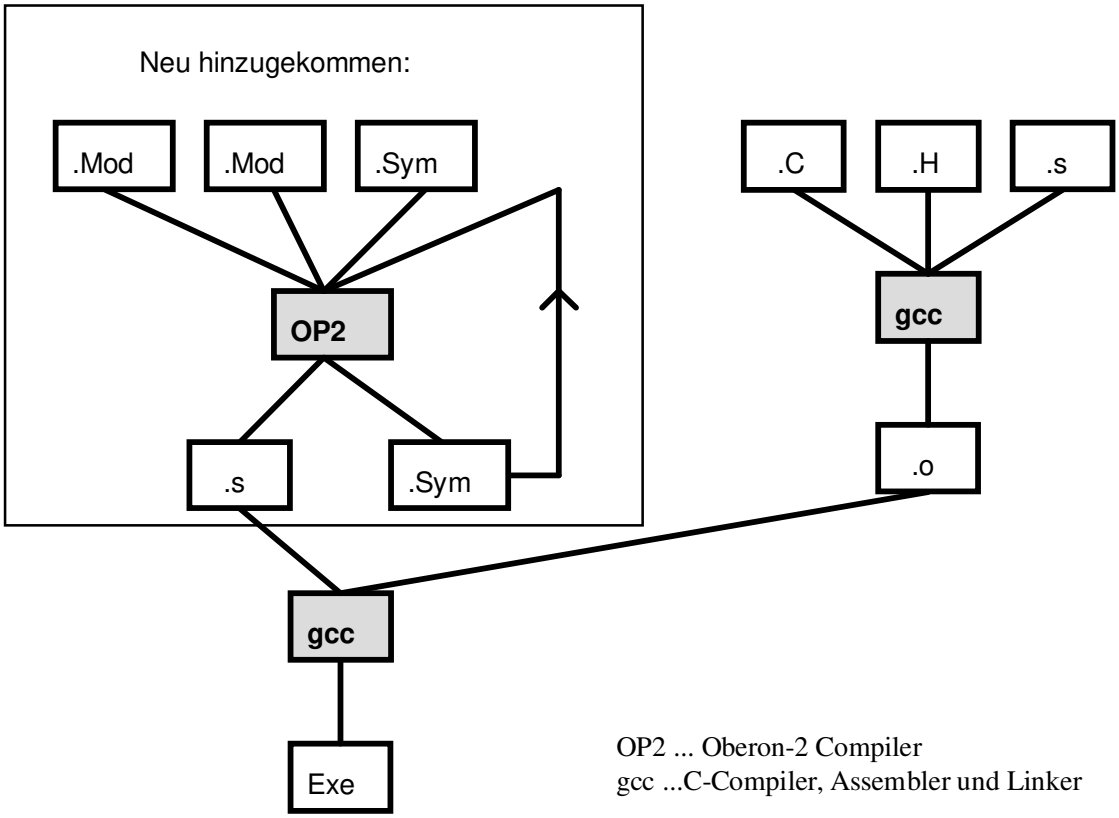

Abbildung 7, Weg von der Source zum Executable

Objix

Als Grundlage für den Mikrokernel habe ich den von Richard Fanta an der Prager Universität entwickelten Kernel "Objix" [FAN94] verwendet. Dieser Kernel ist in Borland C++ 2.0 für den 8086-Prozessor und segmentierte Speicherarchitektur geschrieben.

Analyse der bestehenden C++ Implementation

Die ursprüngliche C++ Implementation wurde mit Borland C++ 2.0 implementiert.

Insgesamt umfaßten die Quelltexte (10 Header-, 4 .cpp- und 4.asm-Files) etwas mehr als 8000 Zeilen, knapp 9000 Wörter und knapp 110000 Zeichen.

Wenn man sich das Verhältnis der Wörter zu der Anzahl der Zeilen ausrechnet kommt man auf 1.08 Wörter pro Zeile. Dazu muß man allerdings bemerken, daß die geschwungenen Klammern von C++ (die ja meist einem BEGIN bzw. END von Oberon entsprechen) nicht als Wörter gezählt wurden.

In der C++ Implementation wurde zu häufig Vererbung eingesetzt, einfach um Funktionalität zu erben. Diese IS-A-Beziehungen haben dann oft eine seltsame Semantik. Auch erfolgte die Codierung der Prozeduren und Funktionen gemischt in den Header- und in den CPP-Files. Es

gab keine saubere Trennung in Definition-Files (die nur die Schnittstelle beschreiben) und Implementation-Files.

Ursprüngliche Klassenhierarchie

Ich möchte hier nicht näher auf die ursprüngliche Klassenhierarchie eingehen. Sie ist in [FAN94] ausführlich beschrieben. Es sei nur erwähnt, daß die Klassen Synch und SemiSynch für die Implementierung der Synchronisation bei getrennten Adreßräumen notwendig waren.

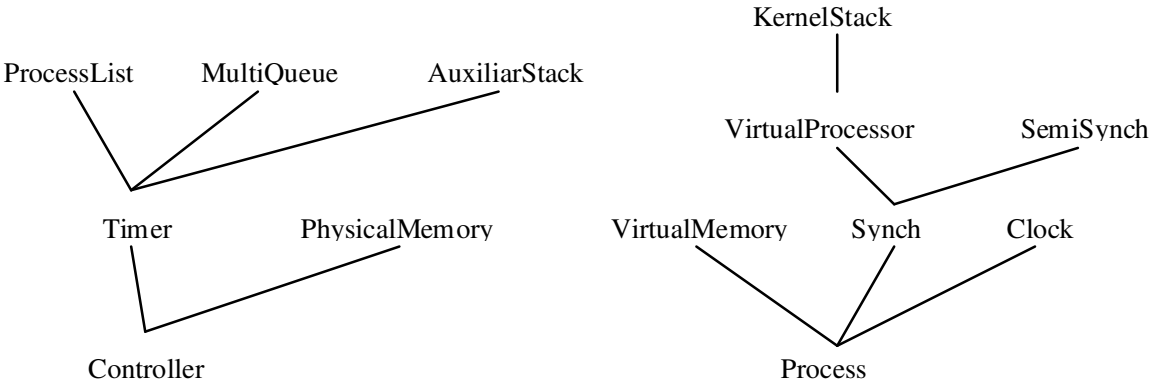

Abbildung 8, ursprüngliche Klassenhierarchie

Vereinfachungen

Ein gemeinsamer Adreßraum

Alle Prozesse laufen in einem gemeinsamen Adreßraum. In typischen Microkontroller-Applikationen ist es so, daß mehrere Prozesse für die Steuerung gewisser Abläufe zuständig sind. Diese Prozesse haben einen Gesamtzustand, der sich aus den Zuständen der einzelnen Prozesse und gewisser globaler Daten ergibt. Die Typsicherheit von Oberon-2 und objektorientiertes Programmieren machen es möglich, einen gemeinsamen Adreßraum zu haben und trotzdem unerlaubten Zugriff auf privaten Speicher unmöglich zu machen. In Oberon-2 zeigen Zeiger entweder auf gültige Objekte oder haben den Wert NIL. Eine Initialisierung eines Zeigers mit einer gewissen Speicheradresse ist ohne die Verwendung des Low-level-Moduls SYSTEM nicht möglich. Da der verwendete Oberon-2-Compiler kein Laufzeitsystem mit Garbage-Kollektor generiert, muß sich der Benutzer um die Freigabe des Speichers selbst kümmern, und es können „dangling pointers“ (Zeiger, die auf nicht mehr gültige Speicherbereiche zeigen) entstehen.

Methoden-Aufruf statt Message-Passing

In der ursprünglichen Implementierung von Objix wurden Systemfunktionen über Softwareinterrupts aufgerufen, die zur Implementierung von Message-Passing verwendet wurden. In typischen Microkontroller-Applikationen werden allerdings die Anwendermodule mit den Systemmodulen gelinkt, und es liegt daher ein direkter Methoden-Aufruf (auch aus Effizienzgründen) nahe.

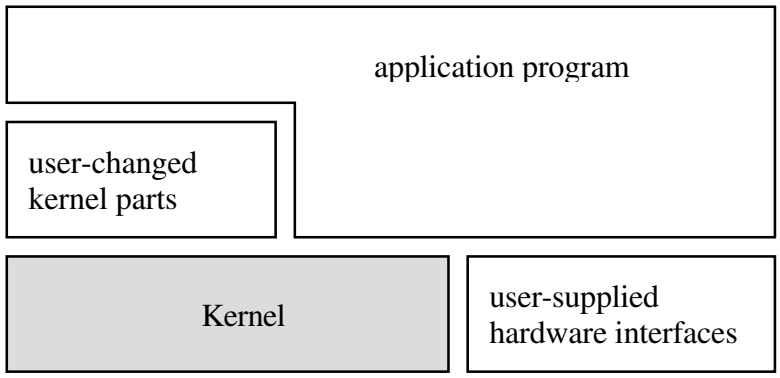

Abbildung 9, Schema einer Mikrokontroller-Applikation

Kein Loader

Da Microkontroller-Anwendungen, sobald sie gelinkt sind, auf das Entwicklungsboard geladen oder in ein EPROM gebrannt werden, ist kein Lader notwendig, der zur Laufzeit bestimmte Module nachladen würde. Das Programm wird in genau der Weise, wie es später laufen soll, vom Linker erzeugt. In einem zusätzlichen Linker-Definition-File gibt man dem Linker weitere Daten bekannt (Größe und Anfangsadresse von RAM, ROM, EPROM, Zugriffszeiten, Wait-States etc.).

Kein Filesystem, Aufruf beliebiger C-Bibliotheksfunktionen

Prinzipiell ist der Aufruf beliebiger C-Bibliotheksfunktionen möglich. Somit kann man zum Beispiel auf die Standard-Ein- und -Ausgabe von C zurückgreifen. Ein Problem dabei ist allerdings, daß diese C-Bibliotheken im allgemeinen nicht reentrant sind, mit meinem Mikrokernel aber durchaus mehrere Prozesse zur gleichen Zeit ein und dieselbe C-Funktion aufrufen und ausführen können. Als Abhilfe dafür kann man eine Zwischenschicht zwischen den Oberon-Prozessen und den C-Bibliotheken schaffen, in der man (z.B. über Mutual-Exclusion-Variablen) nur jeweils einen Prozeß die Bibliothek benutzen läßt, alle weiteren werden blockiert. Diese Alternative ist aber für harte Echtzeit-Anforderungen ungeeignet. Weiters ergibt sich das Problem, daß man dadurch die Deadlock-Gefahr erhöht. Zur Zeit wird deshalb der Zugriff auf C-Bibliotheken nicht serialisiert.

Portierung auf Oberon-2

Umsetzung von Mehrfachvererbung auf Einfachvererbung

In [RUM93, p. 84f] kann man folgende Möglichkeiten finden, Mehrfachvererbung (wenn sie zum Beispiel in der Zielsprache nicht unterstützt wird) von vornherein schon zu vermeiden:

„Wenn eine Unterklasse mehrere gleich wichtige Oberklassen hat, empfiehlt es sich, Delegation zu verwenden und die Symmetrie des Modells zu erhalten.

Wenn eine Oberklasse klar gegenüber anderen dominiert, empfiehlt es sich, Mehrfachvererbung durch Einfachvererbung und Delegation zu implementieren.

Wenn die Zahl der Kombinationen klein ist, sollten Sie eine verschachtelte Generalisierung in Betracht ziehen. Wenn die Zahl der Kombinationen groß ist, sollten Sie sie vermeiden.

Wenn eine Oberklasse deutlich mehr Merkmale besitzt als die anderen Oberklassen oder wenn eine Oberklasse eindeutig der Leistungsengpaß ist, sollten Sie die Vererbung über diesen Pfad hinweg erhalten.

Wenn Sie sich für verschachtelte Generalisierung entscheiden, befassen Sie sich zuerst mit dem wichtigsten Kriterium, dann mit dem zweitwichtigsten, usw.

Versuchen Sie, verschachtelte Generalisierung zu vermeiden, wenn große Codemengen dupliziert werden müssen.

Denken Sie an die Wichtigkeit, strikte Identität zu erhalten. Dies ist nur mit der verschachtelten Generalisierung möglich.“

In [MOE93, p. 139] liest man folgende mahnende Aussage:

„Inheritance establishes an is-a-relationship between a subclass and its superclass. B should only be derived from A if it is an extension or refinement of A. Instead, inheritance is often abused to represent a has-a relationship.“

Ein Beispiel für eine unechte IS-A-Beziehung in der ursprünglichen Klassenhierarchie ist folgende: Process ist abgeleitet von VirtualMemory, um Zugriff auf die Funktionen der Klasse VirtualMemory zu erhalten, nicht aber, weil er ein virtueller Speicher ist.

Die unechten IS-A-Beziehungen der Mehrfachvererbung ließen sich in allen Fällen problemlos auf Einfachvererbung und HAS-A-Beziehungen (Delegation) umsetzen. Im Standardfall wurde aus einer solchen IS-A-Beziehung ein zusätzlicher Pointer. Der Scheduler hat zum Beispiel anstatt der geerbten ProcessList ein Feld, das auf ein Objekt der Klasse ProcessList zeigt. Es ist kaum ein Laufzeitoverhead zu erwarten, wenn man vor einem Funktionsaufruf diesen Pointer dereferenzieren muß. Im Gegensatz dazu ist bei C++ doch ständig ein Laufzeitoverhead gegeben, der notwendig ist, um Mehrfachvererbung prinzipiell zu ermöglichen.

Laufzeittypinformation

Laufzeittypinformationen stehen in Oberon-2 standardmäßig zur Verfügung, Typ-Tests und sichere Typ-Guards sind Teil des Sprachumfangs. In C++ kann man Ähnliches nachimplementieren ist dabei aber wesentlich ineffizienter (Erst in einem aktuelleren ANSI-Standard wird es RTTI, run-time type information, geben. Dieser Standard wird aber erst von den wenigsten Compilern unterstützt.).

Konstruktoren/Destruktoren

Der eingesetzte Oberon-2-Compiler erzeugt Standalone-Programme ohne ein Laufzeitsystem mit Garbage-Kollektor. Deshalb ist die Nachbildung der Konstruktoren und Destruktoren notwendig. Eine pragmatische Vorgehensweise dafür ist die Trennung des Anlegen eines

Objekts in die zwei Phasen Anfordern des Speicherplatzes und Initialisieren des Speicherplatzes. Analog dazu erfolgt bei der Freigabe eines Objektes zuerst eine Deinitialisierung oder Finalisierung.

Sichtbarkeits- und Zugreifbarkeitsstrukturen
Die Nachbildung der Sichtbarkeits- und Zugreifbarkeitsstrukturen stellte ich mir anfangs sehr schwierig vor: In C++ kann eine Klasse private, protected und public Felder haben, man kann Freundschaftsbeziehungen zwischen Klassen definieren und zusätzlich Klassen voneinander private, protected, public und/oder virtual ableiten. In Oberon-2 hingegen gibt es nur die Export-Markierung durch einen Stern beziehungsweise durch ein Minus.

Am Anfang der Portierung hatte ich viele kleine Module (entsprechend den Klassen und ihrer Affinität zueinander), mit der Zeit wurden es weniger, bis ich schließlich fast alle Kernel-internen Klassen in einem Modul zusammengefaßt hatte.

Bei der Portierung versuchte ich außerdem zuerst, genau die vorgegebenen Beziehungen nachzubilden, im Laufe der Zeit bin ich aber, mit besserem Verständnis für die Problematik, problemlos mit den einfachen Exportmechanismen von Oberon-2 ausgekommen.

Weglassen nicht benötigter Klassen
Die Klassen, die für die Implementierung der segmentierten Speicherverwaltung notwendig waren und jene für die Implementierung des Message-Passing, konnte ich in meiner Implementierung weglassen.

Aktueller Stand

Der aktuelle Stand umfaßt 7 Dateien (eine Hauptdatei), rund 2500 Zeilen, 9000 Wörter und 60000 Zeichen. Die meisten der Dateien sind sehr kurz (weniger als eine Seite). Das Hauptmodul heißt Objix und muß von dem Anwendungsprogramm aus importiert werden. Diese Kürze unterstreicht die Einfachheit der Implementierung und den erfolgreichen Einsatz objektorientierter Techniken, sowie der Wiederverwendung gemeinsamen Codes.

Es scheint trivial - ich möchte es aber dennoch erwähnen: Durch das konsequente Einfügen von Kommentaren am Beginn von Prozeduren, als Assertionen und an sonstigen Stellen, wo es notwendig ist, wird der Code wesentlich lesbarer. Auch eine einheitliche Formatierung hilft maßgeblich.

Bestehende Klassen(beziehungen)

Die in den Objektdiagrammen verwendete Notation hält sich an die in [RUM93] vorgeschlagene. Das OMT (Object Modeling Technique) - Symbol für eine Objektklasse ist ein Rechteck mit dem Klassennamen in fetter Schrift. Assoziationen werden durch eine Linie zwischen Objektklassen, Aggregationen werden ebenfalls durch Linien mit einer kleinen Raute auf der Seite der aggregierenden Klasse dargestellt. Bestimmte Symbole an den Enden der Assoziationslinien geben die Multiplizität der Assoziation an: ein fetter schwarzer Punkt steht

für „null oder mehr“, ein transparenter Punkt steht für „null oder eins“. Die Notation für die Vererbung ist ein Dreieck, das eine Oberklasse mit ihren Unterklassen verbindet. Die Oberklasse wird durch eine Linie mit der Spitze des Dreiecks verbunden.

Beschreibung des Objektdiagramms

Im System existiert ein Scheduler, der beliebig viele Prozesse verwaltet. Jeder Prozeß hat einen virtuellen Prozessor, der den Zustand des Prozessors speichert, wenn dieser Prozeß unterbrochen ist. Alarme sind Unterklassen von Prozessen. Jeder Prozeß kann keinen oder mehrere Alarme haben. Mutex ist eine Unterklasse von Semaphore.

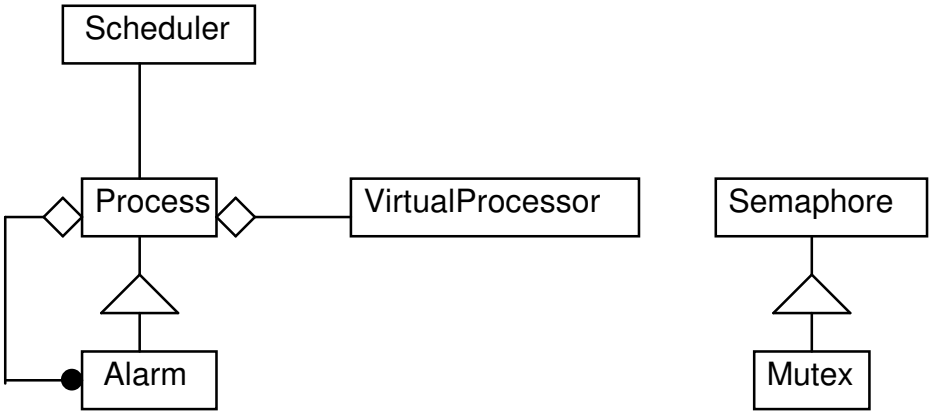

Abbildung 10, Objektmodell des Kernels

Scheduling und Prozeßwechsel

Grundlegendes über Scheduling

Der Scheduler verwaltet die im System existierenden Prozesse. In einem Einprozessorsystem kann immer nur ein Prozeß dem Prozessor zugeordnet sein, d.h. vom Prozessor abgearbeitet werden. Um trotzdem zu erreichen, daß mehrere Prozesse quasiparallel abgearbeitet werden, muß man zwischen den Prozessen umschalten.

In Timesharing-Systemen teilen sich die Prozesse die Prozessorzeit in einzelne Zeitscheiben auf. Jeder Prozeß erhält eine Zeitscheibe bestimmter Größe (“time slice”). Nach Ablauf seiner Zeitscheibe gibt der Prozeß die Kontolle an den nächsten Prozeß ab.

Zusätzlich ist es von Vorteil, den Prozessen verschiedene Prioritäten zuordnen zu können. Damit kann man erreichen, daß wichtige Prozesse öfter bearbeitet werden, unwichtigere seltener.

Der Scheduler verwaltet nicht nur die Prozesse, sondern macht auch die automatischen Prozeßwechsel: er teilt immer dem zur Zeit wichtigsten Prozeß den Prozessor zu. Wegen der Steuerung über die Prioritäten der Prozesse kann man den Scheduler “unterbrechend” (engl. preemptive) nennen, er unterbricht einen Prozeß, wenn ein Prozeß höherer Priorität bereit wird.

Weil Prozesse nicht immer abgearbeitet werden, sondern unter anderem auch unterbrochen sein können, ist eine Unterteilung in verschiedene Zustände sinnvoll. Im hier implementierten Scheduler können Prozesse in einem von 6 Zuständen sein (die Wahl dieser 6 Zustände lehnt sich an die Implementierung existierender Echtzeitbetriebssysteme an):

“Running” ist immer genau ein Prozeß. Dieser Prozeß wird gerade ausgeführt.

“Ready” ist ein Prozeß, wenn er bereit ist, ausgeführt zu werden. Ein Prozeß, der „ready“ ist, kann beim nächsten automatischen Prozeßwechsel „running“ werden.

“Asleep” ist ein Prozeß, wenn er schläft. Schlafende Prozesse warten auf das Ablaufen einer gewissen Zeitspanne (im allgemeinen auf das Eintreten einer Bedingung) und wachen dann von selber wieder auf.

“Suspended” ist ein Prozeß, wenn er unterbrochen wurde und auf die Wiederaufnahme wartet. Ein unterbrochener Prozeß kann nicht mehr von selbst „ready“ werden.

“Asleep-suspended” ist ein Prozeß, der unterbrochen wurde, als er schon „asleep“ war. Ein solcher Prozeß muß sowohl aufwachen als auch wiederaufgenommen werden.

“Zombie” ist ein Prozeß, nachdem er gestorben ist und bevor der von ihm belegte Speicher wiederverwendet werden kann.

Die Zustände „asleep“, „suspended“ und „asleep-suspended“ könnte man zusammenfassen, wenn man ein Sleep(n) als ein Suspend implementiert und dem Prozeß anstelle des Wakeups nach n Zeiteinheiten ein Resume schickt.

Für jede Prioritätsstufe und pro Prozeßzustand gibt es eine FIFOQueue für die Prozesse mit dieser Priorität, die sich in diesem Zustand befinden. Der Prozeß mit der höchsten Priorität wird zum laufenden Prozeß und kann den Prozessor prinzipiell unendlich lange in Anspruch nehmen. Wenn man will, daß mehrere Prozesse gleicher Priorität abwechselnd den Prozessor bekommen, muß man für jeden Prozeß eine Timeslice vergeben. Nach Ablauf dieser Timeslice wird ein Rescheduling-Vorgang initiiert. Dabei wird der aktuell laufende Prozeß aus der zu seiner Priorität gehörenden FIFOQueue entnommen und neu eingefügt (dadurch wird er der letzte Prozeß in dieser Queue). Auf diese einfache Weise kann man Timesharing innerhalb der Prozesse gleicher Priorität erreichen, wobei die Zeitscheibengröße pro Prozeß festgelegt werden kann. Man erspart sich somit die Einführung eines "real time fence". Als „real time fence“ bezeichnet man die Grenze, ab welcher Priorität die Prozesse gleicher Priorität nach dem Round-Robin-Verfahren verwaltet werden. Prozesse mit höherer Priorität können den Prozessor beliebig lange blockieren, Prozesse niedrigerer Priorität teilen sich die Prozessorzeit.

Änderung des Prozeßzustandes

Zur folgenden Beschreibung der Zustandsüberführungen sollte man nebenbei schon das Diagramm auf der nächsten Seite betrachten.

Prozesse sind nach ihrer Erstellung entweder im Zustand „ready“ oder im Zustand „suspended“.

Ein „ready“-Prozeß kann bei jedem automatischen Scheduling-Vorgang vom Scheduler über den Prozeduraufruf DispatchNext „running“ werden. Preempt führt den zur Zeit laufenden Prozeß in den Zustand „ready“ über, DispatchNext macht den Prozeß mit der höchsten Priorität „running“. Ein Rescheduling-Vorgang besteht aus den beiden Aufrufen Preempt und DispatchNext. Automatisches Rescheduling wird in folgenden Fällen durchgeführt: bei einem Sleep(0), in StartScheduling, in SetPriority (wenn sich die Priorität geändert hat und ein anderer Prozeß jetzt der mit der höchsten Priorität sein könnte) und in der periodisch aufgerufenen Schedulingprozedur (wenn die Zeitscheibe des aktuellen Prozesses abgelaufen ist oder ein Prozeß höherer Priorität aufgewacht ist).

Ein Prozeß kann sich (wenn er „running“ ist) selbst unterbrechen und ist dann „suspended“. In diesem Fall wird automatisch der „ready“-Prozeß mit höchster Priorität „running“. Ein Prozeß kann andere Prozesse unterbrechen. Diese Prozesse werden dann entweder „suspended“ oder „asleep-suspended“. Um die Abarbeitung eines Prozesses wiederaufzunehmen, muß Resume aufgerufen werden. Die Aufrufe von Suspend und Resume müssen immer paarweise auftreten, man kann Prozesse allerdings mehrmals unterbrechen. Für jedes Suspend muß es allerdings ein Resume geben.

Ein Prozeß kann sich (wenn er „running“ ist) schlafen legen und ist dann „asleep“. Der Scheduler weckt schlafende Prozesse nach Ablauf des Nickerchens automatisch wieder auf (über Wakeup).

Ein Prozeß kann sich entweder selbst umbringen, er kann von anderen Prozessen umgebracht oder vom Scheduler beseitigt werden.

In der nächsten Abbildung sind die Zustandsüberführungen noch einmal graphisch dargestellt. Zu beachten sind dabei auch die Pfeilarten: eine durchgezogene Linie steht für eine Zustandsüberführung, die vom Scheduler initiiert ist, eine gepunktete Linie für eine vom Prozeß selbst initiierte, eine strichlierte Linie steht für eine Zustandsüberführung, die ein Prozeß auf einen anderen anwendet und eine strich-punktierte Linie schließlich steht für eine Zustandsüberführung, die ein Prozeß auf sich selbst oder auf einen anderen Prozeß anwendet.

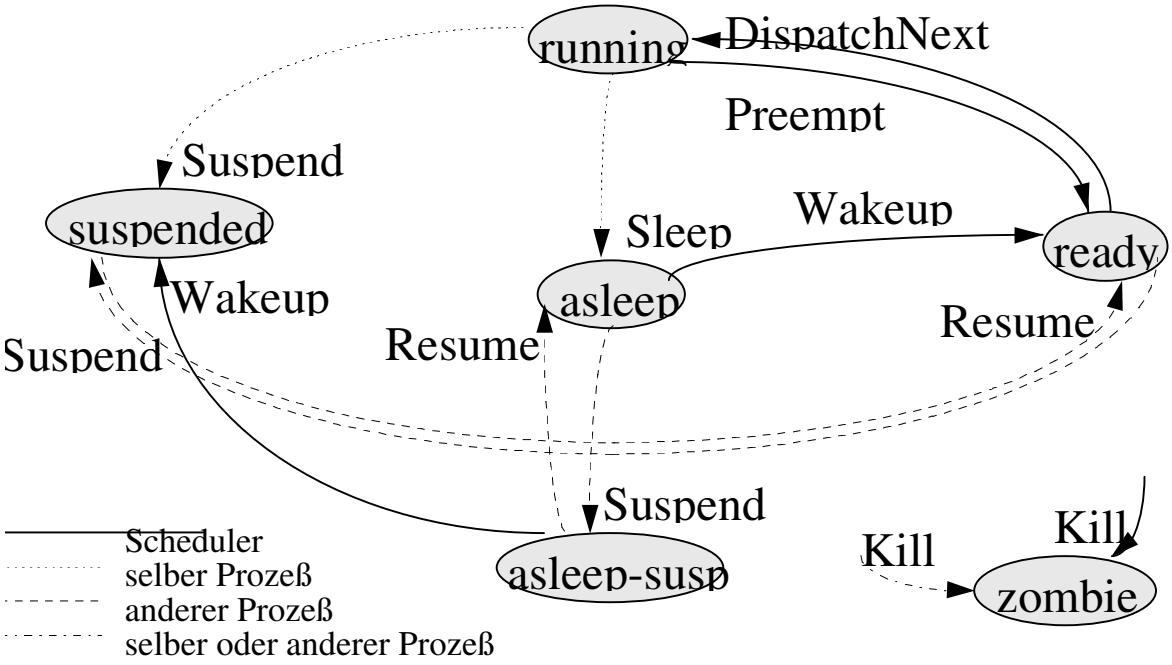

Abbildung 11, Prozeßzustandsüberführungsdiagramm

Wie kann nun ein Anwenderprogramm die Möglichkeiten des Scheduling ausnutzen?

Sleep(ticks)

Durch ein Sleep(ticks) kann sich ein Prozeß für eine bestimmte Anzahl von Timer-Ticks schlafen legen. Der Prozeß wird dazu aus der ReadyQueue entnommen und in die AsleepQueue eingefügt. Bei einem Aufruf von Sleep(0) gibt der Prozeß den Prozessor frei und veranlaßt einen Rescheduling-Vorgang. Ist der Prozeß der mit der höchsten Priorität und einzige dieser Priorität, so läuft der Prozeß auch nach dem Sleep(0) weiter.

Suspend

Mit einem Suspend kann ein Prozeß sich selbst oder andere Prozesse unterbrechen. Der Prozeß wird dazu aus der ReadyQueue entnommen und in die SuspendedQueue eingefügt (bzw. aus der AsleepQueue entnommen und in die AsleepSuspendedQueue eingefügt). Der Prozeß hat keine Chance, von selbst wieder laufend zu werden. Er muß auf ein Resume warten, das ein anderer Prozeß auf ihn anwendet.

Resume

Mit einem Resume kann ein Prozeß die Abarbeitung anderer Prozesse wiederaufnehmen. Der Prozeß wird dazu aus der SuspendedQueue entnommen und in die ReadyQueue eingefügt (bzw. aus der AsleepSuspendedQueue entnommen und in die AsleepQueue eingefügt).

SetPriority(newPrio)

Mit SetPriority(newPrio) kann ein Prozeß seine Priorität ändern. Wenn die neue Priorität niedriger als die aktuelle ist, kann es sein, daß der Prozeß nach dem Ändern der Priorität nicht mehr der mit der höchsten Priorität ist und automatisches Rescheduling notwendig wird.

Kill

Mit Kill kann ein Prozeß sich selbst oder andere Prozesse zerstören. Dazu wird der Prozeß aus der entsprechenden Queue entfernt, und alle seine Resourcen (z.B. Alarme etc.) werden freigegeben.

StopScheduling und StartScheduling

Mit StopScheduling und StartScheduling kann ein Prozeß das Scheduling zeitweise (nur für kurze Zeit empfohlen) unterbrechen bzw. wieder freigeben.

Prozeßwechsel

Der Prozeßwechsel kann prinzipiell auf zwei Arten passieren: freiwillig (aus der Sicht des Prozesses) durch Aufruf von Kernelfunktionen oder unfreiwillig durch das automatische Scheduling. Wenn der Prozeß quasi „freiwillig" Kernelfunktionen aufruft, muß er in Kauf nehmen, daß ein anderer Prozeß den Prozessor zugeteilt bekommt, und er selbst unterbrochen wird.

Freiwilliger Prozeßwechsel

Ein Aufruf von *Sleep(n)* führt zu einem Prozeßwechsel, wenn es weitere Prozesse derselben Priorität gibt. Ein Aufruf von *self.Suspend* führt immer zu einem Prozeßwechsel. Ein Aufruf von *SetPriority* führt zu einem Prozeßwechsel, wenn durch das Herabsetzen der eigenen Priorität ein anderer Prozeß der mit der höchsten Priorität wird. Ein Aufruf von *StartScheduling* führt zu einem Prozeßwechsel, wenn ein Prozeß mit höherer Priorität „ready" ist. Ein *Semaphore-Wait* führt zu einem Prozeßwechsel, wenn die Semaphore belegt ist.

Unfreiwilliger Prozeßwechsel (automatisches Scheduling)

Ein Prozeß kann, ohne daß er es verhindern kann, unterbrochen werden und den Prozessor abgeben müssen: Wenn nämlich seine Zeitscheibe abgelaufen ist oder ein Prozeß mit höherer Priorität aufgewacht ist, initiiert der Scheduler einen Reschedule-Vorgang.

Prozeßwechsel auf dem i960 im Detail

Das Studium dieses Kapitels ist für den Leser sicher anstrengend. Man muß, um die Details verstehen zu können, mit der Prozessorarchitektur und der Problematik des Prozeßwechsels vertraut sein. Ich werde teilweise Wissen aus dem i960-Handbuch [80960] voraussetzen, möchte den Prozeßwechsel aber trotzdem im Detail besprechen, da dieser Teil in der mir zur Verfügung stehenden Literatur über den i960 nicht gut beschrieben ist. Es dürfte also für Folgearbeiten auf diesem Gebiet sicher von Vorteil sein, wenn man auf gesichertem Wissen aufbauen kann.

In meinem Mikrokernel gibt es intern drei Arten von Prozeßwechsel:

1. das Anspringen des ersten Prozesses im System (der IdleProcess)[3],
2. den Prozeßwechsel infolge des automatischen Schedulings von der Timerinterrupt-Service-Routine aus und
3. den freiwilligen Prozeßwechsel von einem Prozeß zu einem anderen.

Ein Prozeßwechsel kann auf dem i960 nur so durchgeführt werden, daß man bei einem Rücksprung aus einer Prozedur nicht mehr dorthin zurückspringt, woher man gekommen ist, sondern zu einem neuen Prozeß.

Jeder Prozeß hat als Datenfelder die Register r0..r15, g0..g15, ac, pc, tc und pfp.

Eine Tatsache beim Prozeßwechsel ist nicht augenfällig und wird deshalb explizit erwähnt: Wie aus der Tabelle 1 ersichtlich ist, sind nach einem Call die Werte des Registers r0 (auf dem Stack-Frame des Aufrufers gesichert) und pfp nicht gleich. Das auf dem Stack gesicherte Register r0 zeigt auf den alten Stack-Frame, während pfp schon auf den neuen zeigt. Man muß sehr genau zwischen den wirklichen Registern, den auf dem Stack gesicherten Registern und den Datenfeldern der Prozesse unterscheiden.

Die Bezeichnungen pfp, sp, rip und fp sind synonym zu den Registern r0, r1, r2 und g15. Die Register r0, r1 und r2 stehen immer am Anfang des Stack-Frames.

Prozeßwechsel durch StartCurrentProcess

Die Prozedur StartCurrentProcess ist in Assembler kodiert. Zur Vermeidung von Schwierigkeiten beim Lesen und Verstehen des Assembler-Dialekts des 80960 und Vereinfachung der Schreibweise gebe die Prozedur hier nur im Pseudocode an. Nach einem *flushreg* wird der *previous frame pointer* auf den *pfp* des aktuellen Prozesses gesetzt. Der Befehl *flushreg* sorgt dafür, daß der Registercache (die auf dem Stack gesicherten lokalen Register) invalidiert wird, d.h. daß die Register auch wirklich auf den Stack geschrieben werden und beim Rücksprung aus dieser Prozedur auch wirklich vom Stack (und nicht aus dem Cache) restauriert werden. Beim Setzen des *pfp* muß man darauf achten, daß die vier niedrigsten Bits den Typ des Returns angeben (ob man aus einer normalen Prozedur, einer

[3] Die schon vorher existierende Laufzeitumgebung (entspricht der Prozedur main()) wird nicht verwendet.

Interrupt-Service-Routine oder einer Fault-Behandlungs-Prozedur zurückspringt), der natürlich nicht verändert werden soll.

```
.globl _StartCurrentProcess
_StartCurrentProcess: flushreg
    /* pfp := currProcess.vp.regs.pfp while keeping the return status in the 4 least significant
        bits of pfp */
ret
```

Nach *InitProcess* und *InstallProcess* sind die Datenfelder eines Prozesses mit folgenden Werten gefüllt: der *pfp* zeigt auf das erste Byte des Stacks des Prozesses, die ersten drei Doppelwörter am Stack sind so gesetzt, daß das erste Doppelwort (*previous frame pointer*) auf sich selbst zeigt, das zweite (*stack pointer*) auf das 64. Doppelwort und das dritte (*return instruction pointer*) auf die erste Instruktion des Prozesses. Die Kontrollregister *ac, pc* und *tc* werden mit sinnvollen Anfangswerten initialisiert. Der Stack-Frame einer Prozedur bietet Platz für die lokalen Register, die bei einem Aufruf einer weiteren Prozedur in eben diesen Speicherbereich kopiert werden (Stack-Frame des Aufrufers), und für lokale Variablen der Prozedur.

Die grau schattierten Felder stellen Datenfelder des aktuellen Prozesses dar (currProcess.vp.r0..r15, currProcess.vp.g0..g15, currProcess.vp.ac/pc/tc und currProcess.vp.pfp). Weiße Felder sind entweder Stack- oder Codebereich, auch die tatsächlichen Register des 80960 sind weiß gezeichnet. Ein Register r' bezeichnet den alten Wert des Registers r.

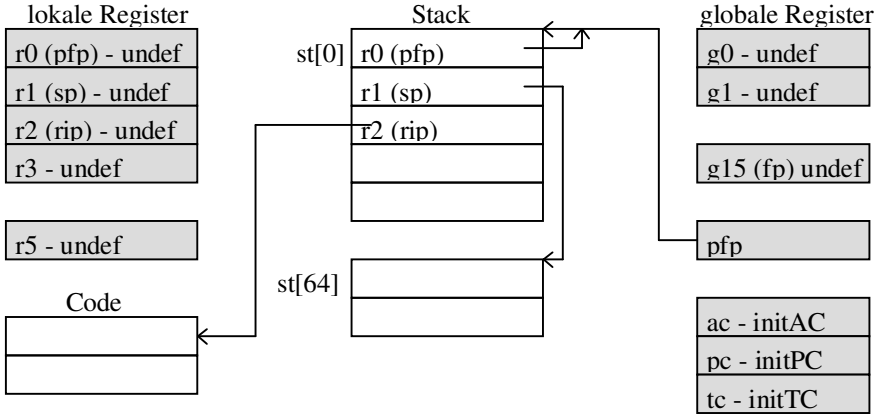

Abbildung 12, Zustand nach InitProcess und InstallProcess

Kurz vor dem Aufruf von *StartCurrentProcess* (also in der Prozedur Start, die der Benutzer zu Beginn in der Prozedur main aufruft) bietet sich uns folgendes Bild: Das Register r0 (*previous frame pointer*) zeigt auf den Stack-Frame des Aufrufers, in unserem Fall ist das der Aufrufer der Prozedur *Start*, also in der Regel die Hauptprozedur des Anwenderprogrammes. Das Register *r1* (*stack pointer*) zeigt auf das letzte Byte des aktuellen Stack-Frames. Das Register *r2* (*return instruction pointer*) zeigt auf den nächsten Befehl, der nach der Abarbeitung der

Prozedur *StartCurrentProcess* ausgeführt werden sollte - das wäre der Befehl nach dem Aufruf von *Start*, der natürlich nie erreicht werden wird. Das Register *g15* (*frame pointer*) zeigt auf den Beginn des aktuellen Stack-Frames.

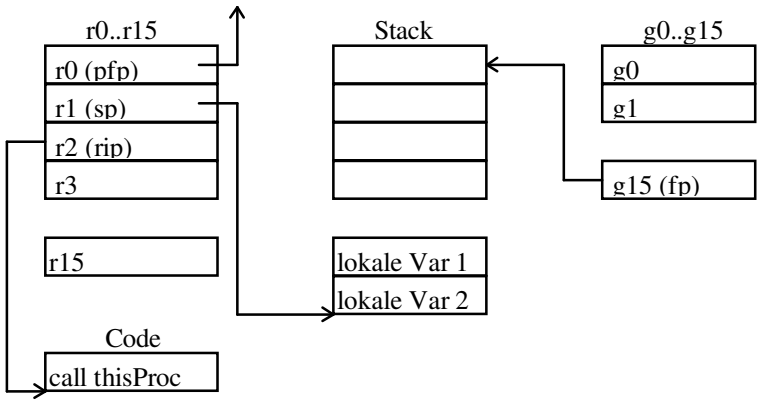

Abbildung 13, Zustand vor dem Aufruf von StartCurrentProcess

Bei einem Aufruf einer Prozedur (call) werden auf dem 80960 folgende Mikroinstruktionen ausgeführt: zuerst wird auf das Ende von gerade parallel ausgeführten Befehlen gewartet. Danach wird die nächste durch 16 teilbare Adresse nach dem aktuellen Stack-Frame berechnet. Im return instruction pointer wird die Adresse der nächsten Instruktion abgelegt. Jetzt werden die lokalen Register in den aktuellen Stack-Frame geschrieben. Der *previous frame pointer* wird auf den *frame pointer* gesetzt, der Return-Typ wird entsprechend eingestellt (normaler Return), der *frame pointer* wird auf den vorher berechneten Hilfswert gesetzt, der *stack pointer* auf das 64. Doppelwort im neuen Stack-Frame. Platz für lokale Variablen muß der Compiler durch Erhöhen des Stackpointers am Anfang der aufgerufenen Prozedur schaffen. Schließlich wird noch der Zeiger auf den nächsten Befehl richtig gesetzt.

0	wait for uncompleted instructions
1	temp := (sp + 16) and not 15
2	rip := next ip
3	mem[fp..fp+63] := r0..r15
4	pfp := fp
5	pfp.rt := 000 / 111 / ...
6	fp := temp
7	sp := temp + 64
8	ip := ip + disp

Tabelle 1, Mikroinstruktionen bei einem Call

Im folgenden Bild sind die Mikroinstruktionen als hochgestellte Indizes dargestellt.

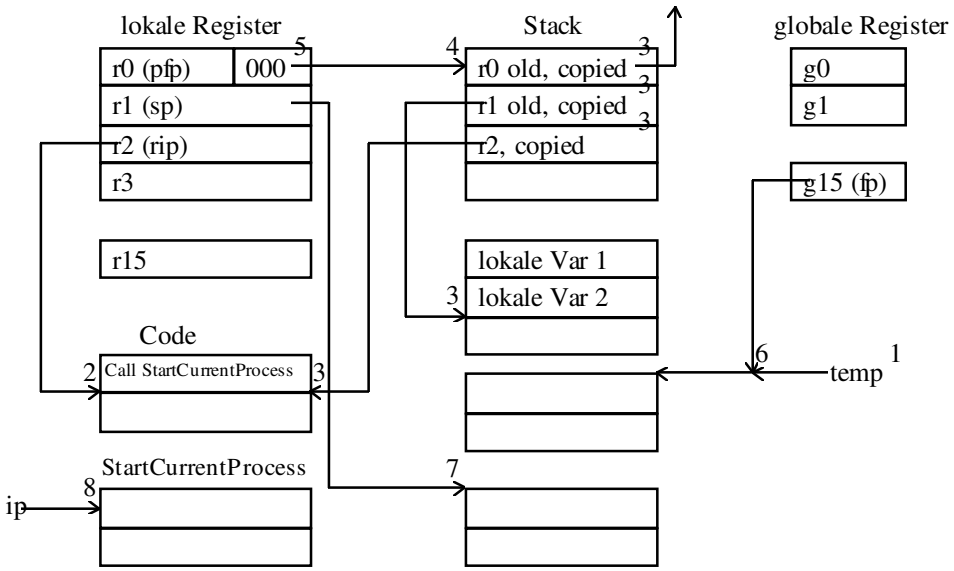

Abbildung 14, Zustand nach dem Aufruf von StartCurrentProcess (Mikroinstruktionen als hochgestellte Indizes)

In *StartCurrentProcess* wird nun nach einem Invalidieren des Registercaches der *previous frame pointer* umgesetzt. Der untere Teil des Bildes ist gleich wie das vorige Bild - mit einem kleinen Unterschied: der *pfp* zeigt bereits auf den Stack des neuen Prozesses.

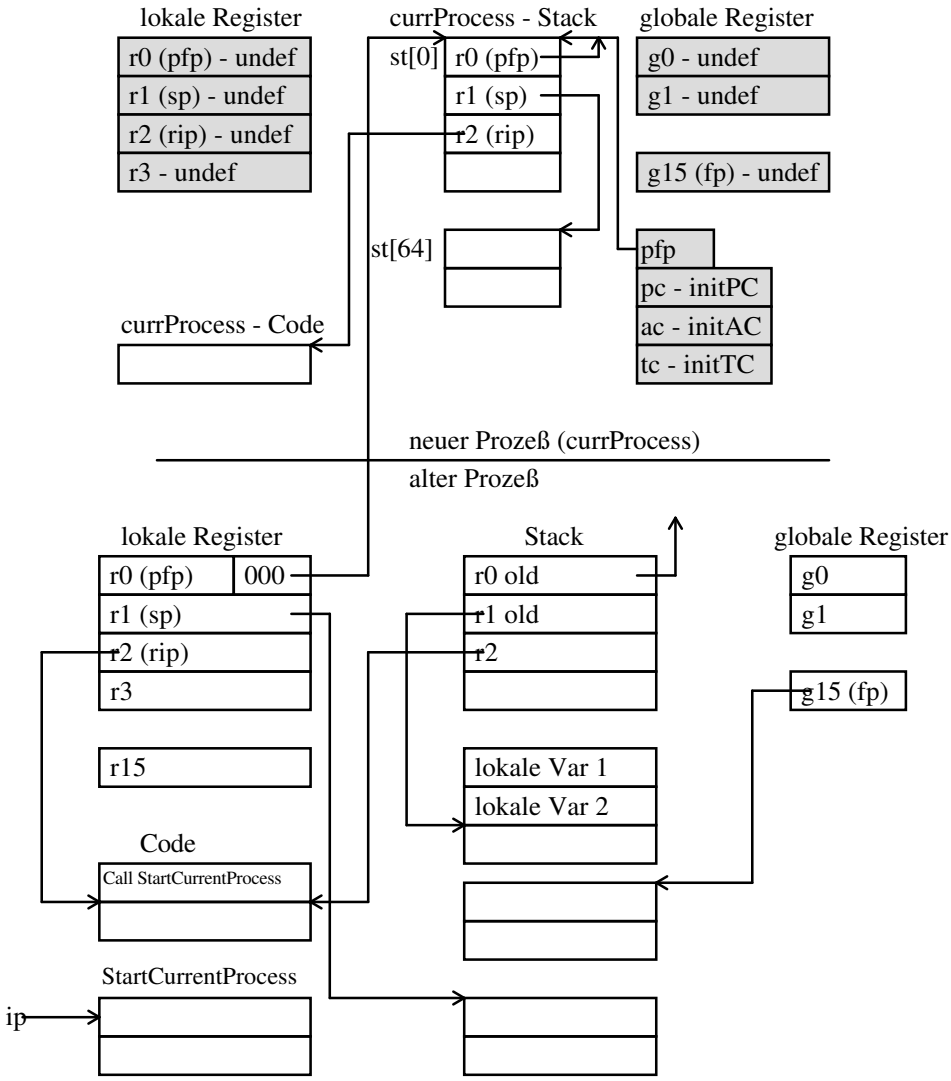

Abbildung 15, Zustand am Ende von StartCurrentProcess

Bei einem Rücksprung aus einer Prozedur (ret) werden folgende Mikroinstruktionen ausgeführt: der *frame pointer* wird auf den *previous frame pointer* gesetzt, die lokalen Register werden aus dem Stack-Frame des Aufrufers (wenn ein *flushreg* passiert ist) oder aus dem Registercache wiederhergestellt und der Zeiger auf die nächste Instruktion wird richtig gesetzt.

1	fp := pfp
2	restore local registers from cache / memory
3	ip := rip

Tabelle 2, Mikroinstruktionen bei einem Return

Im nächsten Bild sieht man, wie zuerst der *fp* umgesetzt wird, und dann vom Stack-Frame des „Aufrufers“ die lokalen Register wiederhergestellt werden. In unserem Fall ist der Aufrufer gar nicht der eigentliche Aufrufer, sondern nur fingiert, da wir ja den *IdleProcess* als ersten Prozeß

im System anspringen. Der Stack-Frame und der Code-Bereich des alten Prozesses werden durch die Register des Prozessors nicht mehr referenziert, womit der Prozeßwechsel vollzogen ist.

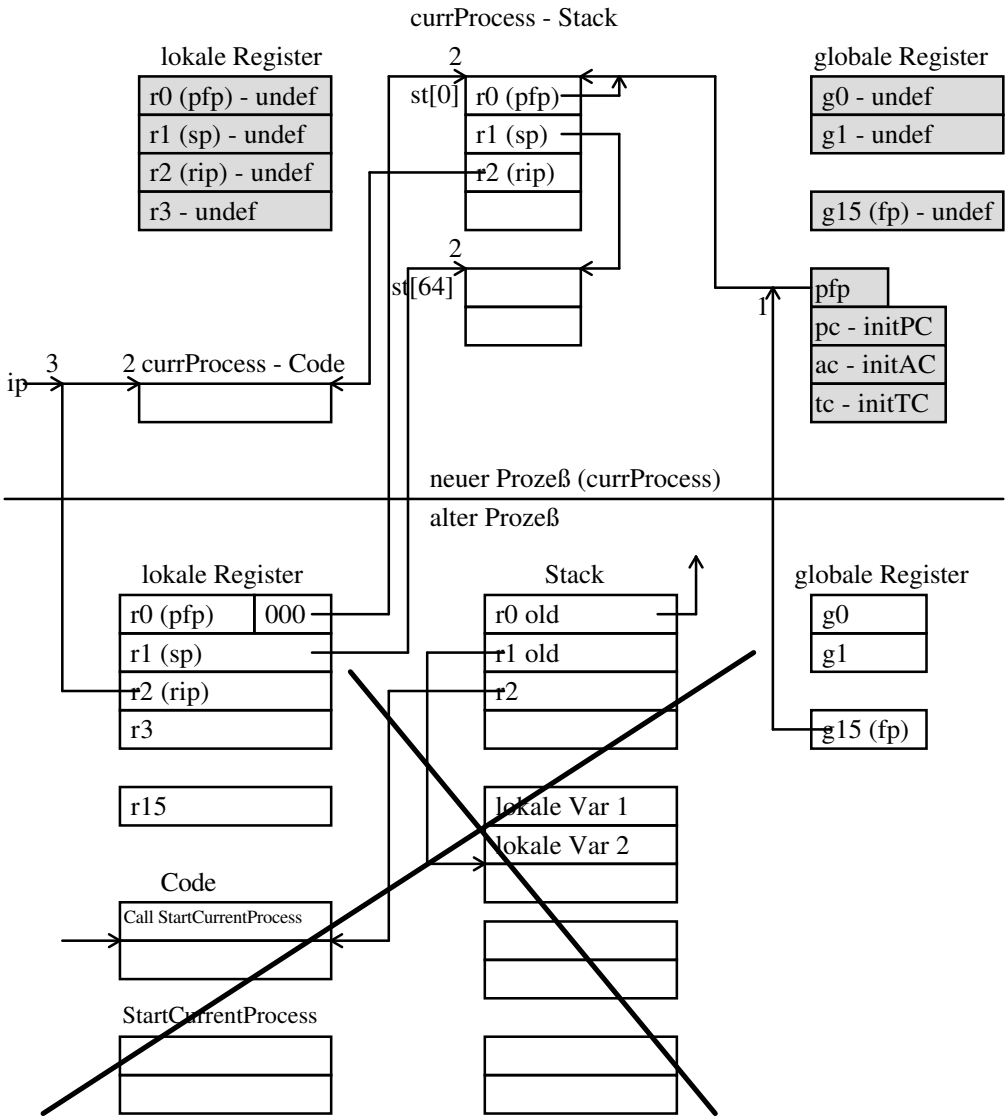

Abbildung 16, Zustand nach dem Return von StartCurrentProcess (Mikroinstruktionen als hochgestellte Indizes)

Prozeßwechsel durch einen Interrupt

Das laufende Programm wird von einem Interrupt unterbrochen. In unserem Fall ist das der periodische Interrupt des Timerbausteins. Wie schon vorher erwähnt, werden bei einem Prozeduraufruf die lokalen Register im Stack-Frame des Aufrufers gesichert. Bei einem Interrupt müssen zusätzlich die globalen Register in den Datenfeldern des virtuellen Prozessors des laufenden Prozesses gesichert werden, da bei Wiederaufnahme der Abarbeitung des unterbrochenen Prozesses dieser im ursprünglichen Zustand weiterarbeiten soll. Danach ruft die Assembler-Interrupt-Service-Routine die Oberon-Scheduling-Prozedur auf, die möglicherweise den Zeiger auf den aktuellen Prozeß verändert. Nach der Rückkehr zur

Assembler-Prozedur werden die lokalen und globalen Register aus den Datenfeldern des virtuellen Prozessors des jetzigen currProcess in die tatsächlichen Register kopiert. Mit einem normalen Return beendet man die Interrupt-Service-Routine und kehrt entweder zum selben Prozeß oder zum neu laufenden Prozeß zurück.

Vor dem Eintreten eines Interrupts haben wir wieder folgendes Bild vor uns:

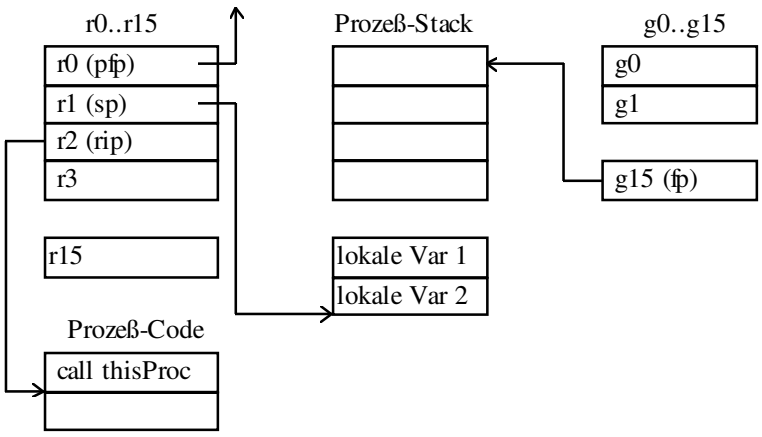

Abbildung 17, Zustand vor Eintreten des Interrupts

Beim Auftreten eines Interrupts werden auf dem 80960 folgende Mikroinstruktionen ausgeführt: Der *return instruction pointer* wird auf die Adresse des nächsten Befehls gesetzt, die lokalen Register werden im Stack-Frame des Aufrufers gesichert, der *previous frame pointer* wird auf den *frame pointer* des Aufrufers gesetzt, der *frame pointer* auf die nächste durch 16 teilbare Adresse auf dem Interruptstack, der *stack pointer* auf das 64. Doppelwort des neuen Stack-Frames, der *instruction pointer* wird auf den ersten Befehl der Interrupt-Service-Routine gesetzt, die Kontrollregister *pc* und *ac* werden im aktuellen Stack-Frame abgelegt, sowie die Nummer des Interrupts. Der Returntyp wird auf „interrupt-return“ gesetzt und gewisse Bits in den *process controls* werden angepaßt.

1	rip := next ip
2	save local registers
3	pfp := current fp
4	fp := (interrupt stack pointer + 16) and not 15
5	sp := fp + 64
6	ip := first instruction of isr (from table)
7	mem[fp-16] := pc
8	mem[fp-12] := ac
9	mem[fp-8] := vector number
10	pfp.rt := 111
11	pc(13) := state interrupted pc(1) := supervisor execution mode pc(16..20) := priority of interrupt (14 for timertick) pc(10, 0) := generate no trace faults, no faults pending

Tabelle 3, Mikroinstruktionen bei einem Interrupt

Im nächsten Bild sehen wir die Abarbeitung dieser Mikroinstruktionen.

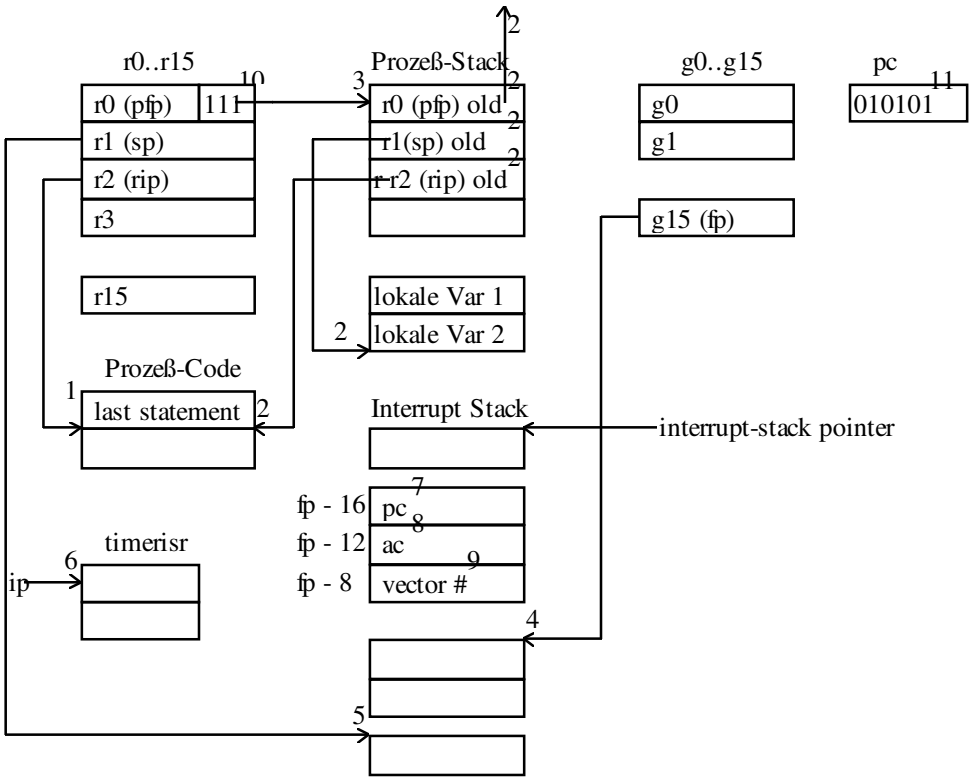

Abbildung 18, Zustand nach Auftreten des Interrupts

Die Timerinterrupt-Service-Routine ist ebenfalls in Assembler geschrieben, jedoch schnell zu erklären: Nach einem *flushreg* (Invalidieren des Registercaches) wird die „Echtzeit“ weitergezählt. Für den Fall, daß die ISR bereits betreten wurde (könnte passieren, wenn die Abarbeitung der ISR länger dauert als die Periode zwischen zwei Interrupts), wird sie sofort wieder verlassen. Ebenfalls verlassen wird die ISR, wenn der Anwender über einen Aufruf von *StopScheduling* das automatische Scheduling unterbunden hat oder wenn durch einen Programmfehler der Zeiger auf den aktuellen Prozeß NIL geworden ist (was einen gravierenden Mißstand darstellt). Wenn man so weit gekommen ist, werden die globalen Register, der aktuelle *previous frame pointer*, die lokalen Register (wäre nicht notwendig, da sie schon auf dem Stack des Aufrufers gerettet sind) und die Kontrollregister in den Datenfeldern des alten Prozesses abgelegt. Nach dem Aufruf der Oberon-Schedulingprozedur werden die globalen Register und die Kontrollregister mit den Werten aus den Datenfeldern des aktuellen Prozesses wiederhergestellt. Nachdem die Oberon-Schedulingprozedur den Zeiger auf den aktuellen Prozeß (*currProcess*) geändert haben kann, kann es sein, daß jetzt ein neuer Prozeß der aktuelle Prozeß ist. Abschließend wird der Zähler, wie oft die ISR betreten wurde, wieder auf ihren ursprünglichen Wert gesetzt. Die lokalen Register werden aus dem Stack-Frame des Aufrufers implizit wiederhergestellt (durch den call-and-return-Mechanismus). Die wichtigste Aktion in der ISR ist wieder das Umsetzen des *previous frame pointer* auf den Stack-Frame des neuen Prozesses.

```
.globl _timerISR
_timerISR: flushreg
```

```
/* INC(realTime) */
/* IF inISRcount > 0 THEN RETURN ELSE oldinISRcount := inISRcount;
     INC(inISRcount) END */
/* IF Kernel.stopSchedulingCount > 0 THEN GOTO endOfISR END */
/* IF currProcess = NIL THEN GOTO endOfISR END */
/* save g0-g15 of the old process */
/* save pfp of the old process */
/* save r0-r15 of the old process, not necessary */
/* save pc, ac and tc of the old process */
call _Objix_timerisr
/* restore pc, ac and tc of the new process */
/* restore g0-g14 of the new process */
/* r0-r15 of the new process are implicitly restored by the ret using the values stored at
     addresses pfp... where they were saved when the interrupt occurred. */
/* restore pfp while keeping the return status in the 4 least significant bits of pfp */
.endOfISR: /* inISRcount := oldinISRcount */
ret
```

Die hochgestellten Indizes zeigen im nächsten Bild das Sichern der globalen Register (1), des *previous frame pointers* (2), der lokalen Register (3) und der Kontrollregister (4) an.

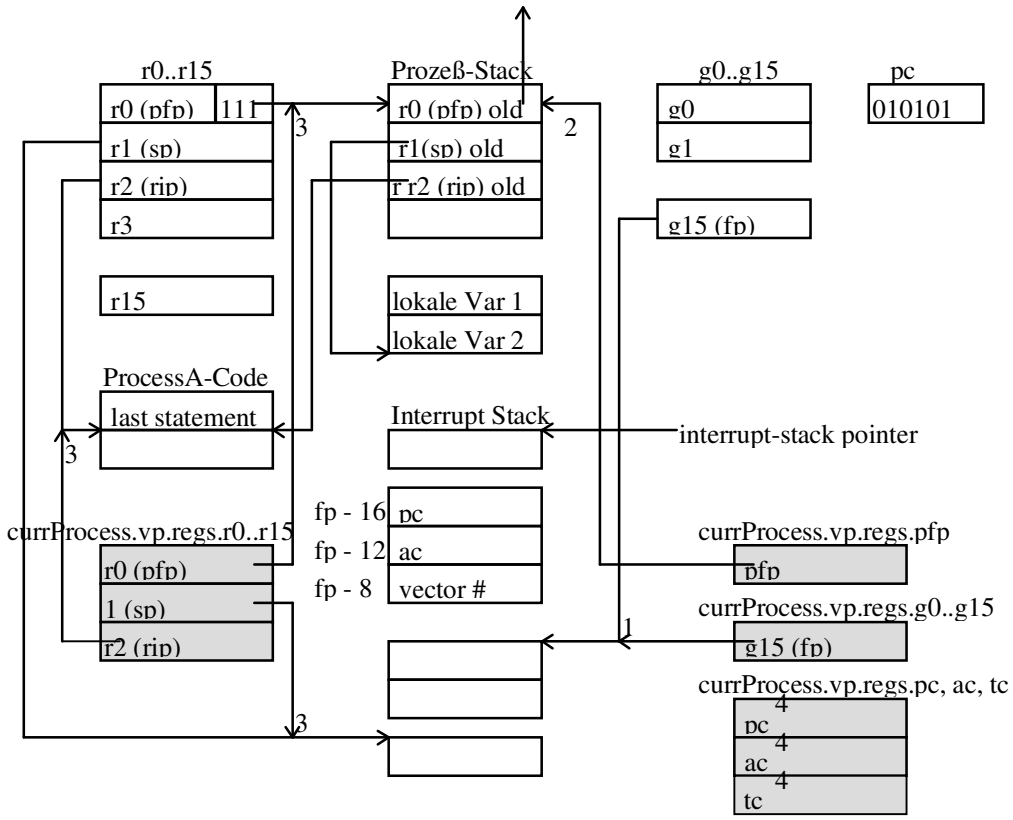

Abbildung 19, Zustand vor Aufruf der Oberon-Schedulingprozedur

Nach der Rückkehr aus der Oberon-Schedulingprozedur ergibt sich folgender Zwischenzustand: die Register des Prozessors haben noch die zum ursprünglichen Prozeß gehörenden Werte, die Datenfelder des aktuellen Prozesses allerdings zeigen auf den eigenen Stack und auf den Codebereich des aktuellen Prozesses.

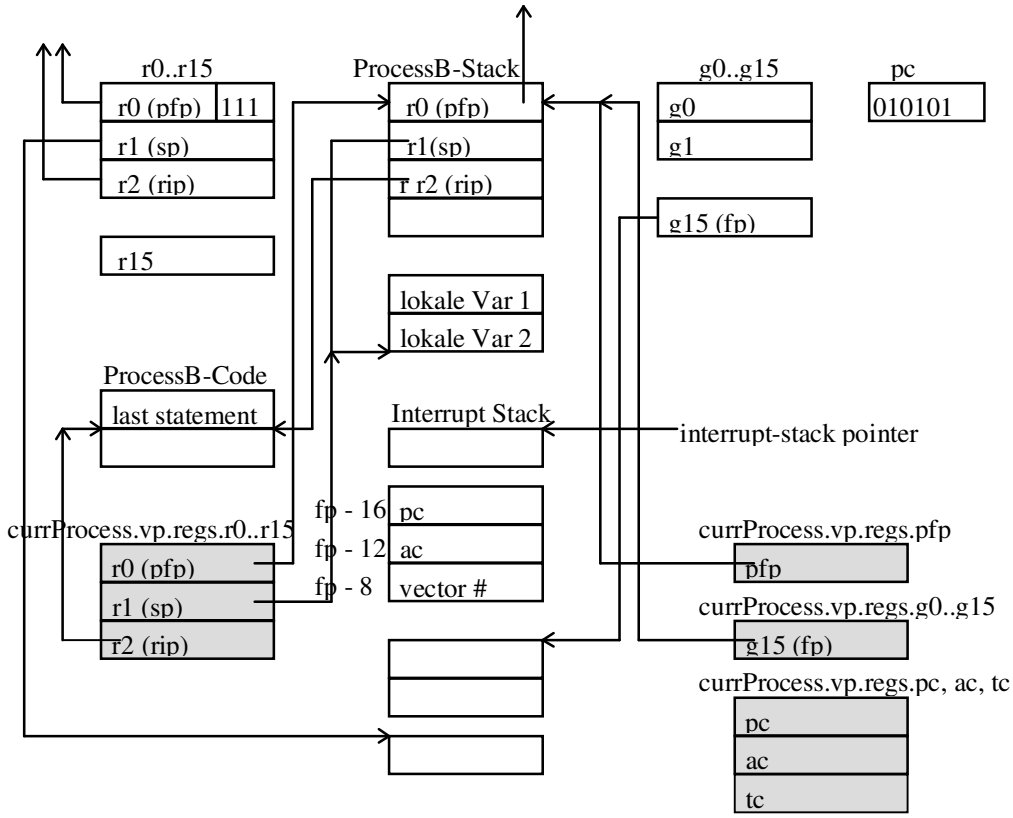

Abbildung 20, Zustand nach Rückkehr aus Oberon-Schedulingprozedur

Die Schritte nach der Rückkehr aus der Oberon-Schedulingprozedur und vor dem Verlassen der ISR sehen wir im nächsten Bild. Die hochgestellten Indizes stellen das Restaurieren der Kontrollregister (1), der globalen Register (2) und des *pfp* (3) dar.

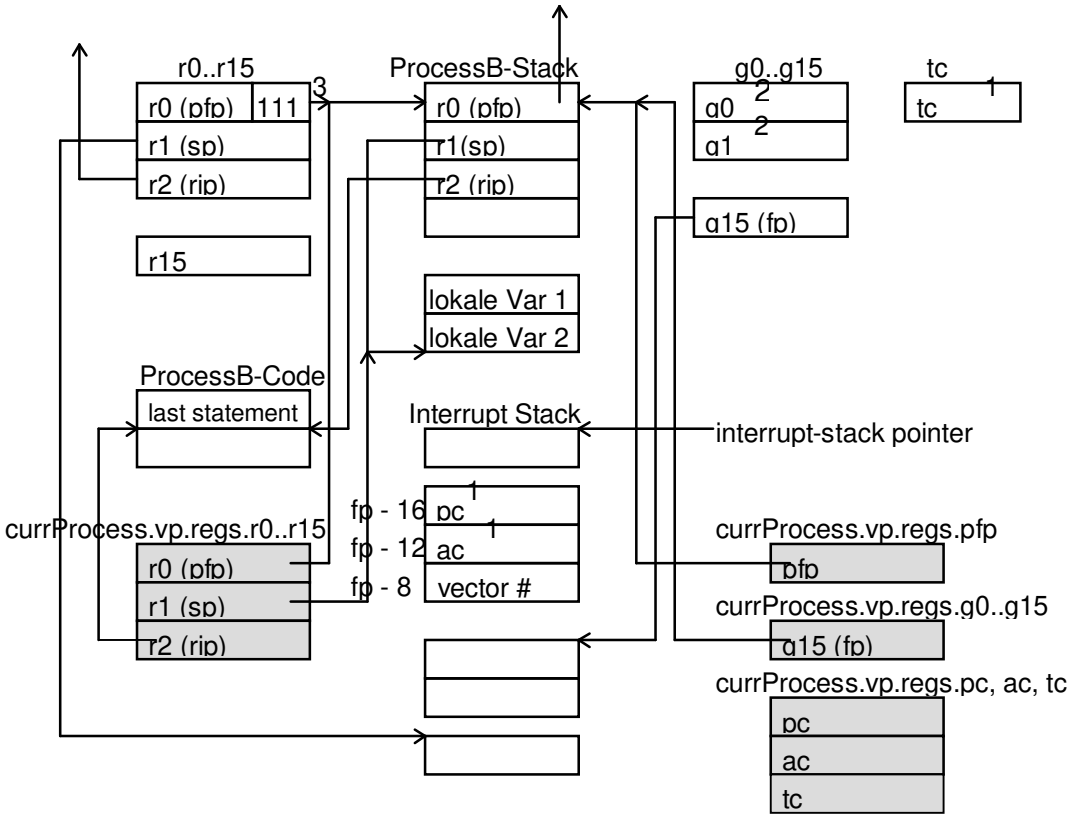

Abbildung 21, Zustand am Ende von timerISR

Bei einem Interupt-Return werden auf dem 80960 folgende Mikroinstruktionen ausgeführt: Die Kontrollregister *ac* und *pc* werden aus dem Interrupt-Rekord wiederhergestellt, der *frame pointer* wird auf den *previous frame pointer* gesetzt, die lokalen Register werden aus dem Stack-Frame des Aufrufers (wenn ein *flushreg* passiert ist) oder aus dem Registercache wiederhergestellt und der Zeiger auf die nächste Instruktion wird richtig gesetzt.

1	ac := ac from interrupt record
2	pc := pc from interrupt record
3	fp := pfp
4	restore local registers from cache / memory
5	ip := rip

Tabelle 4, Mikroinstruktionen bei einem Interrupt-Return

Die Abarbeitung dieser Mikroinstruktionen sehen wir im nächsten Bild. Am Ende haben wir wieder einen konsistenten Zustand erreicht, wobei jetzt der neue Prozeß läuft, die Register des Prozessors zeigen weder auf den Interrupt-Stack, noch auf den Stack des alten Prozesses noch auf den Codebereich des alten Prozesses.

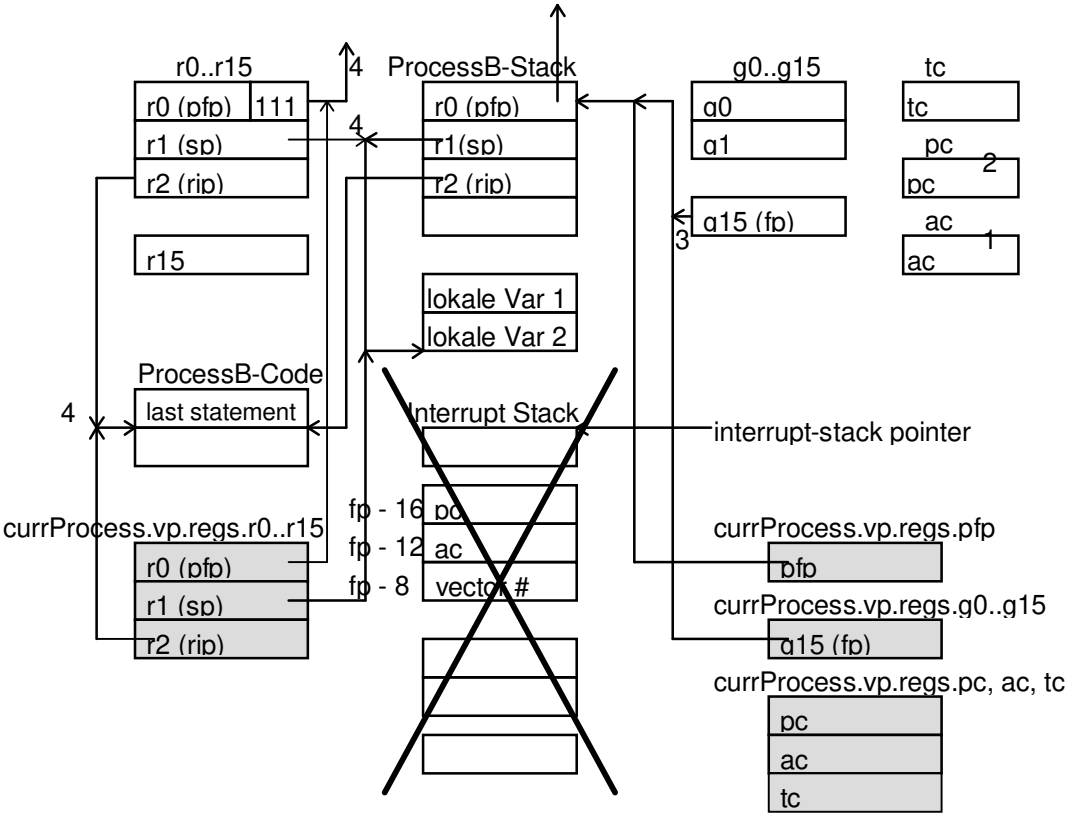

Abbildung 22, Zustand nach dem Interrupt-Return (Mikroinstruktionen als hochgestellte Indizes)

Prozeßwechsel durch ImmediateSwitchUnlock

Die letzte noch zu besprechende Art des Prozeßwechsel ist der freiwillige Prozeßwechsel, der bei einem Aufruf von *Sleep(0), SetPriority, StartScheduling* oder *Semaphore-Wait* passieren kann. Ebenfalls aufgerufen wird *ImmediateSwitchUnlock* bei *Resume*, *Suspend* und *Semaphore-Signal*. In diesen Fällen wechselt man von einem Prozeß zu einem anderen Prozeß, die Prozedur *ImmediateSwitchUnlock* macht also nichts anderes als einen „Koroutinen-Transfer“.

Vor dem Aufruf von ImmediateSwitchUnlock haben wir wieder den nun schon bekannten Zustand.

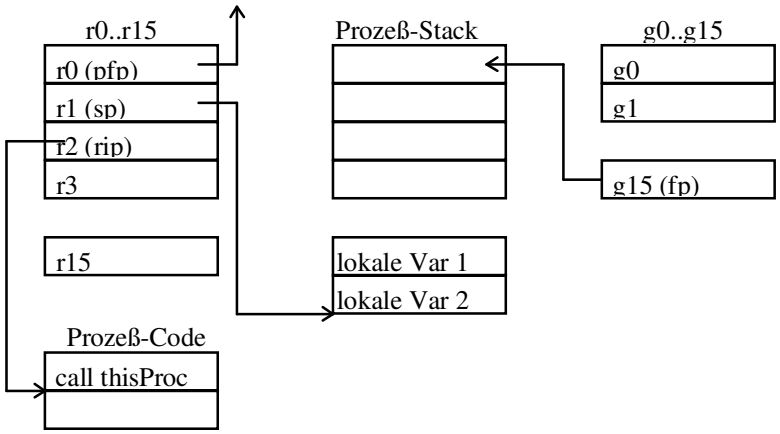

Abbildung 23, Zustand vor Aufruf von ImmediateSwitchUnlock

Beim Aufruf der Prozedur werden wieder die „call“-Mikroinstruktionen ausgeführt.

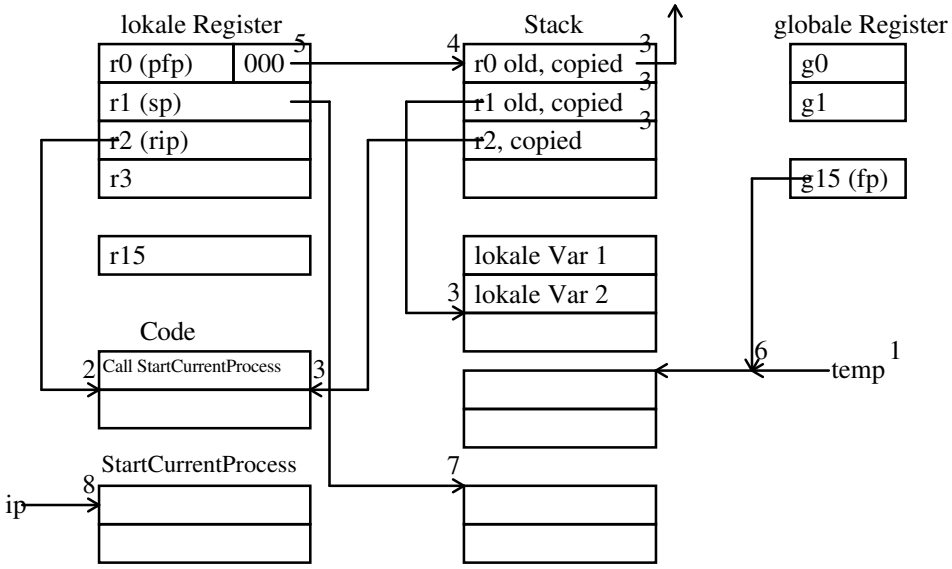

Abbildung 24, Zustand am Beginn von ImmediateSwitchUnlock

Die Prozedur *ImmediateSwitchUnlock* macht folgendes: Sie speichert nach einem Aufruf von *flushreg* die Werte der lokalen und globalen Register, sowie *pfp*, *ac*, *tc* und *pc* in den Datenfeldern des virtuellen Prozessors des Prozesses, der vor dem Scheduling der aktuelle Prozeß war (Objix.oldProcess). Danach werden die Kontrollregister *pc*, *ac* und *tc*, sowie die globalen Register mit den Werten aus den Datenfeldern des virtuellen Prozessors des aktuellen Prozesses (*Objix.currProcess*) geladen. Durch das Setzen des *pfp* auf den Wert des Datenfeldes *pfp* des virtuellen Prozessors und den Mechanismus des automatischen Sicherns bzw. Restaurierens der lokalen Register bei einem call/return werden die lokalen Register automatisch mit den Werten restauriert, die bei der Unterbrechung dieses Prozesses auf dem

Stack gesichert wurden. Abschließend wird noch die Prozedur *OAsmInterface.Unlock* aufgerufen, die das Gegenstück zu *OAsmInterface.Lock* ist und gegebenenfalls das Scheduling wieder freigibt. Eine Precondition von *ImmediateSwitchUnlock* ist, daß *OAsmInterface.Lock* aufgerufen wurde.

```
.globl _ImmediateSwitchUnlock
_ImmediateSwitchUnlock: flushreg
    /* save g0-g15 of the old process */
    /* save pfp of the old process */
    /* save r0-r15 of the old process, not necessary */
    /* save pc, ac and tc of the old process */
    /* restore pc, ac and tc of the new process */
    /* restore g0-g14 of the new process */
    /* r0-r15 of the new process are implicitly restored by the ret using the values stored at
        addresses pfp... where they were saved when ImmediateSwitchUnlock was called. */
    /* restore pfp while keeping the return status in the 4 least significant bits of pfp */
    call _Unlock
ISEnd: ret
```

Im Bild sehen wir das Sichern der globalen Register (1), des *previous frame pointers* (2) und der lokalen Register (3).

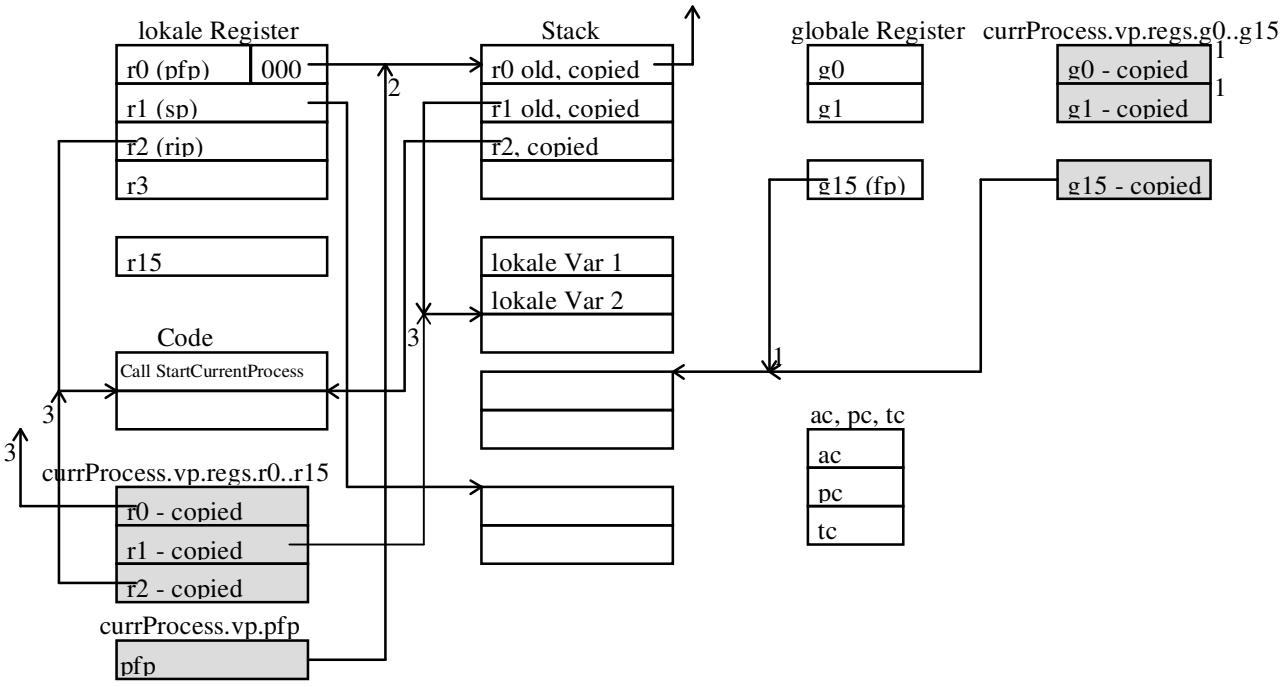

Abbildung 25, Zustand nach Speicherung des Zustandes im alten Prozeß

Anschließend werden die Kontrollregister (1) und die globalen Register (2) mit den Werten aus den Datenfeldern des neuen Prozesses geladen und der *previous frame pointer* (3) wird umgesetzt.

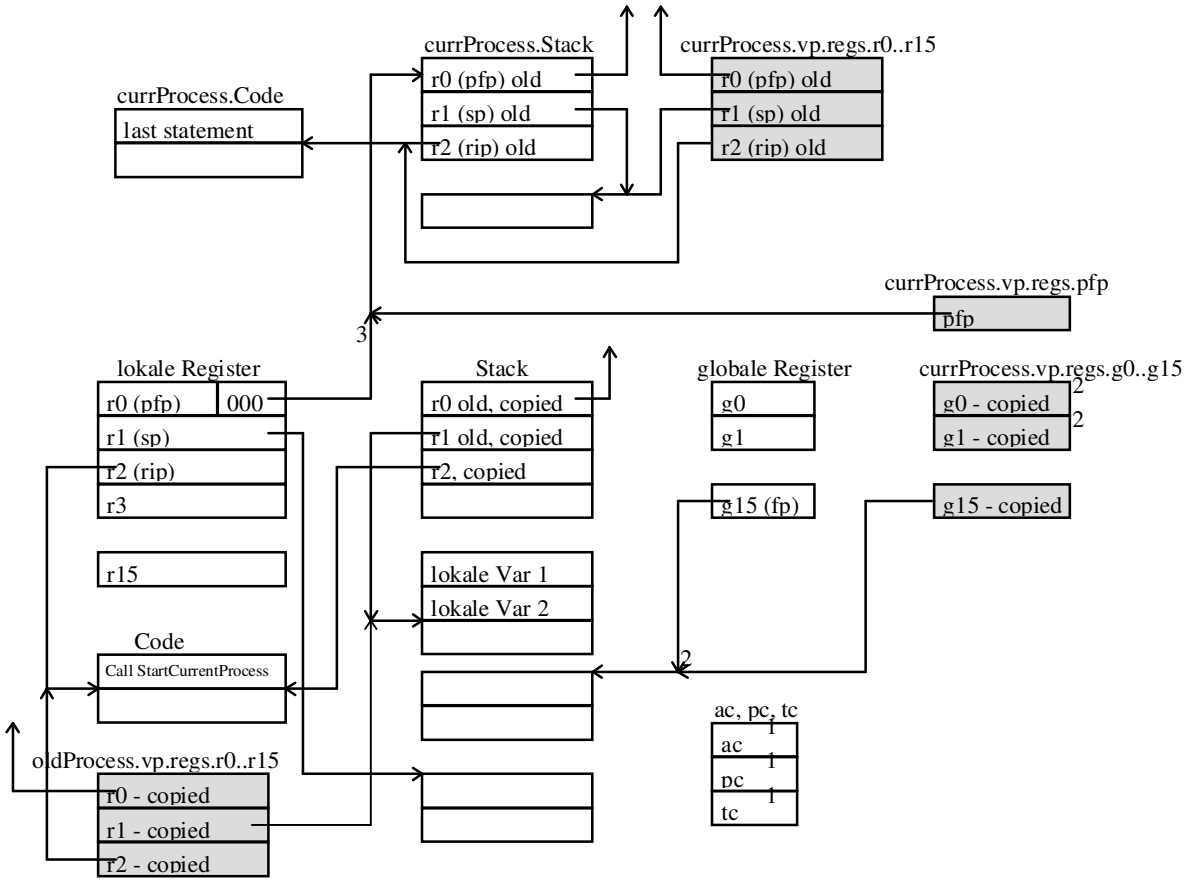

Abbildung 26, Zustand nach Restaurierung des Zustands aus dem neuen Prozeß

Nach dem Aufruf der Prozedur *Unlock* werden beim Return die „ret“-Mikroinstruktionen ausgeführt. Wieder zeigen nach der Abarbeitung von *ImmediateSwitchUnlock* die Register des Prozessors weder auf den alten Stack noch in den alten Codebereich, womit der Prozeßwechsel vollzogen ist.

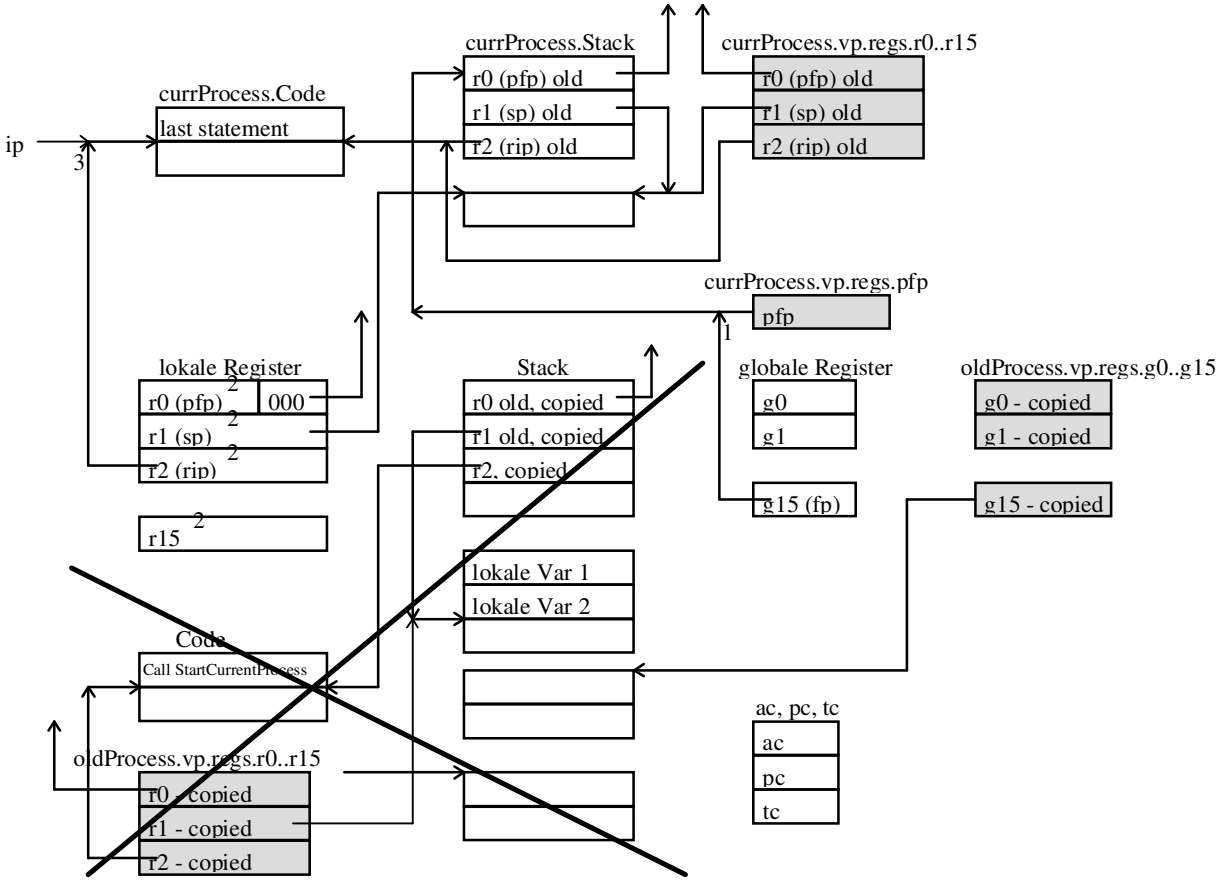

Abbildung 27, Zustand nach dem Return (Mikroinstruktionen als hochgestellte Indizes)

Implementierung im Detail

In diesem Kapitel werden einige wichtige Prozeduren entweder im Detail beschrieben oder direkt im Quellcode angegeben.

Wichtig für den Anwender ist, daß er weiß, was in der Initialisierungsphase geschieht, beziehungsweise, in welcher Reihenfolge was passiert. Deshalb sind die Prozeduren Init, Start, InitSchedulerD, InstallScheduler und InstallProcess genau beschrieben.

Danach folgt eine graphische Darstellung der Aufrufhierarchie (was in welcher Reihenfolge von wem aufgerufen wird). Diese Aufrufhierarchie ist besonders dann wichtig, wenn man durch objektorientierte Erweiterung gewisse Teile des Systems ändern möchte.

Abschließend wird nach einer Beschreibung der Oberon-Schedulingprozedur noch die Implementierung der Alarme und Semaphore besprochen, die Alarme deshalb, weil sie zeigen, mit wie wenig Aufwand es möglich war, dieses Feature in den Mikrokernel einzubauen; die Semaphore deshalb, weil man daran sieht, daß die Implementierung durchaus verständlich ist.

Init (nrOfProcesses: INTEGER; maxElemsQES: LONGINT): BOOLEAN
Init initialisiert einige Datenstrukturen des Kernels, schaltet Überlauferkennung ein, setzt currProcess auf NIL, realTime auf 0 und sichert die aktuellen Werte der Steuerregister AC, PC und TC in initAC, initPC und initTC (werden bei der Initialisierung von Prozessen verwendet). Init liefert TRUE bei Erfolg, sonst FALSE.

Die maximale Anzahl an Prozessen, die der Kernel unterstützen soll, ist der erste Parameter “nrOfProcesses”. Die maximale Anzahl der QueueElements, die für die Speicherung der Prozesse in den internen Datenstrukturen notwendig sind, ist der zweite Parameter “maxElemsQES”, ein sinnvoller Wert für maxElemsQES muß größer als nrOfProcesses + (maxPriority + 1) * 4 sein (z.B. 100).

Start (code: OCT.Code; ticksPerSecond: INTEGER; scheduler: Scheduler)
Start installiert den übergebenen Scheduler (mit einem Aufruf von *InstallScheduler(scheduler, ticksPerSecond)*), der fortan „ticksPerSecond“ Interrupts vom Timerbaustein erhalten soll. Weiters wird der *idleProcess* angelegt, der immer dann aktiv ist, wenn sonst kein Prozeß aktiv ist. Der *idleProcess* erhält die Standard-Zeitscheibe (*OCC.StandardSlice*) und die niedrigste Priorität (*OCC.MinPriority*). Der *idleProcess* ist der Prozeß der am Ende von *Start* durch *StartCurrentProcess* angesprungen wird. Vor dem Aufruf von *StartCurrentProcess* wird der vom Aufrufer spezifizierte Initialisierungscode „code” ausgeführt. In dieser Prozedur alloziert das Anwendungsprogramm typischerweise eigene Ressourcen, legt Prozesse an, installiert sie etc. Das Scheduling wird im *IdleProcess* als erste Aktion aktiviert. Danach wird in einer Endlosschleife die Prozedur *IdleProcessProc* aufgerufen. *IdleProcessProc* ist eine Prozedur-Elementvariable des Schedulers, die vom Anwendungsprogramm auf eine beliebige parameterlose Prozedur gesetzt werden kann. In dieser Prozedur können niederpriore Aufgaben periodisch ausgeführt werden.

InitSchedulerD (VAR s: SchedulerDesc; maxPriority: OCT.Priority)
InitSchedulerD initialisiert den übergebenen Scheduler, setzt die maximal zu unterstützende Priorität auf „maxPriority", initialisiert die Prozedur-Elementvariablen *BeforeProc, AfterProc* und *IdleProcessProc* auf NIL und legt interne Datenstrukturen an. *BeforeProc* und *AfterProc* können vom Anwendungsprogramm auf beliebige parameterlose Prozeduren gesetzt werden, die in der Timer-Interrupt-Behandlungsprozedur am Beginn bzw. am Ende ausgeführt werden. In diesen Prozeduren kann man direkt in das Scheduling eingreifen, muß aber bedenken, daß diese Prozeduren periodisch sehr oft aufgerufen werden und somit laufzeiteffizient implementiert sein müssen.

InstallScheduler (scheduler: Scheduler; ticksPerSecond: INTEGER)
InstallScheduler setzt die modulglobale Variable Kernel (wird exportiert und ist somit vom Anwendungsprogramm aus lesend zugreifbar) auf den übergebenen Scheduler, setzt den Interrupt-Vektor des Timerbausteins auf die Assembler-Schedulingprozedur OAsmInterface.timerISR und initialisiert den Timerbaustein so, daß er „ticksPerSecond" Interrupts pro Sekunde generiert.

(VAR s: SchedulerDesc) InstallProcess (proc: Process; state: OCT.State;
prio: OCT.Priority)
InstallProcess installiert den übergebenen Prozeß im Scheduler als „ready" oder „suspended" mit der übergeben Priorität. Weiters wird auf dem Stack des Prozesses im dritten Doppelwort die Adresse des ersten auszuführenden Befehls eingetragen (siehe "Prozeßwechsel auf dem i960 im Detail").

Oberon Timer-ISR
timerisr ruft als erstes *BeforeProc* (anwenderspezifizierte Prozedur-Elementvariable des Schedulers) auf, erledigt danach notwendige Wakeups durch einen Aufruf von *DoWakeups*, kontrolliert, ob die Zeitscheibe des aktuellen Prozesses abgelaufen ist, macht schließlich (wenn nötig) ein *Reschedule* und ruft abschließend *AfterProc* (anwenderspezifizierte Prozedur-Elementvariable des Schedulers) auf.

Aufrufhierarchie

Die Pfeile in der nächsten Abbildung zeigen jeweils vom Aufrufer zum Aufgerufenen. Die Nummern an den Pfeilspitzen geben die Reihenfolge der Aufrufe an. Wenn an mehreren ausgehenden Pfeilspitzen dieselbe Nummer steht, heißt das, daß eine der Prozeduren mit gleicher Nummer aufgerufen wird (alternativ).

Die grau schattierten Felder zeigen Hotspots an. Als Hotspots bezeichnet man Stellen in einem Framework, die bei Erweiterungen bevorzugt beachtet werden sollen und deshalb vom Framework-Designer flexibel gestaltet sind. Die Prozeduren *BeforeProc* und *AfterProc* werden direkt in der Oberon-Schedulingprozedur vor beziehungsweise nach dem automatischen

Scheduling aufgerufen, die Prozedur *IdleProcessProc* wird, wenn der *IdleProcess* aktiv ist, periodisch aufgerufen.

Wenn man weitere Hardware-Ressourcen mit Prozessen assoziieren will und diese beim Freigeben des Prozesses ebenfalls freigegeben werden sollen, so muß man die Methoden *Kill* und gegebenenfalls *KillAlarms* erweitern.

Für die Implementierung des „priority inheritance protocols“ (siehe Kapitel „Bewertung und Ausblick“) wäre eine Erweiterung der Semaphore-Operationen *Wait* und *Signal* notwendig.

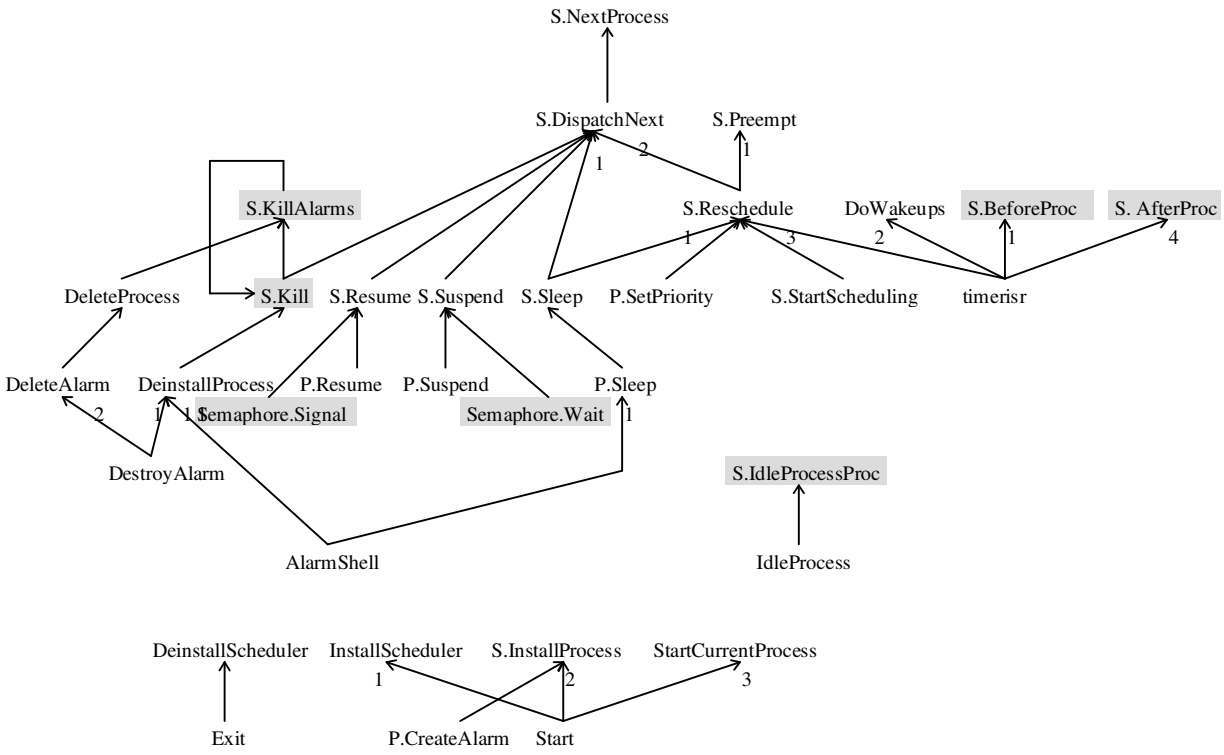

Abbildung 28, Aufrufhierarchie

Alarme

Alarme sind Codeteile, die entweder einmal oder zyklisch ausgeführt werden. Alarme werden normalerweise mit CreateAlarm angelegt. Die Prozeduren InitAlarm und DeleteAlarm werden automatisch aufgerufen und sollten nicht vom Anwendungsprogramm aus aufgerufen werden.

In der folgenden Abbildung sieht man den zeitlichen Ablauf eines zyklischen Alarms (kind = OCC.AlarmCyclic): nach dem Ablauf der Verzögerung wird der Alarm wie ein normaler Prozeß entsprechend seiner maximalen Zeitscheibe geschedulet. Nach der Abarbeitung wird er für die restliche Zeit (cycleTicks - running ticks) schlafend. Wenn er wieder aktiviert wird, wird zuerst die verbliebene Zeitscheibe abgearbeitet. Wenn also für die Abarbeitung eines zyklischen Alarms jeweils 250 Zeiteinheiten benötigt werden, seine Zeitscheibe 100 Einheiten

ist und der Alarm zum Zeitpunkt seiner Ausführung der höchstpriore Prozeß ist, ist seine Zeitscheibe nach der ersten Abarbeitung noch 50 Einheiten groß. Der zweite Zyklus beginnt dann mit der verbliebenen Zeitscheibe von 50 Einheiten.

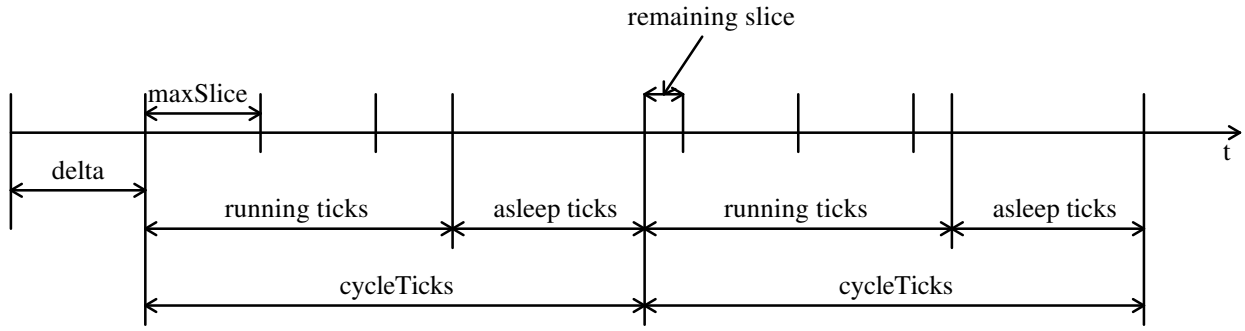

Abbildung 29, zeitlicher Ablauf eines zyklischen Alarms

Ein relativer Alarm (kind = OCC.AlarmRelative) wird nach seiner Abarbeitung automatisch entfernt.

```
Alarm = POINTER TO AlarmDesc;
AlarmDesc = RECORD (ProcessDesc)
   kind-: SHORTINT;
   cycleTicks-: OCT.Ticks;
   parentProcess-: Process;
   alarmCode-: OCT.Code
END;

PROCEDURE InitAlarm (a: Alarm; maxSlice, cycleTicks: OCT.Ticks; code: OCT.Code);
BEGIN
   InitProcess(a, maxSlice, code, OCC.AlarmStackSize);
   a.cycleTicks := cycleTicks;
END InitAlarm;

PROCEDURE DeleteAlarm (a: Alarm);
BEGIN
   DeleteProcess(a)
END DeleteAlarm;

PROCEDURE AlarmShell;
   VAR a: Alarm; cycleStartTime: OCT.Ticks;
BEGIN
   a := currProcess(Alarm);
   IF a.kind = OCC.AlarmCyclic THEN
```

```
        LOOP
            cycleStartTime := realTime;
            a.alarmCode;
            IF realTime - cycleStartTime < a.cycleTicks THEN
                    a.Sleep(a.cycleTicks + cycleStartTime - realTime)
            END
        END
    ELSIF a.kind = OCC.AlarmRelative THEN
        a.alarmCode;
        Kernel.DeinstallProcess(a); (* kills the process and switches to another process *)
    ELSE HALT(85) END;
END AlarmShell;

PROCEDURE (proc: Process) CreateAlarm (kind: SHORTINT; delay, maxSlice, cycleTicks:
OCT.Ticks; priority: OCT.Priority; code: OCT.Code): OCT.AlarmHandle;
    VAR alarm: Alarm;
BEGIN
    ASSERT(delay > 0, 98);                  ⇐   delay muß größer als 0 sein.
    NEW(alarm); InitAlarm(alarm, maxSlice, cycleTicks, AlarmShell);
    Kernel.Lock;                            ⇐   Sperren des Kernels, weil interne
                                                Strukturen verändert werden
    Kernel.InstallProcess(alarm, OCC.StateReady, priority);
    Kernel.ReadyQueues[priority].Remove(alarm.qsid);
    Kernel.AsleepQueues[priority].Enqueue(alarm.qsid);
    alarm.kind := kind;
    alarm.state := OCC.StateAsleep;         ⇐ Prozeß schläft zuerst delay Zeiteinheiten.
    alarm.sleepTicks := delay;
    alarm.alarmCode := code;
    proc.alarms.Enqueue(alarm.qsid);  ⇐ Alarm beim Vaterprozeß vermerken.
    alarm.parentProcess := proc;
    Kernel.Unlock;                          ⇐ Kernel wieder freigeben.
    RETURN alarm.slotID                     ⇐ Alarmhandle zurückgeben.
END CreateAlarm;

PROCEDURE (proc: Process) DestroyAlarm (alarmHandle: OCT.AlarmHandle);
    VAR tmpProcess: Process; alarm: Alarm;
BEGIN
    Kernel.Lock;
    tmpProcess := Kernel.pl.GetProcess(alarmHandle); alarm := tmpProcess(Alarm);
```

```
    IF tmpProcess = currProcess THEN Kernel.Unlock END; (* Kill will call
        ImmediateSwitchUnlock and the following code will not be executed (leaving one
        Kernel.Lock pending) *)
    Kernel.DeinstallProcess(alarm);
    alarm.parentProcess.alarms.Remove(alarm.qsid);
    DeleteAlarm(alarm); SYSTEM.DISPOSE(alarm);
    Kernel.Unlock
END DestroyAlarm;

PROCEDURE (VAR s: SchedulerDesc) KillAlarms (p: Process);
    VAR qdata: QueueData; qsid: QueuedSlotID; tmpProcess: Process; alarm: Alarm;
BEGIN
    WHILE ~p.alarms.Empty() DO
        qdata := p.alarms.Dequeue(); qsid := qdata(QueuedSlotID);
        tmpProcess := s.pl.GetProcess(qsid.slotID); alarm := tmpProcess(Alarm);
        s.Kill(alarm);   (* recursive call for the case that an alarm itself created alarms *)
        DeleteAlarm(alarm); SYSTEM.DISPOSE(alarm)
    END
END KillAlarms;

PROCEDURE (VAR s: SchedulerDesc) Kill (p: Process);
BEGIN  (* this procedure may not be interrupted *)
    IF p.state = OCC.StateZombie THEN RETURN END;
    s.Lock;
    IF p = currProcess THEN
        s.KillAlarms(p);                    ⇐ Erweiterung für Alarme
        s.ReadyQueues[p.prio].Remove(p.qsid);
        p.state := OCC.StateZombie;
        oldProcess := currProcess;
        s.DispatchNext;
        s.ImmediateSwitchUnlock;
        HALT(99) (* never reached *)
    ELSE
        s.KillAlarms(p);                    ⇐ Erweiterung für Alarme
        CASE p.state OF
            OCC.StateReady: s.ReadyQueues[p.prio].Remove(p.qsid)
            | OCC.StateAsleep: s.AsleepQueues[p.prio].Remove(p.qsid)
            | OCC.StateSuspended: s.SuspendedQueues[p.prio].Remove(p.qsid)
            | OCC.StateAsleepSuspended: s.AsleepSuspendedQueues[p.prio].Remove(p.qsid)
        END;
        p.state := OCC.StateZombie;
```

```
        s.Unlock
    END
END Kill;
```

Semaphore

Jede Semaphore hat einen Zähler und eine Warteschlange, in der wartende Prozesse abgelegt werden.

```
Semaphore = POINTER TO SemaphoreDesc;
SemaphoreDesc = RECORD
    qu: FIFOQueue;
    count: INTEGER;
END;
```

Bei der Initialisierung einer Semaphore gibt man einen Anfangswert an, mit dem der mit der Semaphore assoziierte Zähler initialisiert wird. Auf die Semaphore können dann dementsprechend viele Waits ausgeführt werden, bevor ein Wait blockiert.

```
PROCEDURE InitSemaphoreD (VAR s: SemaphoreDesc; initCount: INTEGER);
BEGIN
    NEW(s.qu); InitFIFOQueueD(s.qu^);
    s.count := initCount
END InitSemaphoreD;

PROCEDURE DeleteSemaphoreD (VAR s: SemaphoreDesc);
BEGIN
    DeleteFIFOQueueD(s.qu^); SYSTEM.DISPOSE(s.qu)
END DeleteSemaphoreD;
```

Die Prozedur Wait prüft, ob der mit der Semaphore assoziierte Zähler größer als 0 ist. Wenn er größer als 0 ist, wird der Zähler dekrementiert und fährt normal fort. Wenn der Wert 0 ist, wird der Prozeß in den Zustand *suspended* übergeführt. Das Prüfen des Zählerwertes, das Inkrementieren und das Ändern des Prozeßzustandes passieren in einem Schritt, nichtunterbrechbar. Es ist garantiert, daß kein anderer Prozeß auf die Semaphore zugreift, sobald ein Wait oder Signal initiiert wurde.

```
PROCEDURE (VAR s: SemaphoreDesc) Wait;
BEGIN
    Kernel.Lock;
    IF s.count > 0 THEN
        DEC(s.count);
        Kernel.Unlock;
```

```
    ELSE
        (* suspend current process (process must be the running one), enqueue current process
            in the "queue of suspended processes" of the mutex itself *)
        oldProcess := currProcess;
        Kernel.Suspend(oldProcess);
        s.qu.Enqueue(oldProcess.qsid);
        Kernel.ImmediateSwitchUnlock
    END
END Wait;
```

Die Prozedur Signal inkrementiert den mit der Semaphore assoziierten Zähler. Wenn einer oder mehrere Prozesse durch ein Wait unterbrochen wurden, wird derjenige Prozeß, der als erster unterbrochen wurde, wiederaufgenommen. Solange zur Zeit der Initiierung eines Signals Prozesse (die durch ein Wait unterbrochen wurden) auf die Wiederaufnahme warten, wird nach der Abarbeitung des Signal der mit der Semaphore assoziierte Zähler weiterhin 0 sein, allerdings ist ein Prozeß weniger unterbrochen. Auch das Inkrementieren des Zählers und Wiederaufnehmen eines Prozesses ist nichtunterbrechbar.

```
PROCEDURE (VAR s: SemaphoreDesc) Signal;
    VAR qdata: QueueData; p: Process;
BEGIN
    Kernel.Lock;
    IF s.qu.Empty() THEN
        INC(s.count);
        Kernel.Unlock;
    ELSE
        qdata := s.qu.Dequeue(); p := Kernel.pl.GetProcess(qdata(QueuedSlotID).slotID);
        oldProcess := currProcess;
        Kernel.Resume(p);
        IF oldProcess # currProcess THEN
            Kernel.ImmediateSwitchUnlock
        ELSE
            Kernel.Unlock
        END
    END;
END Signal;
```

Anwendungen

Einfache Testfälle zum Anfangen

Im folgenden Programm soll beispielhaft gezeigt werden, wie man parallele Programme mit dem Mikrokernel implementieren kann. Wichtige Stellen im Code sind jeweils fett gedruckt und mit einem Pfeil und einer Bemerkung versehen.

```
MODULE UseObjix;

IMPORT OCC := OCommonConstants, OCT := OCommonTypes, OBoard := OBoardSpecific, Objix, Out := FOut960, SYSTEM, OAsmInterface;

CONST
   NoError = 0; ErrorInit = 1;
   maxPriority = 9;

TYPE
   TestProcess = POINTER TO TestProcessDesc;
   TestProcessDesc = RECORD (Objix.ProcessDesc)      ⇐ abgeleiteter Prozeß
      i: INTEGER;
   END;

VAR
   proc1, proc2, proc3, proc4: TestProcess;

PROCEDURE UserUseCase1A; (* Dieser Prozeß schaltet in einer Endlosschleife die LED 2, 3, 4 oder 5 auf dem Entwicklungsboard ein, verbraucht einige Zeit und versucht dann, sich schlafen zu legen. Ein Sleep(0) gibt den Prozessor ab, schaut nach, ob andere Prozesse laufend werden sollen, und läßt den höchstprioren weiterlaufen. *)
   VAR i, x: LONGINT; this: TestProcess;
BEGIN
   this := Objix.currProcess(TestProcess);
   LOOP
      ASSERT(Objix.currProcess = this);          ⇐ muß immer gelten
      OBoard.LED({2}, OBoard.set);               ⇐ LED 2, 3, 4 oder 5 einschalten
      FOR i := 0 TO 1000000 DO x := 2 END;       ⇐ etwas Zeit verbrauchen
      this.Sleep(0);                             ⇐ Rescheduling initiieren
   END
END UserUseCase1A;
```

Die Prozesse UserUseCase1B bis UserUseCase1D unterscheiden sich von UserUseCase1A nur in der Zeile „OBoard.LED({i}, Oboard.set)“, i ist dann entsprechend 3, 4 oder 5.

```
PROCEDURE UseCase1; (* In diesem Testfall werden 4 Prozesse angelegt, jeweils mit der
Standard-Zeitscheibe und minimaler Priorität. Diese Prozesse lassen jeweils eine andere
Leuchtdiode auf dem Entwicklungsboard leuchten. Da die Prozesse im Round-Robin-
Verfahren geschedult werden und die niedrigste Priorität haben, werden sie abwechselnd
laufend (auch der IdleProcess des Systems kommt an die Reihe). *)
BEGIN
   Objix.outOptions := {Objix.Print, Objix.PrintRQ};
   NEW(proc1);
   Objix.InitProcess(proc1, OCC.StandardSlice, UserUseCase1A, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc1, OCC.StateReady, OCC.MinPriority);
   NEW(proc2);
   Objix.InitProcess(proc2, OCC.StandardSlice, UserUseCase1B, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc2, OCC.StateReady, OCC.MinPriority);
   NEW(proc3);
   Objix.InitProcess(proc3, OCC.StandardSlice, UserUseCase1C, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc3, OCC.StateReady, OCC.MinPriority);
   NEW(proc4);
   Objix.InitProcess(proc4, OCC.StandardSlice, UserUseCase1D, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc4, OCC.StateReady, OCC.MinPriority);
END UseCase1;

PROCEDURE UserUseCase2A; (* Dieser Prozeß schaltet in einer Endlosschleife die LED 2
ein, verbraucht etwas Zeit, schaltet dann die LED 3 ein, verbraucht wieder etwas Zeit und
suspendiert sich selbst. *)
   VAR i, x: LONGINT; this: TestProcess;
BEGIN
   this := Objix.currProcess(TestProcess);
   LOOP
      ASSERT((Objix.currProcess = this) & (Objix.currProcess = proc1));
      OBoard.LED({2}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      OBoard.LED({3}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      this.Suspend(this);
   END
END UserUseCase2A;

PROCEDURE UserUseCase2B; (* Dieser Prozeß schaltet in einer Endlosschleife die LED 4
ein, verbraucht dann etwas Zeit, schaltet dann die LED 5 ein, verbraucht wieder etwas Zeit,
```

```
legt sich für 200 Zeiteinheiten schlafen und nimmt dann bei jedem zweiten Schleifendurchlauf
den ersten Prozeß wieder auf. *)
   VAR count, i, x: LONGINT; this: TestProcess;
BEGIN
   count := 0;
   this := Objix.currProcess(TestProcess);
   LOOP
      ASSERT((Objix.currProcess = this) & (Objix.currProcess = proc2));
      OBoard.LED({4}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      OBoard.LED({5}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      this.Sleep(200);
      INC(count);
      IF count MOD 2 = 0 THEN Objix.Kernel.Resume(proc1) END;
   END
END UserUseCase2B;

PROCEDURE UseCase2; (* In diesem Testfall werden 2 Prozesse angelegt, jeweils mit der
Standard-Zeitscheibe und mittlerer Priorität. Prinzipiell werden diese zwei Prozesse im
Round-Robin-Verfahren gleichzeitig abgearbeitet, die LEDs 2-4 sollten abwechselnd
leuchten. Da sich der erste Prozeß allerdings periodisch suspendiert und der zweite Prozeß
den ersten nur bei jedem zweiten Schleifendurchlauf wiederaufnimmt, kommt der zweite
Prozeß (trotz Schlafphase) doppelt so oft zum Zug wie der erste. *)
BEGIN
   Objix.outOptions := {Objix.Print, Objix.PrintRQ, Objix.PrintSQ};
   NEW(proc1);
   Objix.InitProcess(proc1, OCC.StandardSlice, UserUseCase2A, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc1, OCC.StateReady, 5);
   NEW(proc2);
   Objix.InitProcess(proc2, OCC.StandardSlice, UserUseCase2B, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc2, OCC.StateReady, 5);
END UseCase2;

PROCEDURE Alarm;
   VAR i, x: LONGINT;
BEGIN
   Out.Ln; Out.String("Alarm"); Out.Ln;
   Objix.PrintQueues;
   OBoard.LED({5}, Oboard.set);
   FOR i := 0 TO 1000000 DO x := 2 END
```

```
END Alarm;

PROCEDURE UserUseCase3; (* Dieser Prozeß erstellt einen Alarm, der relativ zum
Zeitpunkt der Erstellung in 1000 Zeiteinheiten einmalig ausgeführt wird. Der Alarm wird mit
der Standard-Zeitscheibe und Priorität 8 abgearbeitet werden, der zum Alarm gehörende
Code ist die parameterlose Prozedur Alarm. Nach der Erstellung dieses Alarms gibt der
Prozeß in einer Endlosschleife den Zustand der Scheduling-Queues aus, schaltet die LED 3
ein, verbraucht etwas Zeit, schaltet die LED 4 ein, verbraucht etwas Zeit und legt sich für 500
Zeiteinheiten schlafen. *)
   VAR i, x: LONGINT; this: TestProcess; a: OCT.AlarmHandle;
BEGIN
   this := Objix.currProcess(TestProcess);
   a := this.CreateAlarm(OCC.AlarmRelative, 1000, OCC.StandardSlice, 1000, 8, Alarm);
   LOOP
      Out.Ln; Out.String("UserUseCase3 - "); Out.Int(this.i, 0); Out.Ln;
      Objix.PrintQueues;
      OBoard.LED({3}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      OBoard.LED({4}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      this.Sleep(500);
   END
END UserUseCase3;

PROCEDURE UseCase3; (* In diesem Testfall wird ein Prozeß mit der Standard-Zeitscheibe
und mittlerer Priorität angelegt Der Prozeß selber erstellt einen relativen Alarm. Der Prozeß
wird in seiner Abarbeitung durch den von ihm installierten Alarm unterbrochen, da dieser
eine höhere Priorität hat. *)
BEGIN
   Objix.outOptions := {Objix.Print, Objix.PrintRQ, Objix.PrintAQ, Objix.PrintSQ};
   NEW(proc1); proc1.i := 1;
   Objix.InitProcess(proc1, OCC.StandardSlice, UserUseCase3, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc1, OCC.StateReady, 5);
   Out.String("proc1="); Out.Int(SYSTEM.VAL(LONGINT, proc1), 0); Out.Ln;
END UseCase3;

PROCEDURE UserUseCase4; (* Dieser Prozeß erstellt einen Alarm, der relativ zum
Zeitpunkt der Erstellung in 1000 Zeiteinheiten zyklisch ausgeführt werden soll. Der Alarm
wird dann mit der Standard-Zeitscheibe und mittlerer Priorität in Intervallen von 1000
Zeiteinheiten aktiv werden. Nach der Erstellung dieses Alarms gibt der Prozeß in einer
Endlosschleife den Zustand der Scheduling-Queues aus, schaltet die LED 3 ein, verbraucht
```

```
etwas Zeit, schaltet die LED 4 ein, verbraucht etwas Zeit und legt sich für 500 Zeiteinheiten schlafen.*)
   VAR i, x: LONGINT; this: TestProcess; a: OCT.AlarmHandle;
BEGIN
   this := Objix.currProcess(TestProcess);
   a := this.CreateAlarm(OCC.AlarmCyclic, 1000, OCC.StandardSlice, 1000, 5, Alarm);
   LOOP
      Out.Ln; Out.String("UserUseCase4 - "); Out.Int(this.i, 0); Out.Ln;
      Objix.PrintQueues;
      OBoard.LED({3}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      OBoard.LED({4}, OBoard.set);
      FOR i := 0 TO 1000000 DO x := 2 END;
      this.Sleep(500);
   END
END UserUseCase4;

PROCEDURE UseCase4; (* In diesem Testfall wird ein Prozeß mit der Standard-Zeitscheibe und mittlerer Priorität angelegt, der selber einen zyklischen Alarm erstellt. Da der Alarm und der Prozeß beide die gleiche Priorität haben, werden sie, wenn sie beide „ready“ sind, nach dem Round-Robin-Verfahren (gemäß ihren Zeitscheiben) abgearbeitet. *)
BEGIN
   Objix.outOptions := {Objix.Print, Objix.PrintRQ, Objix.PrintAQ, Objix.PrintSQ};
   NEW(proc1); proc1.i := 1;
   Objix.InitProcess(proc1, OCC.StandardSlice, UserUseCase4, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc1, OCC.StateReady, 5);
   Out.String("proc1="); Out.Int(SYSTEM.VAL(LONGINT, proc1), 0); Out.Ln;
END UseCase4;

PROCEDURE UserUseCase5; (* Dieser Prozeß stoppt in einer Endlosschleife das automatische Scheduling, schaltet dann die LED 3 ein, verrichtet eine lange dauernde Aktion, schaltet das automatische Scheduling wieder ein und legt sich dann für 300 Zeiteinheiten schlafen. Im Normalfall sollten Anwenderprozesse das automatische Scheduling nicht abschalten, zumindest nicht wie hier für eine lange dauernde Aktion. *)
   VAR i, x: LONGINT;
BEGIN
   LOOP
      Objix.Kernel.StopScheduling;
      ASSERT(Objix.currProcess = proc1);
      OBoard.LED({3}, OBoard.set);
      FOR i := 0 TO 2000000 DO x := 2 END;
```

```
      Objix.Kernel.StartScheduling;
      Objix.currProcess.Sleep(300);
   END
END UserUseCase5;

PROCEDURE UseCase5; (* In diesem Testfall wird ein Prozeß mit Standard-Zeitscheibe und
mittlerer Priorität angelegt. Dieser Prozeß schaltet periodisch das automatische Scheduling
für eine längere Zeit ab. Würde es noch weitere Prozesse im System geben, würden diese nicht
geschedult (auch nicht höherpriore). Die einzige Art, wie der Prozeß den Prozessor abgeben
kann, ist durch „freiwilligen“ Prozeßwechsel infolge eines Aufrufs von blockierenden
Kernelfunktionen. Dann hat man allerdings das Problem, daß der zum StopScheduling
gehörende StartScheduling Aufruf nicht mehr ausgeführt wird. Deshalb ist dringend davon
abzuraten, in Bereichen ohne automatisches Scheduling freiwilliges Scheduling zu
provozieren (z.B. durch ein Sleep(0) oder durch ein Semaphore-Wait. *)
BEGIN
   Objix.outOptions := {Objix.PrintRQ};
   NEW(proc1);
   Objix.InitProcess(proc1, OCC.StandardSlice, UserUseCase5, OCC.ProcessStackSize);
   Objix.Kernel.InstallProcess(proc1, OCC.StateReady, 5);
   Out.String("proc1="); Out.Int(SYSTEM.VAL(LONGINT, proc1), 0); Out.Ln;
END UseCase5;

PROCEDURE InitApplication;
BEGIN
   UseCaseX                    ⇐ statt dem X steht hier die Nummer des jeweiligen UseCases
END InitApplication;

PROCEDURE main+(): LONGINT;          ⇐ „+“ für C-kompatible Deklaration
   VAR scheduler: Objix.Scheduler;
BEGIN
   Out.Open("objix1.txt");          ⇐ Die Ausgaben werden in die Datei „objix1.txt“
      geschrieben.
   IF Objix.Init(5, 100) THEN       ⇐ maximal 5 Prozesse (und maximal 100 Elemente im
      intern verwendeten QueueElementSpace)
      NEW(scheduler); Objix.InitSchedulerD(scheduler^, maxPriority);
      Objix.Start(InitApplication, 1000, scheduler);   ⇐ 1000x pro Sekunde
         automatisch schedulen, die Prozedur InitApplication wird in dier
         Initialisierungsphase des Kernels automatisch aufgerufen.
      RETURN NoError
   ELSE
      Out.Ln; Out.String("Error when initializing the Objix modules");
```

```
        Objix.Exit;
        RETURN ErrorInit
    END
END main;

END UseObjix.
```

Producer / Consumer-Problem

Beim Producer / Consumer-Problem laufen parallel zwei Tasks ab. Der „producer“ erzeugt gewisse „items“, die der „consumer“ aufbraucht. Die beiden Tasks sind jeweils als parameterlose Prozeduren kodiert, die in einer LOOP-Schleife ihre Arbeit verrichten. Weiters wird in diesem Beispiel gezeigt, wie man eine Erweiterung des normalen Prozesses (Objix.ProcessDesc) definiert und einen erweiterten Prozeß im Scheduler anmeldet. Die Mutual-Exclusion-Variable *bufProtect* wird zum Schutz vor gleichzeitigem Zugriff auf den gemeinsamen Puffer verwendet. Die zwei Semaphor-Variablen *full* und *empty* zeigen den Füllstand des Buffers an: *full* gibt die Anzahl der zu lesenden Elemente im Puffer an (anfangs 0, wird beim Erzeugen hinaufgezählt, beim Aufbrauchen hinabgezählt), *empty* gibt die Anzahl der freien Stellen im Puffer an (anfangs gleich der Puffergröße, beim Aufbrauchen hinaufgezählt, beim Erzeugen hinabgezählt).

Die beiden Tasks werden mit maximaler Timeslice (OCC.MaxSlice), mittlerer Priorität (5 im Bereich von 0..9) und im Zustand OCC.StateReady erstellt.

```
MODULE ProducerConsumer;

IMPORT OCC := OCommonConstants, OCT := OCommonTypes, OBoard :=
   OBoardSpecific, Objix, Out, SYSTEM, OAsmInterface;

CONST
   NoError = 0; ErrorInit = 1;

   maxPriority = 9;

TYPE
   TestProcess = POINTER TO TestProcessDesc;
   TestProcessDesc = RECORD (Objix.ProcessDesc)      ⇐ abgeleiteter Prozeß
      i: INTEGER
   END;

VAR
   proc1, proc2: TestProcess;
   bufProtect: Objix.Mutex;
   full, empty: Objix.Semaphore;
   buf: ARRAY 10 OF LONGINT;
   bufIn, bufOut: INTEGER;

PROCEDURE Producer;
   VAR count: LONGINT; this: TestProcess;
BEGIN
```

```
    count := 0;
    this := Objix.currProcess(TestProcess);        ⇐ Typ-Guard auf erweiterten Typ, liefert
        bei Nicht-Erfüllung einen Laufzeit-Fehler
    LOOP
      ASSERT(Objix.currProcess = this);    ⇐ muß immer gelten
      INC(count);
      Out.Ln; Out.String("Produced "); Out.Int(count, 0);
      OBoard.LED({4}, OBoard.set);
      empty.Wait;                          ⇐ auf  freien Platz warten
      bufProtect.Wait;                     ⇐ für geschützten Zugriff auf gemeinsamen
          Puffer
      buf[bufIn] := count;                 ⇐ Zugriff auf gemeinsamen Puffer
      bufIn := (bufIn + 1) MOD 10;
      bufProtect.Signal;
      full.Signal                          ⇐ neues Element anzeigen
    END
END Producer;

PROCEDURE Consumer;
    VAR count: LONGINT; this: TestProcess; c: CHAR;
BEGIN
    this := Objix.currProcess(TestProcess);        ⇐ Typ-Guard auf erweiterten Typ, liefert
        bei Nicht-Erfüllung einen Laufzeit-Fehler
    LOOP
      ASSERT(Objix.currProcess = this);    ⇐ muß immer gelten
      full.Wait;                           ⇐ auf  Element warten
      bufProtect.Wait;                     ⇐ für geschützten Zugriff auf gemeinsamen
          Puffer
      count := buf[bufOut];                ⇐ Zugriff auf gemeinsamen Puffer
      bufOut := (bufOut + 1) MOD 10;
      bufProtect.Signal;
      empty.Signal;                        ⇐ freien Platz anzeigen
      Out.Ln; Out.String("Consumed "); Out.Int(count, 0);
      OBoard.LED({5}, OBoard.set)
    END
END Consumer;

PROCEDURE InitApplication;
BEGIN
    NEW(proc1); proc1.i := 1;
```

```
    Objix.InitProcess(proc1, OCC.MaxSlice, Producer, OCC.ProcessStackSize);
    Objix.Kernel.InstallProcess(proc1, OCC.StateReady, 5);
    NEW(proc2); proc2.i := 2;
    Objix.InitProcess(proc2, OCC.MaxSlice, Consumer, OCC.ProcessStackSize);
    Objix.Kernel.InstallProcess(proc2, OCC.StateReady, 5);
    NEW(bufProtect); Objix.InitMutexD(bufProtect^);
    NEW(full); Objix.InitSemaphoreD(full^, 0);
    NEW(empty); Objix.InitSemaphoreD(empty^, 10);
    bufIn := 0; bufOut := 0;
END InitApplication;

PROCEDURE main+(): LONGINT;          ⇐ „+“ für C-kompatible Deklaration
    VAR scheduler: Objix.Scheduler;
BEGIN
    IF Objix.Init(3, 100) THEN       ⇐ maximal 3 Prozesse (und maximal 100 Elemente im
        intern verwendeten QueueElementSpace)
        NEW(scheduler);
        Objix.InitSchedulerD(scheduler^, maxPriority);
        Objix.Start(InitApplication, 1000, scheduler);   ⇐ 1000x pro Sekunde
            automatisch schedulen, die Prozedur InitApplication wird in dier
            Initialisierungsphase des Kernels automatisch aufgerufen.
        RETURN NoError
    ELSE
        Out.Ln; Out.String("Error when initializing the Objix modules");
        Objix.Exit;
        RETURN ErrorInit
    END
END main;

END ProducerConsumer.
```

Bewertung und Ausblick

Ziele

Objektorientiertheit, Erweiterbarkeit
Es ist keine Klassenhierarchie mit tiefen Ableitungsstufen entstanden, dafür aber gibt es viele Uses- oder Has-A-Beziehungen zwischen den Klassen.

Der Scheduler arbeitet mit beliebigen vom Typ *Process* abgeleiteten Prozessen. Das Anwendungsprogramm installiert üblicherweise einen neuen, vom Standard-Prozeß abgeleiteten Prozeß mit neuen Datenfeldern. Man hat somit innerhalb der typgebundenen Prozeduren des Prozesses Zugriff auf die prozeßlokalen Daten, auf dessen erweiterten Zustand.

Die Implementierung von Alarmen erfolgte als Unterklasse der Klasse Process in nur 85 Zeilen. Auch das ist ein Hinweis, daß sich Erweiterungen fast nahtlos in die bestehende Implementierung einfügen.

Der implementierte Scheduling-Mechanismus kann durch einfache objektorientierte Erweiterungen (durch Überschreiben verschiedener Zustandsüberführungen etc.) an die persönlichen Bedürfnisse angepaßt werden. Es sind prinzipiell schon einige Erweiterungen vorgesehen, einige Punkte, an denen man gezielt in das Scheduling eingreifen kann (z.B. BeforeProc und AfterProc in Interruptroutine, IdleProcessProc).

Echtzeitfähigkeit
Der unterbrechende, prioritätengesteuerte Scheduler teilt immer dem Prozeß mit der höchsten Priorität den Prozessor zu. Der Prozessor selber hat eine maximale Leistung von 66 MIPS (Millionen Instruktionen pro Sekunde). Zeitlich ist es kein Problem, jede Millisekunde einen Timer-Interrupt zu verarbeiten (samt dem dadurch initiierten Scheduling-Vorgang), aber je mehr Scheduling-Vorgänge man pro Zeiteinheit verlangt, desto langsamer wird das System. Getestet wurde der Mikrokernel mit bis zu 7800 Scheduling-Vorgängen pro Sekunde (entspricht einer Periodendauer von 128µs).

Lesbarkeit, Verständlichkeit
Hier muß der prinzipielle Vorteil von Oberon-2 gegenüber C++ angeführt werden (z.B. nur qualifizierter Import von Modulen, man weiß sofort, woher eine Funktion kommt, wo globale Daten deklariert sind). Auch der hypertext-ähnliche Editor des Oberon-Systems hilft beim Lesen des Quelltextes.

Die objektorientierte Implementierung und intuitive Namensgebung, sowie der Einbau von Assertionen machen das Verstehen des Kernels leichter.

Einfachheit
Einfachheit ist nicht immer möglich, durch Dokumentation der grundlegenden Gedankengänge aber meist erreichbar.

Portabilität
Portabilität ist im Fall eines Mikrokernels nur schwer zu erreichen, da zumindest der Prozeßwechsel prozessorspezifisch ist. Die Portabilität von einem Entwicklungsboard auf ein anderes wurde durch ein Modul, das die Hardwareabhängigkeiten kapselt, erreicht (OBoardSpecific.Mod, enthält Frequenz des Timerbausteins, externe und interne Timer-Interruptnummer, Basisadresse des Timerbausteins, Ansteuerung von LEDs etc). Nur etwa 250 Zeilen sind in Assembler geschrieben, davon sind sich wiederum die Hälfte der Zeilen sehr ähnlich, außerdem wurde, wo möglich, vom Compiler generierter Assemblercode verwendet. Alles Darüberliegende wurde in Oberon implementiert und sollte somit leicht portiert werden können.

Vorteile gegenüber bisherigen Entwicklungsunterstützungen

Die einfache und schnelle Entwicklung paralleler Programme ist sicher der Hauptvorteil, den die Verwendung des Mikrokernels bietet. Man muß sich nur an wenige Richtlinien halten, die generell bei der Programmierung paralleler Prozesse maßgebend sind (Schutz vor gleichzeitigem Zugriff auf Hardware-Ressourcen), erreicht damit aber mit wenig Aufwand einfache Problemlösungen.

Man kann das automatische Scheduling nach Prioritäten gezielt ausnützen, um Aktivitäten mit niedriger Priorität hintanzustellen, Aktivitäten mit hoher Priorität, die nur selten oder kurz aktiv werden, aber sofort den Prozessor zuteilen zu lassen. Man muß sich nur anfangs die Prioritäten der einzelnen Prozesse überlegen und kann dann auf das automatische Scheduling vertrauen.

Der Mikrokernel bietet einfache Mechanismen zur Synchronisation der parallel ablaufenden Prozesse.

Ausblick

Priority inheritance protocol
Wenn mehrere parallele Prozesse gleichzeitig arbeiten wollen, sollte immer der mit der höheren Priorität den Prozessor zugeteilt bekommen. Was passiert aber, wenn ein Prozeß hoher Priorität auf Hardware-Ressourcen zugreifen will, die zur Zeit von einem Prozeß mit niedriger Priorität blockiert werden? In diesem Fall müßte der Prozeß höherer Priorität solange warten, bis der Prozeß mit niedriger Priorität seine Arbeit beendet hat. Diese Situation nennt man *„priority inversion“*. Im schlimmsten Fall könnte der Prozeß mit der geringsten Priorität den Prozeß mit der nächst höheren Priorität blockieren, der wieder den nächsten usw., sodaß der Prozeß mit der niedrigsten Priorität den Prozeß mit der höchsten Priorität blockieren kann! Dieses Problem läßt sich mit dem „priority inheritance protocol“ [RAJ89, SRL87, TOK89,

SIL94] lösen, in dem all diese Prozesse, die Hardware-Ressourcen benutzen, die ein höherpriorer Prozeß benötigt, deren höhere Priorität erben, solange sie die Hardware-Ressource verwenden. Danach erhalten sie wieder ihre ursprüngliche Priorität.

Durch Ableiten einer neuen Semaphore von der Standard-Semaphore müßte die Implementierung des „priority inheritance protocols“ auf einfache Weise möglich sein.

Deadlock prevention, deadlock detection

„Deadlock prevention“ und „deadlock detection“ sind prinzipiell in Systemen mit parallelen Prozessen sehr angenehm, da sich der Programmierer nicht allzu viel Sorgen machen muß, ob das System eventuell blockiert. Im Fall von Echtzeitanwendungen ist aber abzuwägen, ob man dem Programmierer diese Last nicht doch aufbürden soll, da auf der anderen Seite die Deadlockerkennung oder -vermeidung bei jedem Sperren von Hardware-Ressourcen einen beträchtlichen Laufzeit-Overhead mit sich bringt.

Browser für den Crosscompiler

Normalerweise gibt es unter dem Oberon-System den "Browser", ein Programm zum automatischen Extrahieren der exportierten Schnittstelle von Modulen. Da der Oberon-2-Compiler von Richard Hable [HAB95] kleine Erweiterungen gegenüber einem Standard-Oberon-Compiler enthält, wäre ein angepaßter Browser notwendig.

Oberon-to-C Translator

Josef Templ hat einen Oberon-2-to-C-Translator implementiert [TEM95]. Dieser Translator übersetzt vollautomatisch Oberon-2 Programme in C-Programme. Diese C-Programme können dann mit einem (z.B. optimierenden) C-Compiler auf der Zielplattform übersetzt und gelinkt werden. Der Translator erzeugt aus jedem Oberon-Modul ein C- und ein H-File, zusätzlich die Dateien SYSTEM.C und SYSTEM.H, die das Laufzeitsystem mit Garbage Collector enthalten. Die objektorientierten Konzepte von Oberon-2 werden auf natürliche Weise in C nachgebildet (Typdeskriptoren, virtuelle Methoden-Tabellen, Typerweiterungen, Laufzeittypinformation etc.), der entstehende C-Quellcode ist weiterhin sehr gut lesbar und editierbar. Im Laufe der Übersetzung mußten im Mikrokernel die Stellen angepaßt werden, die nicht dem Standard-Oberon entsprechen (z.B. die Compiler-Erweiterungen des Oberon-2-Crosscompilers). Weitere Anpassungen waren im Laufzeitsystem notwendig, da man dort in gewissen Fällen das automatische Scheduling ausschalten muß (z.B. beim Anlegen von Speicherplatz). Wie sich der Garbage Collector im Betrieb verhält, wurde noch nicht getestet. Da der zu verwaltende Speicher allerdings nur 1 MB groß ist, müßte ein Einsammeln des nicht mehr benötigten Speichers in kurzer Zeit erledigt sein. Eine noch zu implementierende Erweiterung auf Seiten des Translators ist das Berücksichtigen von mehreren Stacks beim Garbage Collector Durchlauf, da auch auf den Stacks der nicht aktiven Prozesse Zeiger liegen können, die auf sonst nicht mehr referenzierte Objekte zeigen.

User Manual

Allgemeines

Es gibt zu jedem Typ im System zwei Prozeduren, deren Namen sich aus dem Typ ableiten: eine Initialisierungsprozedur und eine Finalisierungsprozedur. In der Initialisierungsprozedur (entspricht dem Konstruktor von C++) werden die Datenfelder des jeweiligen Typs initialisiert und eventuell dynamischer Speicher angelegt. In der Finalisierungsprozedur wird eventuell angelegter dynamischer Speicher wieder freigegeben. Eine Variable vom Typ X muß mit InitX(x) initialisiert werden und mit DeleteX(x) finalisiert werden.

In den meisten Fällen wird man mit Pointern auf Records arbeiten. Für diese Records muß man zuerst Speicher allozieren und diesen danach initialisieren. Beispiel:

```
TYPE
   T = POINTER TO TDesc;
   TDesc = RECORD i: INTEGER END;

VAR
   t: T;
   tdesc: TDesc;

BEGIN
   NEW(t); InitTD(t^);
   InitTD(tdesc);
END
```

Die Initialisierungsprozeduren sind in den meisten Fällen so definiert, daß ihr erster Parameter ein VAR-Record-Parameter ist. Deshalb muß man einen Zeiger auf ein Record bei der Übergabe als Parameter dereferenzieren. Record-Variablen können direkt übergeben werden.

Die meisten typgebundenen Prozeduren haben als Receiver einen VAR-Record. Lediglich die an den Typ ProcessDesc gebundenen Prozeduren haben als Receiver einen Zeiger auf ProcessDesc. Der Grund dafür ist, daß man einen Zeiger auf ein Record durch Dereferenzierung einfach in ein Record umwandeln kann. Somit ist man bei typgebundenen Prozeduren, die als Receiver einen VAR-Record haben, für beide Fälle gerüstet (Prozedur an Zeiger zu binden, Prozedur an VAR-Record zu binden). Von einem Record kommt man nicht wieder auf einen Zeiger auf dieses Record. SYSTEM.ADR (angewendet auf ein Record) liefert die Adresse des ersten Elements des Records. Man kann diese Adresse allerdings nicht mit SYSTEM.VAL (die Standardprozedur für Typumwandlungen) auf einen Zeiger auf das Record umwandeln. Da im Kernel intern oftmals Zeiger auf einen Prozeß benötigt werden (currProcess, oldProcess etc.) und auch Zeiger auf ProcessDesc miteinander verglichen werden müssen (z.B. um zu sehen, ob der Receiver einer typgebundenen Prozedur der currProcess ist),

ist es notwendig, daß der Receiver von an ProcessDesc gebundenen Prozeduren eine Zeigervariable ist.

Um unvorhersehbar lange dauernde Unterbrechungen zu vermeiden, die durch die Verwendung von dynamischem Speicher entstehen können, gibt es im Kernel sogenannte Spaces, das sind Arrays, die Elemente eines bestimmten Typs enthalten. Aus diesen Arrays können Elemente mit Allocate angefordert werden und mit Deallocate wieder zurückgegeben werden. Für die Dauer dieser Prozeduraufrufe lassen sich obere Schranken angeben, da sie nur Arrays einer bestimmten Größe traversieren müssen.

Im Sinne der Erweiterbarkeit werden viele Datenfelder und Prozeduren aus dem Kernelmodul Objix exportiert. Viele dieser Datenfelder und Prozeduren waren ursprünglich nicht öffentlich gedacht. Man handelt sich mit freizügigem Exportieren einige Probleme ein, die man eigentlich durch objektorientierte Programmierung zu vermeiden versuchte, z.B. den Zugriff auf private Datenfelder nicht nur durch die exportierten Zugriffsprozeduren. Deshalb sind die meisten, ursprünglich als privat gedachten Datenfelder nur read-only exportiert, d.h. sie können von außen nicht gesetzt werden. Es gibt somit eigentlich zwei Schnittstellen: einmal die ursprünglich gedachte, enge Schnittstelle, die höchstmögliche Systemintegrität garantiert (der Zugriff auf kernelinterne Datenstrukturen erfolgt nur über geeignete Zugriffsprozeduren) und die (alles umfassende) weite Schnittstelle, die es dem Anwender erlaubt, in die ursprüngliche Kernelfunktionalität einzugreifen und diese zu erweitern. Bei den unten angeführten Typen wird jeweils darauf hingewiesen, welche Datenfelder zur intendierten Programmierschnittstelle und welche zur erweiterten Programmierschnittstelle gehören.

Kernel allgemein

Die Initialisierung des Kernels erfolgt durch die Funktionsprozedur Init, dabei gibt man die maximal zu unterstützende Anzahl von Prozessen an. Die internen Datenstrukturen des Kernels werden dann für diese Maximalanzahl ausgelegt.

Nachdem der Kernel initialisiert ist, kann er mit der Prozedur Start gestartet werden. Der erste Parameter, initCode, ist eine Prozedur, in der typischerweise die Prozesse und Systemressourcen des Anwendungsprogrammes angelegt und initialisiert werden. Der zweite Parameter, ticksPerSecond, gibt an, wie viele Schedulingvorgänge pro Sekunde abgearbeitet werden sollen. Der dritte Parameter, scheduler, ist der Scheduler, der anschließend das Scheduling durchführt. Der Scheduler muß vor dem Aufruf von Init angelegt und initialisiert werden. Normalerweise sollte der Benutzer mit den Statements „NEW(scheduler); Objix.InitScheduler(scheduler, maxProcessPriority)“ den Standardscheduler anlegen und initialisieren. Will man aber ein spezielleres Scheduling-Verhalten, kann man durch Ableiten eines neuen Schedulers vom Standardscheduler dessen Verhalten erweitern und anschließend den erweiterten Scheduler als Parameter an die Initialisierungsprozedur Init übergeben.

Die Finalisierung des Kernels erfolgt durch die Prozedur Exit.

Zu Testzwecken kann man die Prozedur *PrintQueues* aufrufen. *PrintQueues* gibt die zum Scheduling notwendigen FIFOQueues über die Standardausgabe aus. Zur Steuerung, was und wieviel ausgegeben werden soll, gibt es die Variable *outOptions*. Wenn die SET-Variable *outOptions* das Bit *Print* enthält, erfolgt eine Ausgabe, wenn das Bit *PrintNotWhenCritical* gesetzt ist, erfolgt eine Ausgabe nur, wenn der Kernel nicht durch *Kernel.Lock* gesperrt wurde. Je nachdem, welche weiteren Bits gesetzt sind, werden die FIFOQueues für die Prozeßzustände *ready* (PrintRQ), *asleep* (PrintAQ), *suspended* (PrintSQ) und *asleepsuspended* (PrintASQ) ausgegeben. Die Prozedur *OutLine* gibt den ihr übergegenen String über die Standardausgabe aus und ruft anschließend *PrintQueues* auf. Die Steuerung der Ausgabe erfolgt auch hier über die Variable *outOptions*.

Eine zentrale Datenstruktur im Kernel ist die *FIFOQueue*. Sie verwaltet Daten in „First in first out"-Reihenfolge, d.h. Daten, die als erste in die Queue gelangen, verlassen diese auch wieder als erste. Eine *FIFOQueue* ist in dieser Implementierung von einer normalen *Queue* abgeleitet und erbt von dieser alle Datenfelder und Prozeduren. Somit hat jede FIFOQueue einen Zeiger auf das erste Element („head"), einen Zeiger auf das letzte Element („tail") und die Anzahl der Elemente („count"). Die einzelnen Elemente der Queue sind vom Typ *QueueElement* und über das Feld „next" miteinander verkettet. Die Daten, die eine Queue verwalten kann, müssen (mindestens) vom Typ *QueueData* sein. Über Typerweiterung kann man beliebige, von *QueueData* abgeleitete Daten in eine Queue einfügen. In von *QueueData* abgeleiteten Typen kann die Prozedur *GetPriority* überschrieben werden. Die Prozedur zum Einfügen von Daten ist *Enqueue*. *Dequeue* liefert die Daten desjenigen QueueElements, das als erstes eingehängt wurde. Mit *Remove* kann man die spezifizierten Daten (befinden sich an beliebiger Stelle innerhalb der Queue) entfernen. Die Funktionsprozedur *Empty* liefert TRUE, wenn die Queue leer ist. Die Prozedur *Zap* leer die Queue. Die Prozeduren *First* und *Next* kann man zum Iterieren über die Queue verwenden. Dazu verwendet man typischerweise folgenden Programmcode:

```
qelem := q.First();
WHILE qelem # NIL DO
   (* ... *)
   qelem := q.Next(qelem)
END
```

Die hier vorgestellte Datenstruktur entspricht also nicht mehr einer *FIFOQueue* im engeren Sinn, da man Daten an beliebiger Stelle entnehmen kann und über den Inhalt der Queue iterieren kann. Diese Funktionalität ist allerdings im Scheduler notwendig. Weiters möchte ich darauf hinweisen, daß der Anwender diese *FIFOQueue* keineswegs selbst verwenden muß, sie wird lediglich in ihrer Implementierung zur Verfügung gestellt (was Mehrfachimplementierungen derselben Datenstrukturen vermeiden helfen sollte). Variablen vom Typ *QueueElement* können aus einem *QueueElementSpace* mit *Allocate* angefordert

werden und sollten dann mit *Deallocate* wieder freigegeben werden. Der Kernel stellt einen *QueueElementSpace* mit der Variablen *space* zur Verfügung.

Ein Prozeß ist zu jeder Zeit in einem Array von Prozessen, der *ProcessList* des Schedulers, enthalten. Dieses enthält alle Prozesse. Je nachdem, in welchem Zustand ein Prozeß ist, ist er zusätzlich in anderen Queues des Schedulers eingequeuet. Dazu wird in die entsprechende *FIFOQueue* ein *QueueElement* eingehängt.

Programmierschnittstelle

```
CONST
   Print = 0;
   PrintAQ = 2; PrintASQ = 4; PrintNotWhenCritical = 5;
   PrintRQ = 1; PrintSQ = 3;

VAR
   Kernel-: Scheduler;
   currProcess, idleProcess-: Process;
   inISRcount-: LONGINT;
   lockvar-, numberOfProcesses-: INTEGER;
   oldProcess: Process;
   outOptions: SET;
   prcbPtr-: OCT.ProcessControlBlock;
   realTime-: LONGINT;
   space-: QueueElementSpace;

TYPE
   Queue = POINTER TO QueueDesc;
   QueueDesc = RECORD
      n-: LONGINT;
      head-: QueueElement;
   END ;

   PROCEDURE InitQueueD (VAR q: QueueDesc);
   PROCEDURE DeleteQueueD (VAR q: QueueDesc);


   FIFOQueue = POINTER TO FIFOQueueDesc;
   FIFOQueueDesc = RECORD (QueueDesc)
      tail-: QueueElement;
      PROCEDURE (VAR q: FIFOQueueDesc) Dequeue (): QueueData;
```

```
    PROCEDURE (VAR q: FIFOQueueDesc) Empty (): BOOLEAN;
    PROCEDURE (VAR q: FIFOQueueDesc) Enqueue (d: QueueData);
    PROCEDURE (VAR q: FIFOQueueDesc) First (): QueueElement;
    PROCEDURE (VAR q: FIFOQueueDesc) Next (this: QueueElement): QueueElement;
    PROCEDURE (VAR q: FIFOQueueDesc) Remove (d: QueueData);
    PROCEDURE (VAR q: FIFOQueueDesc) Zap;
END ;

PROCEDURE InitFIFOQueueD (VAR q: FIFOQueueDesc);
PROCEDURE DeleteFIFOQueueD (VAR q: FIFOQueueDesc);


QueueElement = POINTER TO QueueElementDesc;
QueueElementDesc = RECORD
    data-: QueueData;
    next-: QueueElement;
END ;

PROCEDURE DeleteQueueElementD (VAR qelem: QueueElementDesc);
PROCEDURE InitQueueElementD (VAR qelem: QueueElementDesc);


QueueElementSpace = POINTER TO QueueElementSpaceDesc;
QueueElementSpaceDesc = RECORD
    elems-: POINTER TO ARRAY OF QueueElement;
    inUse-: POINTER TO ARRAY OF BOOLEAN;
    PROCEDURE (VAR qsp: QueueElementSpaceDesc) Allocate (): QueueElement;
    PROCEDURE (VAR qsp: QueueElementSpaceDesc) Deallocate (q: QueueElement):
        INTEGER;
END ;

PROCEDURE InitQueueElementSpaceD (VAR qsp: QueueElementSpaceDesc; maxElems:
    LONGINT): BOOLEAN;
PROCEDURE DeleteQueueElemSpaceD (VAR qsp: QueueElementSpaceDesc);


QueueData = POINTER TO QueueDataDesc;
QueueDataDesc = RECORD
    PROCEDURE (VAR qdata: QueueDataDesc) GetPriority (): OCT.Priority;
END ;
```

```
PROCEDURE InitQueueDataD (VAR qdata: QueueDataDesc);
PROCEDURE DeleteQueueDataD (VAR qdata: QueueDataDesc);


PROCEDURE IdleProcess;
PROCEDURE Init (nrOfProcesses: INTEGER; maxElemsQES: LONGINT): BOOLEAN;
PROCEDURE Start (code: OCT.Code; ticksPerSecond: INTEGER; scheduler: Scheduler);
PROCEDURE Exit;
PROCEDURE PrintQueues;
PROCEDURE OutLine (s: ARRAY OF CHAR);
```

Process

Ein Prozeß ist konzeptuell ein Stück Code, das unabhängig von anderen ausgeführt wird.

Ein Prozeß enthält einen Zeiger auf einen virtuellen Prozessor („vp“). Ein virtueller Prozessor enthält den Stack, der zur Abarbeitung des Prozesses verwendet wird, und einen Satz von Registern, der die Registerinhalte der Register des i960 zwischenspeichert, wenn der Prozeß unterbrochen ist. Das Feld „code“ ist eine Prozedurvariable und enthält den Code, der als Prozeß ausgeführt werden soll. Die Felder „prio“, „state“, „suspendCount“, „schedTicks“, „sleepTicks“ und „maxSlice“ werden vom Scheduler verwendet, um dem Prozeß den Prozessor zuzuteilen. Die Felder „qsid“ und „slotID“ sind notwendig für das Abspeichern eines Prozesses in FIFOQueues. Schließlich gibt es noch eine Liste der zum Prozeß gehörigen Alarme. Auf keines der Datenfelder sollte vom Anwendungsprogramm aus direkt zugegriffen werden.

Über die typgebundenen Prozeduren *GetPriority* und *SetPriority* ändert man die Priorität eines Prozesses. Je höher (numerisch höher) die Priorität ist, umso wahrscheinlicher ist es, daß der Prozeß den Prozessor zugeteilt bekommt.

Die Prozeduren *Suspend*, *Resume* und *Sleep* ändern den Zustand des Prozesses und werden im Rahmen des Schedulings näher erklärt.

Mit der Prozedur *CreateAlarm* kann man einen zum Prozeß gehörigen Alarm erstellen und mit *DestroyAlarm* wieder zerstören.

Objektdiagramm

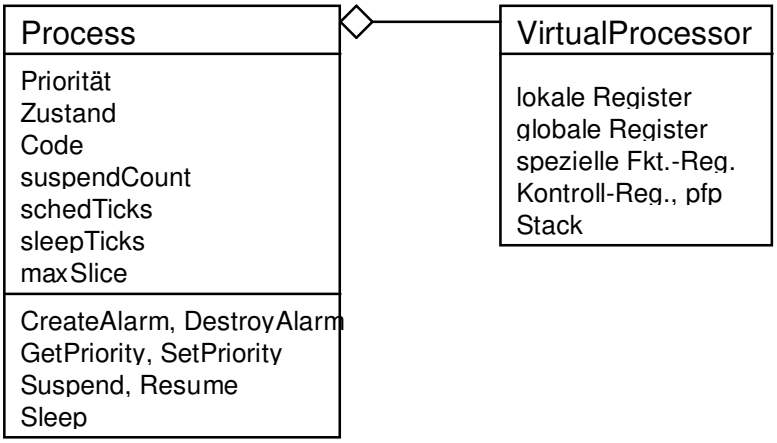

Abbildung 30, Objektdiagramm Process und VirtualProcessor

Programmierschnittstelle

```
VirtualProcessor = POINTER TO VirtualProcessorDesc;
VirtualProcessorDesc = RECORD
    regs-: OCT.Registers;
    st-: ARRAY OCC.ProcessStackSize DIV SIZE(LONGINT) OF LONGINT;
END ;

PROCEDURE InitVirtualProcessorD (VAR this: VirtualProcessorDesc; stackSize:
    LONGINT);
PROCEDURE DeleteVirtualProcessorD (VAR this: VirtualProcessorDesc);


Process = POINTER TO ProcessDesc;
ProcessDesc = RECORD
    vp-: VirtualProcessor;
    prio-: OCT.Priority;
    state-: OCT.State;
    suspendCount-: SHORTINT;
    schedTicks-, sleepTicks-, maxSlice-: OCT.Ticks;
    alarms-: FIFOQueue;
    qsid-: QueuedSlotID;
    slotID-: INTEGER;
    code-: OCT.Code;
```

```
    PROCEDURE (thisProcess: Process) CreateAlarm (kind: SHORTINT; delay, maxSlice,
        cycleTicks: OCT.Ticks; priority: OCT.Priority; code: OCT.Code):
        OCT.AlarmHandle;
    PROCEDURE (thisProcess: Process) DestroyAlarm (alarmHandle: OCT.AlarmHandle);
    PROCEDURE (this: Process) GetPriority (): OCT.Priority;
    PROCEDURE (thisProcess: Process) Resume (proc: Process);
    PROCEDURE (thisProcess: Process) SetPriority (prio: OCT.Priority);
    PROCEDURE (thisProcess: Process) Sleep (ticks: OCT.Ticks);
    PROCEDURE (thisProcess: Process) Suspend (proc: Process);
END ;

PROCEDURE InitProcess (this: Process; maxSlice: OCT.Ticks; code: OCT.Code;
    stackSize: LONGINT);
PROCEDURE DeleteProcess (this: Process);
```

Detaillierte Beschreibung der Programmierschnittstelle

CreateAlarm (kind, delay, maxSlice, cycleTicks, priority, code) erstellt einen Alarm und liefert ein Handle auf diesen Alarm zurück. Dieses Handle kann verwendet werden, um den Alarm mit *DestroyAlarm* wieder zu entfernen. Wenn der Alarm nicht mit *DestroyAlarm* gelöscht wird, wird er spätestens gelöscht, wenn sein Vaterprozeß gekillt wird. Der Alarm ist entweder relativ (kind = OCommonConstants.AlarmRelative) oder zyklisch (kind = OCommonConstants.AlarmCyclic). “delay” gibt die Zeitspanne an, die verstreichen soll, bis der Alarm „ready“ wird (muß größer als 0 sein). Danach wird ein relativer Alarm einmal ausgeführt, zyklische Alarme werden danach mit einer Periode von “cycleTicks” immer wieder ausgeführt. “maxSlice” ist die Zeitscheibe für den Alarm. Nach “maxSlice” Ticks versucht der Scheduler, einem anderen “ready”-Prozeß den Prozessor zuzuweisen (falls es andere “ready”-Prozesse gibt). “priority” ist die Priorität, mit der der Prozeß laufen soll. Da man meistens will, daß Alarme auch tatsächlich zum spezifizierten Zeitpunkt ausgeführt werden, ist es meistens anzuraten, Alarmen eine hohe Priorität zu geben (da sie sonst durch höherpriore Anwendungsprozesse blockiert werden). “code” ist eine parameterlose Prozedur, die als Alarm ausgeführt werden soll.

GetPriority liefert die aktuelle Priorität des Prozesses, die man mit SetPriority verändern kann. Die Priorität eines Prozesses darf höchstens gleich der Priorität sein, die man als maximale beim Initialisieren des Schedulers angegeben hat. Weiters muß man beachten, daß beim Aufruf der Prozedur SetPriority möglicherweise die Kontrolle abgegeben wird. Wenn man nämlich seine eigene Priorität erniedrigt und man somit nicht mehr der höchstpriore Prozeß ist, schaltet der Scheduler zum nunmehr höchstprioren Prozeß um.

Suspend führt eine Suspend-Operation auf dem übergebenen Prozeß aus. Man kann sich auch selbst unterbrechen (durch proc.Suspend(proc)). Resume führt eine Resume-Operation auf

dem übergebenen Prozeß aus, wodurch es wieder zur Abgabe der Kontrolle kommen kann (wenn ein höherpriorer Prozeß wiederaufgenommen wurde).

Sleep führt eine Sleep-Operation auf den Prozeß selbst aus. “ticks” ist die Anzahl der Ticks, die der Prozeß schlafen soll. Danach wacht er automatisch wieder auf.

Scheduler

Der Scheduler teilt den parallel ablaufenden Prozessen abwechselnd den Prozessor zu. Bei der Initialisierung des Schedulers gibt man die maximale Priorität eines Prozesses an. Der Scheduler dimensioniert die internen Datenstrukturen so, daß er Prozesse mit der maximalen Priorität verarbeiten kann. Die Prozedur *InstallScheduler* installiert den Scheduler. Fortan werden pro Sekunde “ticksPerSecond” Schedulingvorgänge ausgeführt. Normalerweise wird die Prozedur *InstallScheduler* automatisch von der Prozedur *Start* aufgerufen. Analog dazu wird die Prozedur *DeinstallScheduler* automatisch von der Prozedur Exit aufgerufen.

Ein Scheduler enthält einen Zeiger auf eine *ProcessList*. Jeder Prozeß, der mit *InstallProcess* dem Scheduler bekanntgegeben wird, wird in dieses Array eingefügt, wo er bleibt, bis er mit *DeinstallProcess* daraus entfernt wird. Je nachdem, in welchem Zustand ein Prozeß ist, befindet er sich zusätzlich in einer der FIFOQueues. Ein Prozeß der Priorität x, der im Zustand y ist, befindet sich in der x. FIFOQueue des Zustands y (z.B. ReadyQueues[x]).

Das automatische Scheduling kann man mit den Prozeduren *StopScheduling* und *StartScheduling* (inkrementieren bzw. dekrementieren die Elementvariable „stopSchedulingCount“) bzw. *Lock* und *Unlock* (inkrementieren bzw. dekrementieren die modulglobale Variable „lockvar“) gesteuert werden. Scheduling erfolgt nämlich nur, wenn stopSchedulingCount = 0 und lockvar = 0 sind. Die beiden Varianten haben allerdings eine unterschiedliche Semantik: *Lock* und *Unlock* sperren die Interrupts vom Timerbaustein. Dieser periodische Interrupt bewirkt normalerweise den automatischen Prozeßwechsel. In der vom Interruptcontroller aufgerufenen Timer-ISR wird unter anderem die Echtzeit (modulglobale Variable “realTime”) fortgezählt, was natürlich dann auch unterbleibt. Die Variable “stopSchedulingCount” wird in der Timer-ISR auf Null geprüft. Ist die Variable größer als Null, wird kein Prozeßwechsel vorgenommen, die Echtzeit wird aber weiter fortgezählt.

Man sollte vom Anwendungsprogramm aus nur in speziellen Fällen und nur für kurze Zeit das automatische Scheduling unterbinden. Wenn es sein muß, sind die Prozeduren *StopScheduling* und *StartScheduling* vorzuziehen. Man muß weiters darauf achten, daß die entsprechenden Prozeduren immer paarweise ausgeführt werden.

Die Funktionsprozedur *GetRealTime* liefert die bisher abgelaufene Echtzeit (modulglobale Variable “realTime”).

Die Prozeduren *Kill*, *Suspend*, *Resume* und *Sleep* ändern den Zustand des angegebenen Prozesses.

Die Prozeduren *Reschedule*, *DoWakeups*, *Preempt*, *DispatchNext* und *NextProcess* sollten nicht verwendet werden, können jedoch bei einem erweiterten Scheduler überschrieben werden. Die Prozedur *ImmediateSwitchUnlock* sollte nur in besonderen Fällen vom Anwendungsprogramm aus aufgerufen werden, da sie einen sofortigen Prozeßwechsel vom Prozeß *oldProcess* zum Prozeß *currProcess* (beides modulglobale Variablen des Moduls Objix) bewirkt. Die Prozedur *KillAlarms* wird nur in *Kill* und *DeleteProcess* aufgerufen.

Die Elementvariable “es” stellt einen *QPSpace*, also einen QueuedProcessSpace, bereit, von dem QueuedSlotIDs mit *Allocate* angefordert werden können (müssen mit Deallocate wieder freigegeben werden). Es ist nicht notwendig, auf die Datenstrukturen *Slot*, *QueuedSlotID*, *ProcessList* und *QPSpace* vom Anwendungsprogramm aus direkt zuzugreifen (auch nicht über die Zugriffsprozeduren). Sie dienen der internen Implementierung

Die Prozedur *timerisr* ist die Oberonprozedur, die von der Assembler-Interrupt-Service-Routine aufgerufen wird und automatischen Prozeßwechsel implementiert. Die Prozedur *timerisr* selbst kann nicht überschrieben werden, sie wird automatisch von der Assembler-ISR aufgerufen. Man kann allerdings die Prozedurvariablen *BeforeProc* und *AfterProc* des Schedulers auf beliebige parameterlose Prozeduren setzen. Diese werden dann am Anfang bzw. am Ende der *timerisr* aufgerufen. Dadurch ist es möglich, auch in das automatische Scheduling im nachhinein noch einzugreifen. Dabei muß man allerdings beachten, daß diese Prozeduren laufzeiteffizient sind (sie werden schließlich „ticksPerSecond“ mal pro Sekunde aufgerufen).

Die Elementvariable *IdleProcessProc* kann auf eine beliebige parameterlose Prozedur des Anwendungsprogramms gesetzt werden. Sie wird vom *IdleProcess* aus periodisch aufgerufen. Man hat dadurch die Möglichkeit, periodische niederpriore Aktivitäten ausführen zu lassen.

Objektdiagramm

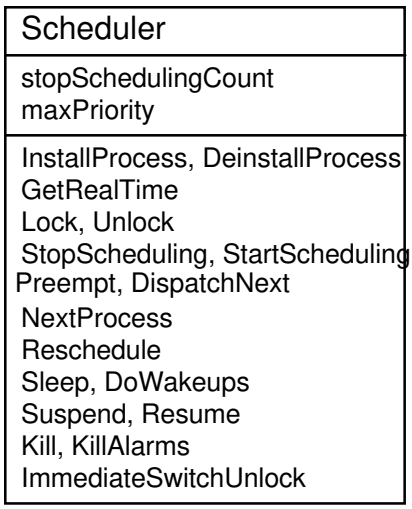

Abbildung 31, Objektdiagramm Scheduler

Programmierschnittstelle

```
Scheduler = POINTER TO SchedulerDesc;
SchedulerDesc = RECORD
    stopSchedulingCount-: SHORTINT;
    es-: QPSpace;
    pl-: ProcessList;
    maxPriority-: OCT.Priority;
    ReadyQueues-: POINTER TO ARRAY OF FIFOQueue;
    AsleepQueues-: POINTER TO ARRAY OF FIFOQueue;
    SuspendedQueues-: POINTER TO ARRAY OF FIFOQueue;
    AsleepSuspendedQueues-: POINTER TO ARRAY OF FIFOQueue;
    oldSchedulerISR-, BeforeProc, AfterProc, IdleProcessProc: OCT.Code;
    PROCEDURE (VAR this: SchedulerDesc) DeinstallProcess (proc: Process);
    PROCEDURE (VAR this: SchedulerDesc) DispatchNext;
    PROCEDURE (VAR this: SchedulerDesc) DoWakeups (VAR rescheduleNecessary:
        BOOLEAN);
    PROCEDURE (VAR this: SchedulerDesc) GetRealTime (): LONGINT;
    PROCEDURE (VAR this: SchedulerDesc) ImmediateSwitchUnlock;
    PROCEDURE (VAR this: SchedulerDesc) InstallProcess (proc: Process; state: OCT.State;
        prio: OCT.Priority);
    PROCEDURE (VAR this: SchedulerDesc) Kill (p: Process);
    PROCEDURE (VAR this: SchedulerDesc) KillAlarms (p: Process);
    PROCEDURE (VAR this: SchedulerDesc) Lock;
    PROCEDURE (VAR this: SchedulerDesc) NextProcess (): QueuedSlotID;
    PROCEDURE (VAR this: SchedulerDesc) Preempt;
    PROCEDURE (VAR this: SchedulerDesc) Reschedule;
    PROCEDURE (VAR this: SchedulerDesc) Resume (p: Process);
    PROCEDURE (VAR this: SchedulerDesc) Sleep (p: Process; ticks: OCT.Ticks);
    PROCEDURE (VAR this: SchedulerDesc) StartScheduling;
    PROCEDURE (VAR this: SchedulerDesc) StopScheduling;
    PROCEDURE (VAR this: SchedulerDesc) Suspend (p: Process);
    PROCEDURE (VAR this: SchedulerDesc) Unlock;
END ;

    PROCEDURE InitSchedulerD (VAR this: SchedulerDesc; maxPriority: OCT.MaxPriority);
    PROCEDURE InstallScheduler (scheduler: Scheduler; ticksPerSecond: INTEGER);
    PROCEDURE timerisr;
    PROCEDURE DeinstallScheduler (scheduler: Scheduler);
    PROCEDURE DeleteSchedulerD (VAR this: SchedulerDesc);
```

Detaillierte Beschreibung der Programmierschnittstelle

InstallProcess installiert den Prozeß “proc”. Der Zustand des Prozesses kann „ready“ (OCommonConstants.StateReady) oder „suspended“ (OCommonConstants.StateSuspended) sein. Die Priorität des Prozesses muß im Bereich [0, maxPriority] liegen („maxPriority“ ist die Priorität, die man beim Aufruf von *InitScheduler* angegeben hat)

Semaphore

Semaphore wurden erstmals 1965 von E. W. Dijkstra vorgeschlagen. Bei der Initialisierung einer Semaphore gibt man einen Anfangswert an, mit dem der mit der Semaphore assoziierte Zähler initialisiert wird. Auf die Semaphore können dann dementsprechend viele *Waits* ausgeführt werden, bevor ein *Wait* blockiert.

Die Prozedur *Wait* prüft, ob der mit der Semaphore assoziierte Zähler größer als 0 ist. Wenn er größer als 0 ist, wird der Zähler dekrementiert und fährt normal fort. Wenn der Wert 0 ist, wird der Prozeß in den Zustand suspended übergeführt. Das Prüfen des Zählerwertes, das Inkrementieren und das Ändern des Prozeßzustandes passieren in einem Schritt, nichtunterbrechbar. Es ist garantiert, daß kein anderer Prozeß auf die Semaphore zugreifen kann, sobald ein *Wait* oder *Signal* initiiert wurde.

Die Prozedur *Signal* inkrementiert den mit der Semaphore assoziierten Zähler. Wenn einer oder mehrere Prozesse durch ein *Wait* unterbrochen wurden, wird derjenige Prozeß, der als erster unterbrochen wurde, wiederaufgenommen. Solange zur Zeit der Initiierung eines *Signals* Prozesse (die durch ein Wait unterbrochen wurden) auf die Wiederaufnahme warten, wird nach der Abarbeitung des *Signal* der mit der Semaphore assoziierte Zähler weiterhin 0 sein, allerdings ist ein Prozeß weniger unterbrochen. Auch das Inkrementieren des Zählers und Wiederaufnehmen eines Prozesses ist nichtunterbrechbar.

Objektdiagramm

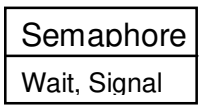

Abbildung 32, Objektdiagramm Semaphore

Programmierschnittstelle

```
Semaphore = POINTER TO SemaphoreDesc;
SemaphoreDesc = RECORD
   PROCEDURE (VAR this: SemaphoreDesc) Signal;
   PROCEDURE (VAR this: SemaphoreDesc) Wait;
END ;

PROCEDURE InitSemaphoreD (VAR this: SemaphoreDesc; initCount: INTEGER);
PROCEDURE DeleteSemaphoreD (VAR this: SemaphoreDesc);
```

Mutual Exclusion

Eine Mutual Exclusion Variable ist eine mit dem Wert 1 initialisierte Semaphore und kann dadurch dazu verwendet werden sicherzustellen, daß sich höchstens ein Prozeß in einem kritischen Bereich („critical region“) befindet. Critical regions sind zum Beispiel gemeinsam (von mehreren Prozessen) verwendete Systemressourcen, in den meisten Fällen gemeinsam verwendete Variablen. „Race conditions“ sind Situationen, in denen zwei oder mehrere Prozesse lesend oder schreibend auf gemeinsame Daten zugreifen und das Ergebnis vom genauen zeitlichen Ablauf der Prozesse abhängt. Mutual Exclusion Variablen kann man einsetzen, um „race conditions“ zu vermeiden, indem man gleichzeitigen Zugriff auf gemeinsame Daten unterbindet und eine bestimmte Abarbeitungsreihenfolge erzwingt.

Da Mutual Exclusion Variablen von Semaphore abgeleitet sind, haben sie auch die Zugriffsprozeduren Wait und Signal geerbt.

Objektdiagramm

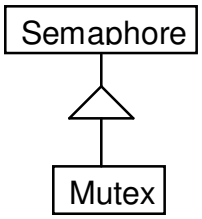

Abbildung 33, Objekdiagramm Mutual Exclusion

Programmierschnittstelle

```
Mutex = POINTER TO MutexDesc;
MutexDesc = RECORD (SemaphoreDesc) END ;

PROCEDURE InitMutexD (VAR this: MutexDesc);
PROCEDURE DeleteMutexD (VAR this: MutexDesc);
```

Alarme

Alarme sind Codeteile, die entweder einmal oder zyklisch ausgeführt werden. Alarme werden normalerweise mit *CreateAlarm* angelegt. Die Prozeduren *InitAlarm* und *DeleteAlarm* werden automatisch aufgerufen und sollten nicht vom Anwendungsprogramm aus aufgerufen werden.

Objektdiagramm

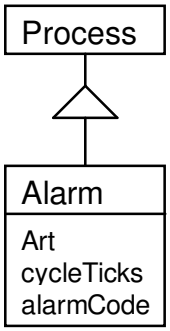

Abbildung 34, Objektdiagramm Alarm

Programmierschnittstelle

```
Alarm = POINTER TO AlarmDesc;
AlarmDesc = RECORD (ProcessDesc)
   kind-: SHORTINT;
   cycleTicks-: OCT.Ticks;
   parentProcess-: Process;
   alarmCode-: OCT.Code;
END ;

PROCEDURE InitAlarm (this: Alarm; maxSlice, cycleTicks: OCT.Ticks; code:
    OCT.Code);
PROCEDURE DeleteAlarm (this: Alarm);
PROCEDURE (thisProcess: Process) CreateAlarm (kind: SHORTINT; delay, maxSlice,
    cycleTicks: OCT.Ticks; priority: OCT.Priority; code: OCT.Code): OCT.AlarmHandle;
PROCEDURE (thisProcess: Process) DestroyAlarm (alarmHandle: OCT.AlarmHandle);
```

Gesamt-Objektdiagramm

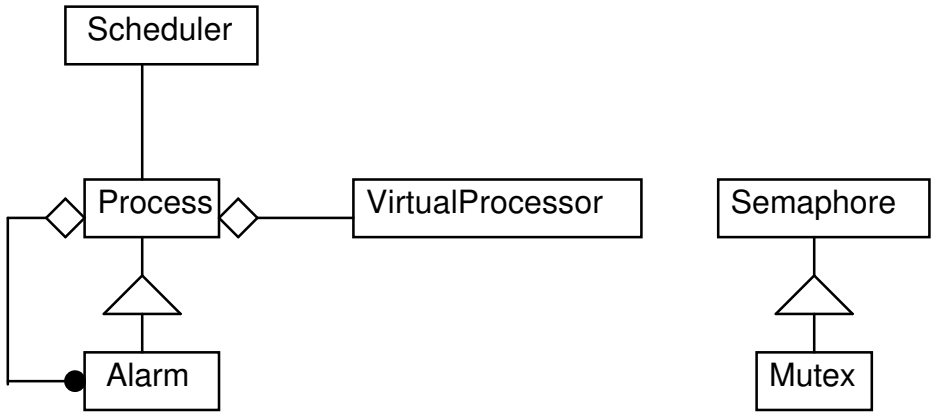

Abbildung 35, Gesamt-Objektdiagramm

Komplette Programmierschnittstelle des Mikrokernels

DEFINITION Objix;

```
IMPORT OCT := OCommonTypes;

CONST
   Print = 0;
   PrintAQ = 2;
   PrintASQ = 4;
   PrintNotWhenCritical = 5;
   PrintRQ = 1;
   PrintSQ = 3;

TYPE
   Alarm = POINTER TO AlarmDesc;
   AlarmDesc = RECORD (ProcessDesc)
      kind-: SHORTINT;
      cycleTicks-: OCT.Ticks;
      parentProcess-: Process;
      alarmCode-: OCT.Code;
   END ;

   VirtualProcessor = POINTER TO VirtualProcessorDesc;
   VirtualProcessorDesc = RECORD
      regs-: OCT.Registers;
      st-: ARRAY OCC.ProcessStackSize DIV SIZE(LONGINT) OF LONGINT;
   END ;

   FIFOQueue = POINTER TO FIFOQueueDesc;
   FIFOQueueDesc = RECORD (QueueDesc)
      tail-: QueueElement;
      PROCEDURE (VAR q: FIFOQueueDesc) Dequeue (): QueueData;
      PROCEDURE (VAR q: FIFOQueueDesc) Empty (): BOOLEAN;
      PROCEDURE (VAR q: FIFOQueueDesc) Enqueue (d: QueueData);
      PROCEDURE (VAR q: FIFOQueueDesc) First (): QueueElement;
      PROCEDURE (VAR q: FIFOQueueDesc) Next (this: QueueElement): QueueElement;
      PROCEDURE (VAR q: FIFOQueueDesc) Remove (d: QueueData);
      PROCEDURE (VAR q: FIFOQueueDesc) Zap;
```

```
END ;

QueuedSlotID = POINTER TO QueuedSlotIDDesc;
QueuedSlotIDDesc = RECORD (QueueDataDesc)
   slotID-: INTEGER;
END ;

Process = POINTER TO ProcessDesc;
ProcessDesc = RECORD
   vp-: VirtualProcessor;
   prio-: OCT.Priority;
   state-: OCT.State;
   suspendCount-: SHORTINT;
   schedTicks-, sleepTicks-, maxSlice-: OCT.Ticks;
   alarms-: FIFOQueue;
   qsid-: QueuedSlotID;
   slotID-: INTEGER;
   code-: OCT.Code;
   PROCEDURE (thisProcess: Process) CreateAlarm (kind: SHORTINT; delay, maxSlice,
      cycleTicks: OCT.Ticks; priority: OCT.Priority; code: OCT.Code):
      OCT.AlarmHandle;
   PROCEDURE (thisProcess: Process) DestroyAlarm (alarmHandle: OCT.AlarmHandle);
   PROCEDURE (this: Process) GetPriority (): OCT.Priority;
   PROCEDURE (thisProcess: Process) Resume (proc: Process);
   PROCEDURE (thisProcess: Process) SetPriority (prio: OCT.Priority);
   PROCEDURE (thisProcess: Process) Sleep (ticks: OCT.Ticks);
   PROCEDURE (thisProcess: Process) Suspend (proc: Process);
END ;

Condition = POINTER TO ConditionDesc;
ConditionDesc = RECORD
   PROCEDURE (VAR this: ConditionDesc) Broadcast;
   PROCEDURE (VAR this: ConditionDesc) Signal;
   PROCEDURE (VAR this: ConditionDesc) Wait (VAR m: MutexDesc);
END ;

QueueElement = POINTER TO QueueElementDesc;
QueueElementDesc = RECORD
   data-: QueueData;
   next-: QueueElement;
END ;
```

```
QueueData = POINTER TO QueueDataDesc;
QueueDataDesc = RECORD
   PROCEDURE (VAR qdata: QueueDataDesc) GetPriority (): OCT.Priority;
END ;

Mutex = POINTER TO MutexDesc;
MutexDesc = RECORD (SemaphoreDesc) END ;

ProcessList = POINTER TO ProcessListDesc;
ProcessListDesc = RECORD
   inUse-: INTEGER;
   processList-: POINTER TO ARRAY OF Slot;
   PROCEDURE (VAR this: ProcessListDesc) ChooseSlotID (proc: Process): INTEGER;
   PROCEDURE (VAR this: ProcessListDesc) FindSlot (): INTEGER;
   PROCEDURE (VAR this: ProcessListDesc) GetProcess (sid: INTEGER): Process;
   PROCEDURE (VAR this: ProcessListDesc) ReleaseSlot (sid: INTEGER);
   PROCEDURE (VAR this: ProcessListDesc) SetProcess (proc: Process; sid: INTEGER);
END ;

QPSpace = POINTER TO QPSpaceDesc;
QPSpaceDesc = RECORD
   qps-: POINTER TO ARRAY OF QueuedSlotID;
   PROCEDURE (VAR this: QPSpaceDesc) Allocate (slotID: INTEGER): QueuedSlotID;
   PROCEDURE (VAR this: QPSpaceDesc) Deallocate (qs: QueuedSlotID);
END ;

Queue = POINTER TO QueueDesc;
QueueDesc = RECORD
   n-: LONGINT;
   head-: QueueElement;
END ;

QueueElementSpace = POINTER TO QueueElementSpaceDesc;
QueueElementSpaceDesc = RECORD
   elems-: POINTER TO ARRAY OF QueueElement;
   inUse-: POINTER TO ARRAY OF BOOLEAN;
   PROCEDURE (VAR qsp: QueueElementSpaceDesc) Allocate (): QueueElement;
   PROCEDURE (VAR qsp: QueueElementSpaceDesc) Deallocate (q: QueueElement):
        INTEGER;
END ;
```

```
Scheduler = POINTER TO SchedulerDesc;
SchedulerDesc = RECORD
   stopSchedulingCount-: SHORTINT;
   es-: QPSpace;
   pl-: ProcessList;
   maxPriority-: OCT.Priority;
   ReadyQueues-: POINTER TO ARRAY OF FIFOQueue;
   AsleepQueues-: POINTER TO ARRAY OF FIFOQueue;
   SuspendedQueues-: POINTER TO ARRAY OF FIFOQueue;
   AsleepSuspendedQueues-: POINTER TO ARRAY OF FIFOQueue;
   oldSchedulerISR-, BeforeProc, AfterProc, IdleProcessProc: OCT.Code;
   PROCEDURE (VAR this: SchedulerDesc) DeinstallProcess (proc: Process);
   PROCEDURE (VAR this: SchedulerDesc) DispatchNext;
   PROCEDURE (VAR this: SchedulerDesc) DoWakeups (VAR rescheduleNecessary:
      BOOLEAN);
   PROCEDURE (VAR this: SchedulerDesc) GetRealTime (): LONGINT;
   PROCEDURE (VAR this: SchedulerDesc) ImmediateSwitchUnlock;
   PROCEDURE (VAR this: SchedulerDesc) InstallProcess (proc: Process; state:
      OCT.State; prio: OCT.Priority);
   PROCEDURE (VAR this: SchedulerDesc) Kill (p: Process);
   PROCEDURE (VAR this: SchedulerDesc) KillAlarms (p: Process);
   PROCEDURE (VAR this: SchedulerDesc) Lock;
   PROCEDURE (VAR this: SchedulerDesc) NextProcess (): QueuedSlotID;
   PROCEDURE (VAR this: SchedulerDesc) Preempt;
   PROCEDURE (VAR this: SchedulerDesc) Reschedule;
   PROCEDURE (VAR this: SchedulerDesc) Resume (p: Process);
   PROCEDURE (VAR this: SchedulerDesc) Sleep (p: Process; ticks: OCT.Ticks);
   PROCEDURE (VAR this: SchedulerDesc) StartScheduling;
   PROCEDURE (VAR this: SchedulerDesc) StopScheduling;
   PROCEDURE (VAR this: SchedulerDesc) Suspend (p: Process);
   PROCEDURE (VAR this: SchedulerDesc) Unlock;
END ;

Semaphore = POINTER TO SemaphoreDesc;
SemaphoreDesc = RECORD
   PROCEDURE (VAR this: SemaphoreDesc) Signal;
   PROCEDURE (VAR this: SemaphoreDesc) Wait;
END ;

Slot = POINTER TO SlotDesc;
```

```
    SlotDesc = RECORD
        proc-: Process;
        used-: BOOLEAN;
    END ;


VAR
    Kernel-: Scheduler;
    currProcess, idleProcess-: Process;
    inISRcount-: LONGINT;
    lockvar-, numberOfProcesses-: INTEGER;
    oldProcess: Process;
    outOptions: SET;
    prcbPtr-: OCT.ProcessControlBlock;
    realTime-: LONGINT;
    space-: QueueElementSpace;

    PROCEDURE DeinstallScheduler (scheduler: Scheduler);
    PROCEDURE DeleteAlarm (this: Alarm);
    PROCEDURE DeleteConditionD (VAR this: ConditionDesc);
    PROCEDURE DeleteFIFOQueueD (VAR q: FIFOQueueDesc);
    PROCEDURE DeleteMutexD (VAR this: MutexDesc);
    PROCEDURE DeleteProcess (this: Process);
    PROCEDURE DeleteProcessListD (VAR this: ProcessListDesc);
    PROCEDURE DeleteQPSpaceD (VAR this: QPSpaceDesc);
    PROCEDURE DeleteQueueD (VAR q: QueueDesc);
    PROCEDURE DeleteQueueDataD (VAR qdata: QueueDataDesc);
    PROCEDURE DeleteQueueElemSpaceD (VAR qsp: QueueElementSpaceDesc);
    PROCEDURE DeleteQueueElementD (VAR qelem: QueueElementDesc);
    PROCEDURE DeleteQueuedSlotIDD (VAR qsid: QueuedSlotIDDesc);
    PROCEDURE DeleteSchedulerD (VAR this: SchedulerDesc);
    PROCEDURE DeleteSemaphoreD (VAR this: SemaphoreDesc);
    PROCEDURE DeleteSlotD (VAR this: SlotDesc);
    PROCEDURE DeleteVirtualProcessorD (VAR this: VirtualProcessorDesc);
    PROCEDURE Exit;
    PROCEDURE IdleProcess;
    PROCEDURE Init (nrOfProcesses: INTEGER; maxElemsQES: LONGINT): BOOLEAN;
    PROCEDURE InitAlarm (this: Alarm; maxSlice, cycleTicks: OCT.Ticks; code:
        OCT.Code);
    PROCEDURE InitConditionD (VAR this: ConditionDesc; initial: BOOLEAN);
    PROCEDURE InitFIFOQueueD (VAR q: FIFOQueueDesc);
```

```
    PROCEDURE InitMutexD (VAR this: MutexDesc);
    PROCEDURE InitProcess (this: Process; maxSlice: OCT.Ticks; code: OCT.Code;
        stackSize: LONGINT);
    PROCEDURE InitProcessListD (VAR this: ProcessListDesc);
    PROCEDURE InitQPSpaceD (VAR this: QPSpaceDesc);
    PROCEDURE InitQueueD (VAR q: QueueDesc);
    PROCEDURE InitQueueDataD (VAR qdata: QueueDataDesc);
    PROCEDURE InitQueueElementD (VAR qelem: QueueElementDesc);
    PROCEDURE InitQueueElementSpaceD (VAR qsp: QueueElementSpaceDesc; maxElems:
        LONGINT): BOOLEAN;
    PROCEDURE InitQueuedSlotIDD (VAR qsid: QueuedSlotIDDesc);
    PROCEDURE InitSchedulerD (VAR this: SchedulerDesc; maxPriority: OCT.MaxPriority);
    PROCEDURE InitSemaphoreD (VAR this: SemaphoreDesc; initCount: INTEGER);
    PROCEDURE InitSlotD (VAR this: SlotDesc);
    PROCEDURE InitVirtualProcessorD (VAR this: VirtualProcessorDesc; stackSize:
        LONGINT);
    PROCEDURE InstallScheduler (scheduler: Scheduler; ticksPerSecond: INTEGER);
    PROCEDURE OutLine (s: ARRAY OF CHAR);
    PROCEDURE PrintQueues;
    PROCEDURE Start (code: OCT.Code; ticksPerSecond: INTEGER; scheduler: Scheduler);
    PROCEDURE timerisr;

END Objix.
```

MODULE OBoardSpecific;

```
CONST
    set* = 0; reset* = 1;

    (* timer-chip 82C54 *)
    TimerBaseAdr* = 0DFE00000H; (* base address of the timer chip on the EVCA960 Board
        *)
    TimerInterruptNr* = 226; (* internal interrupt number of the timer interrupt *)
    TimerExtInterruptNr* = 3; (* hardwired interrupt from the timer to the interrupt controller
        on the EVCA960 Board *)
    TimerFrequency* = 8; (** in MHz *)

    PROCEDURE LED (bits: SET; mode: SHORTINT);

END OBoardSpecific.
```

MODULE OTimer;

```
    (* constant declarations for the timer-chip 82c54 *)
```

```
IMPORT OBoard := OBoardSpecific, OCT := OCommonTypes, OAsmInterface;

CONST
   InterruptNumber= OBoard.TimerInterruptNr;
   ExternalInterruptNumber = OBoard.TimerExtInterruptNr;
   TimerBaseAdr = OBoard.TimerBaseAdr; (* base address of timer *)
   Counter0 = TimerBaseAdr + 0; (* addr. of the 1. counter on the EVCA960 Board *)
   Counter1 = TimerBaseAdr + 1; (* addr. of the 2. counter *)
   Counter2 = TimerBaseAdr + 2; (* addr. of the 3. counter *)
   ControlReg = TimerBaseAdr + 3; (* addr. of the control register *)
   Roll32Bits = 0FFFFFFFFH;

   RegionBurst = 0; RegionReady = 1; RegionPipeline = 2; RegionNRAD = 3;
   RegionNRDD = 8; RegionNXDA = 10; RegionNWAD = 12; RegionNWDD = 17;
   RegionBusWidth = 19; RegionByteOrder = 22;

   CrystalTime = OBoard.TimerFrequency;

   PROCEDURE InitTimer (ct: OCT.ControlTable; ticksPerSecond: INTEGER);
   PROCEDURE DisableTimer* (ct: OCT.ControlTable);

END OTimer.
```

MODULE OAsmInterface;

```
   PROCEDURE Lock; END Lock;
   PROCEDURE Unlock; END Unlock;

   PROCEDURE SysCtl (src1, src2, src3: LONGINT); END SysCtl;

   PROCEDURE GetIMSK (): SET; END GetIMSK;
   PROCEDURE SetIMSK (s: SET); END SetIMSK;

   PROCEDURE GetIPND (): SET; END GetIPND;
   PROCEDURE SetIPND (s: SET); END SetIPND;

   PROCEDURE MODAC (mask, source: LONGINT): LONGINT; END MODAC;
      (* tmp := AC; AC := (source AND mask) OR (AC AND NOT mask); RETURN tmp *)
   PROCEDURE MODTC (mask, source: LONGINT): LONGINT; END MODTC;
      (* tmp := TC; TC := (source AND mask) OR (tmp AND NOT mask); RETURN tmp *)
   PROCEDURE MODPC (source, mask: LONGINT): LONGINT; END MODPC;
```

```
    (* tmp := PC;  := (source AND mask) OR (PC AND NOT mask); RETURN tmp *)

  PROCEDURE timerISR; END timerISR;
  PROCEDURE StartCurrentProcess; END StartCurrentProcess;
  PROCEDURE ImmediateSwitchUnlock; END ImmediateSwitchUnlock;

END OAsmInterface.
```

MODULE OCommonConstants;

```
CONST
  ProcessStackSize = 4096; IdleProcessStackSize = 256; AlarmStackSize = 1024;

  MinPriority = 0;

  StandardSlice = 100; MaxSlice = MAX(LONGINT);

  AlarmCyclic = 1; AlarmRelative = 2;

  StackInit = 444;

  NoError = 0; InvalidParameter = -2; NoIndex = -1;

  StateErroneous = -2; StateUnready = -1;
  StateRunning = 1; StateReady = 2; StateAsleep = 3; StateSuspended = 4;
  StateAsleepSuspended = 5; StateZombie = 6;

  OffsetR0 = 0; OffsetSP = 4; OffsetRIP = 8; OffsetR4 = 16; OffsetR8 = 32;
  OffsetR12 = 48; OffsetG0 = 64; OffsetG4 = 80; OffsetG8 = 96; OffsetG12 = 112;
  OffsetFP = 124; OffsetPC = 128; OffsetAC = 132; OffsetTC = 136; OffsetPFP = 152;

  PROCEDURE Init*(): BOOLEAN;

END OCommonConstants.
```

MODULE OCommonTypes;

```
IMPORT SYSTEM, OCC := OCommonConstants, Out, OAsmInterface;

  (* Mnemonics, Register Names:
  r0 => Previous Frame Pointer PFP
  r1 => Stack Pointer SP
  r2 => Return Instruction Pointer RIP
```

```
    g15 => Frame Pointer FP
    sf0  => Interrupt Pending Register IPND
    sf1  => Interrupt Mask Register IMSK
    sf2  => DMA Command Register DMAC
    *)

•TYPE
    Code = PROCEDURE;

    Register = LONGINT;
    LocalRegisters = RECORD
        r0, r1, r2, r3, r4, r5, r6, r7, r8 , r9 ,  r10  , r11 , r12 , r13 , r14 , r15: Register
    END;

    GlobalRegisters = RECORD
        go, g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13, g14, g15: Register
    END;

    Registers = RECORD
        r0, r1, r2, r3, r4, r5, r6, r7, r8 , r9 ,  r10  , r11 , r12 , r13 , r14 , r15: Register;
        g0, g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13, g14, g15: Register;
        pc, ac, tc, sf0, sf1, sf2: Register;          (* 80960CA only supports sf0, sf1 and sf2 *)
        (*, sf3, sf4, sf5, sf6, sf7, sf8, sf9, sf10, sf11, sf12, sf13, sf14,
        sf15, sf16, sf17, sf18, sf19, sf20, sf21, sf22, sf23, sf24, sf25, sf26, sf27,
        sf28, sf29, sf30, sf31: Register; *)
        pfp: Register;
        (* The offsets of r0..r15, g0..15, pc, ac, tc and pfp within the record must remain the
            same, as they are hardcoded in OAsmInterface.s in the timerisr when accessing
            proc.vp.regs. *)
    END;


    InterruptVectorDesc = RECORD (* SIZE: 4 *)
    END;

    InterruptTable = POINTER TO InterruptTableDesc;
    InterruptTableDesc = ARRAY 1 OF RECORD (* SIZE: 1040 *)
            (* must be declared as ARRAY 1 OF RECORD ... END because then the array
                elements all have a static type and there is no need for a dynamic type, i.e. there is
                no type descriptor. Pointer to records are handled as indirect pointers (pointer to
                a block which contains a pointer to the type descriptor and a pointer to the actual
                record). As that table (and the following) are system data structures, there is no
```

```
            type information for the fields when returned by the system. Thus all pointers
            must be direct pointers which is only possible if they are pointers to ARRAY 1 of
            RECORD ... END instead of pointers to RECORD ... END. *)
    PendingPriorities: SET; (* offset 0 *)
    PendingInterrupts: ARRAY 8 OF SET; (* offset 4 *)
    Entries: ARRAY 248 OF InterruptVectorDesc (* offset 36 *)
END;


CallEntry = RECORD
END;

FaultTable = POINTER TO FaultTableDesc;
FaultTableDesc = ARRAY 1 OF RECORD
    ParallelFaultEntry: CallEntry;
    TraceFaultEntry: CallEntry;
    OperationFaultEntry: CallEntry;
    ArithmeticFaultEntry: CallEntry;
    ConstraintFaultEntry: CallEntry;
    ProtectionFaultEntry: CallEntry;
    TypeFaultEntry: CallEntry;
END;


ControlTable = POINTER TO ControlTableDesc;
ControlTableDesc• = ARRAY 1 OF RECORD
    IPB0, IPB1: LONGINT; (* IP Breakpoint *)
    DAB0, DAB1: LONGINT; (* Data Address Breakpoint *)
    IMAP0, IMAP1, IMAP2: LONGINT; (* Interrupt Map *)
    ICON: LONGINT;
    MCON0, MCON1, MCON2, MCON3, MCON4, MCON5, MCON6, MCON7,
    MCON8, MCON9, MCON10, MCON11, MCON12, MCON13, MCON14, MCON15:
        LONGINT;
      (* Memory Region Configuration *)
    BPCON: LONGINT; (* Breakpoint Control *)
    TC: LONGINT; (* Trace Control *)
    BCON: LONGINT; (* Bus Configuration *)
END;


ProcedureEntry• = RECORD
```

```
END;

SystemProcedureTable = POINTER TO SystemProcTableDesc;
SystemProcTableDesc• = ARRAY 1 OF RECORD
    SupervisorStackPtrBase: SYSTEM.PTR;
    Entries: ARRAY 260 OF ProcedureEntry;
END;


ProcessControlBlock = POINTER TO ProcessControlBlockDesc;
ProcessControlBlockDesc•=   ARRAY 1 OF RECORD (* SIZE: 48 *)
    FaultTableBA: FaultTable;       (* base address of fault table *) (* offset 0 *)
    ControlTableBA: ControlTable; (* offset 4 *)
    ACRegisterInitialImage: Register; (* offset 8 *)
    FaultCW: SET; (* Fault configuration word *) (* offset 12 *)
    InterruptTableBA: InterruptTable; (* offset 16 *)
    SystemProcedureTableBA: SystemProcedureTable; ( offset 20 )
    InterruptStackPointer: LONGINT; (* offset 28 *)
    InstructionCacheCW: SET; (* offset 32 *)
    RegisterCacheCW: SET (* offset 36 *)
END;


Priority = SHORTINT;

State = SHORTINT;

Error = INTEGER;

Ticks = LONGINT;

AlarmHandle = INTEGER;


PROCEDURE PrintRegs (l: SYSTEM.PTR);
PROCEDURE PrintRegisters2 (l: SYSTEM.PTR);
PROCEDURE PrintRegisters (regs: Registers);
PROCEDURE PrintR0R1R2G15 (r0, r1, r2, g15: LONGINT);
PROCEDURE GetProcessorPriority (): SHORTINT;
PROCEDURE SetProcessorPriority (newPriority: SHORTINT): SHORTINT;
PROCEDURE DecodeAC (reg: LONGINT);
```

```
    PROCEDURE DecodePC (reg: LONGINT);
    PROCEDURE DecodeIPND (reg: SET);
    PROCEDURE DecodeIMSK (reg: SET);
    PROCEDURE InitFaultTable (this: FaultTable);
    PROCEDURE PrintFaultTable (this: FaultTable);
    PROCEDURE DecodeIMAP (map: LONGINT; i: INTEGER);
    PROCEDURE DecodeICON (reg: LONGINT);
    PROCEDURE DecodeMCON (reg: LONGINT; i: INTEGER);
    PROCEDURE DecodeBPCON (reg: LONGINT);
    PROCEDURE DecodeTC (reg: LONGINT);
    PROCEDURE DecodeBCON (reg: LONGINT);
    PROCEDURE PrintControlTable (this: ControlTable);
    PROCEDURE InitInterruptTable (this: InterruptTable);
    PROCEDURE PrintInterruptTable (this: InterruptTable);
    PROCEDURE SetInterruptVector (this: InterruptTable; nr: INTEGER; code: Code;
        reenable: BOOLEAN);
    PROCEDURE GetInterruptVector (this: InterruptTable; nr: INTEGER; VAR code: Code;
        reenable: BOOLEAN);
    PROCEDURE InitSystemProcTable (this: SystemProcedureTable; supervisorStack:
        ARRAY OF SYSTEM.BYTE);
    PROCEDURE PrintSystemProcTable (this: SystemProcedureTable);

    PROCEDURE InitPRCBDesc (VAR this: ProcessControlBlockDesc);
    PROCEDURE PrintPRCB (this: ProcessControlBlock);

    PROCEDURE Init (): BOOLEAN;

END OCommonTypes.
```

Syntax von Oberon-2

```
Module = MODULE ident ";" [ImportList] DeclSeq  [BEGIN StatementSeq] END ident ".".
ImportList = IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".
DeclSeq = { CONST {ConstDecl ";" } | TYPE {TypeDecl ";"} | VAR {VarDecl ";"}}
    {ProcDecl ";" | ForwardDecl ";"}.
ConstDecl = IdentDef "=" ConstExpr.
TypeDecl = IdentDef "=" Type.
VarDecl = IdentList ":" Type.
ProcDecl = PROCEDURE [Receiver] IdentDef [FormalPars] ";" DeclSeq [BEGIN
    StatementSeq] END ident.
ForwardDecl = PROCEDURE "^" [Receiver] IdentDef [FormalPars].
FormalPars = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection = [VAR] ident {"," ident} ":" Type.
Receiver = "(" [VAR] ident ":" ident ")".
Type = Qualident
    | ARRAY [ConstExpr {"," ConstExpr}] OF Type
    | RECORD ["("Qualident")"] FieldList {";" FieldList} END
    | POINTER TO Type
    | PROCEDURE [FormalPars].
FieldList = [IdentList ":" Type].
StatementSeq= Statement {";" Statement}.
Statement   =[  Designator ":=" Expr
    | Designator ["(" [ExprList] ")"]
    | IF Expr THEN StatementSeq {ELSIF Expr THEN StatementSeq} [ELSE StatementSeq]
        END
    | CASE Expr OF Case {"|" Case} [ELSE StatementSeq] END
    | WHILE Expr DO StatementSeq END
    | REPEAT StatementSeq UNTIL Expr
    | FOR ident ":=" Expr TO Expr [BY ConstExpr] DO StatementSeq END
    | LOOP StatementSeq END
    | WITH Guard DO StatementSeq {"|" Guard DO StatementSeq} [ELSE StatementSeq]
        END
    | EXIT
    | RETURN [Expr]].
Case = [CaseLabels {"," CaseLabels} ":" StatementSeq].
CaseLabels = ConstExpr [".." ConstExpr].
Guard = Qualident ":" Qualident.
ConstExpr  = Expr.
Expr = SimpleExpr [Relation SimpleExpr].
SimpleExpr = ["+" | "-"] Term {AddOp Term}.
Term = Factor {MulOp Factor}.
Factor = Designator ["(" [ExprList] ")"] | number | character | string | NIL | Set | "(" Expr ")" |
      " ~ " Factor.
Set    = "{" [Element {"," Element}] "}".
Element = Expr [".." Expr].
Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp = "+" | "-" | OR.
MulOp = " * " | "/" | DIV | MOD | "&".
Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " | "(" Qualident ")"}.
ExprList =  Expr {"," Expr}.
```

```
IdentList = IdentDef {"," IdentDef}.
Qualident = [ident "."] ident.
IdentDef = ident [" * " | "-"].
```

Literaturverzeichnis

[80960] „80960CA User's Manual". Intel Corporation, Santa Clara. Order Number 270710-001

[FAN94] Richard Fanta: „OBJIX, Object Implementation of MINIX". Diploma Thesis at the Department of Computer Science, Prague, 1994

[HAB95] Richard Hable: „Oberon 960, ein Oberon-2-Compiler für 80960 Prozessoren". Diploma Thesis at the FIM, Linz, 1995

[HAM92] R. Hampl: „Object Implementation of MINIX". Master Thesis at the Department of Computer Science, CTU Prague, 1992

[HEE93] Norbert Heesel: „Mikrocontroller-Praxis: ein praxisorientierter Leitfaden für Hard- und Software-Entwicklung auf der Basis der 80(C)51x-Familie". Vieweg 1993. ISBN 3-528-05366-6

[iRMK] „iRMK™ 960, REAL-TIME Kernel Reference Manual". Intel Corporation, Santa Clara. Order Number 463863-001

[KLE92] J. Kleindienst: „Object Implementation of MINIX". Master Thesis at the Department of Computer Science, CTU Prague, 1992

[MOE93] Hanspeter Mössenböck: „Object-Oriented Programming in Oberon-2". Springer 1993. ISBN 0-387-56411-X

[MOW92] Hanspeter Mössenböck, Niklaus Wirth: „The Programming Language Oberon-2". On-line Dokumentation im Oberon-System, „Oberon2.Report.Text"

[OOP90] Workshop: Using OOP for Real-Time Programming, OOPSLA/ ECOOP ´90 Report, Addendum to the Proceedings

[RAJ89] R. Rajkumar, „Task Synchronization in Real-Time Systems", Ph. D. Dissertation, Carnegie Mellon University, August 1989

[RUM93] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: „Objektorientiertes Modellieren und Entwerfen". Hanser 1993. ISBN 3-446-17520-2

[SIL94] A. Silberschatz, P. Galvin, „Operating system concepts", Addison-Wesley 1994. ISBN 0-201-59292-4

[SRL87] L. Sha, R. Rajkumar, J. P. Lehoczky, „Priority inheritance protocols: An approach to real-time synchronization", Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987

[STR90] Margaret A. Ellis, Bjarne Stroustrup: „The Annotated C++ Reference Manual". Addison-Wesley 1990. ISBN 0-201-51459-1

[STR92] Bjarne Stroustrup: „Die C++-Programmiersprache“, Addison-Wesley 1992. ISBN 3-89319-386-3

[TAN92] Andrew S. Tanenbaum: „Modern Operating Systems“. Prentice-Hall 1992. ISBN 0-13-595752-4

[TEM95] Josef Templ: „Ofront User Guide“, URL: ftp://oberon.ssw.uni-linz.ac.at/pub/Papers/OfrontUser.A4.ps.Z

[TOK89] H. Tokuda, C. W. Mercer, Y. Ishikawa, T. E. Marchok, „Priority inversions in real-time communication“, Proceedings of 10th IEEE Real-Time Systems Symposium, December 1989

Index

8

80960 . 5; 16; 19; 34; 35; 36; 40; 45; 105 *See* i960

A

Aufrufhierarchie 6; 52; 53; 54

C

CPU 8
Crosscompiler 5; 21; 22; 73

D

deadlock detection 73
Deadlock prevention 73

E

Echtzeit 6; 9; 10; 25; 41; 83; 84
Einfachheit 7; 27; 72
Embedded-Controller 9
EPROM 18; 25

F

Finalisierung 27; 77

G

globale Register 19

H

Hotspots 54

I

i960 ..... 5; 6; 16; 18; 20; 21; 22; 34; 53; 80
Initialisierungsphase 52; 67; 70
Interface 8; 21
Interrupt 9; 19; 35; 39; 40; 41; 45; 46; 53; 71; 83; 84; 98; 100

L

Lesbarkeit 7; 71
lokale Register 19

M

Mikrocomputer 6; 8
Mikrocontroller 5; 6; 8; 9; 16; 105
Mikroinstruktionen 36; 37; 38; 39; 40; 41; 45; 46; 47; 51
Mikrokernel 6; 10; 23; 25; 34; 52; 61; 71; 72; 73
Mikroprozessor 6; 8

O

Oberon-2 11
Oberon-System 20; 21; 71; 73
Objektorientiertheit 7; 71
OOP 10

P

parallel 16; 17; 18; 36; 68; 72; 83
Portabilität 7; 21; 72
previous-frame pointer 19
Priority inheritance protocol 72; 105
priority inversion 72
Prozeßwechsel 6; 29; 33; 34; 39; 46; 51; 53; 66; 72; 83; 84

R

RAM 8; 9; 18; 25
register bypassing 17
register scoreboarding 17
Register-Cache 17
result forwarding 17
return-instruction pointer 19
RISC 18
ROM 8; 9; 18; 25

S

Scheduling 6; 29; 30; 31; 32; 33; 40; 41; 47; 52; 53; 54; 64; 65; 66; 71; 7
stack pointer 19; 35; 36; 40
Stack-Frame 17; 39
static branch prediction 17
superskalar 16

T

Timesharing 29; 30
Typerweiterung 11; 77
Typ-Guard 12; 13

Z

Zeitscheibe 29; 30; 33; 52; 53; 55; 62; 63; 64; 65; 66; 82

Abbildungsverzeichnis

Abbildung 1, beispielhafter Aufbau eines Mikrocomputer-Systems 8
Abbildung 2, beispielhafter Aufbau eines Mikrocontroller-Systems 9
Abbildung 3, Typerweiterung 11
Abbildung 4, Programmierumgebung auf i960 18
Abbildung 5, Hauptmodul mit zusammengeklappten Faltelementen 20
Abbildung 6, Hauptmodul mit geöffneten Faltelementen 21
Abbildung 7, Weg von der Source zum Executable 23
Abbildung 8, ursprüngliche Klassenhierarchie 24
Abbildung 9, Schema einer Mikrokontroller-Applikation 25
Abbildung 10, Objektmodell des Kernels 28
Abbildung 11, Prozeßzustandsüberführungsdiagramm 31
Abbildung 12, Zustand nach InitProcess und InstallProcess 35
Abbildung 13, Zustand vor dem Aufruf von StartCurrentProcess 36
Abbildung 14, Zustand nach dem Aufruf von StartCurrentProcess (Mikroinstruktionen als hochgestellte Indizes) 37
Abbildung 15, Zustand am Ende von StartCurrentProcess 38
Abbildung 16, Zustand nach dem Return von StartCurrentProcess (Mikroinstruktionen als hochgestellte Indizes) 39
Abbildung 17, Zustand vor Eintreten des Interrupts 40
Abbildung 18, Zustand nach Auftreten des Interrupts 41
Abbildung 19, Zustand vor Aufruf der Oberon-Schedulingprozedur 42
Abbildung 20, Zustand nach Rückkehr aus Oberon-Schedulingprozedur 43
Abbildung 21, Zustand am Ende von timerISR 44
Abbildung 22, Zustand nach dem Interrupt-Return (Mikroinstruktionen als hochgestellte Indizes) 45
Abbildung 23, Zustand vor Aufruf von ImmediateSwitchUnlock 46
Abbildung 24, Zustand am Beginn von ImmediateSwitchUnlock 46
Abbildung 25, Zustand nach Speicherung des Zustandes im alten Prozeß 47
Abbildung 26, Zustand nach Restaurierung des Zustands aus dem neuen Prozeß 48
Abbildung 27, Zustand nach dem Return (Mikroinstruktionen als hochgestellte Indizes) 49
Abbildung 28, Aufrufhierarchie 52
Abbildung 29, zeitlicher Ablauf eines zyklischen Alarms 53
Abbildung 30, Objektdiagramm Process und VirtualProcessor 77
Abbildung 31, Objektdiagramm Scheduler 80
Abbildung 32, Objektdiagramm Semaphore 82
Abbildung 33, Objekdiagramm Mutual Exclusion 83
Abbildung 34, Objektdiagramm Alarm 84
Abbildung 35, Gesamt-Objektdiagramm 85

Tabellenverzeichnis

Tabelle 1, Mikroinstruktionen bei einem Call 36
Tabelle 2, Mikroinstruktionen bei einem Return 38
Tabelle 3, Mikroinstruktionen bei einem Interrupt 40
Tabelle 4, Mikroinstruktionen bei einem Interrupt-Return 44

Inhaltsverzeichnis

ZUSAMMENFASSUNG - ABSTRACT 5

ZUSAMMENFASSUNG 5
ABSTRACT 5

ÜBERSICHT UND ZIELE 6

ÜBERSICHT 6
ZIELE 7

GRUNDLAGEN 8

MIKROCOMPUTER / MIKROCONTROLLER 8
ECHTZEIT-BETRIEBSSYSTEM 9
MIKROKERNEL 9
OBJEKTORIENTIERTE PROGRAMMIERUNG (OOP) IM BEREICH VON ECHTZEIT-BETRIEBSSYSTEMEN 10
EINFÜHRUNG IN OBERON-2 10
EINIGE MERKMALE VON C++ 14
DER I960 16
Details über die Hardware 16
Details über die Software 18

IMPLEMENTIERUNG IM ÜBERBLICK 20

PROGRAMMENTWICKLUNG FÜR DEN I960 20
Entwicklungsumgebung 20
Von der Source zum Executable 21
OBJIX 23
ANALYSE DER BESTEHENDEN C++ IMPLEMENTATION 23
URSPRÜNGLICHE KLASSENHIERARCHIE 24
VEREINFACHUNGEN 24
Ein gemeinsamer Adreßraum 24
Methoden-Aufruf statt Message-Passing 24
Kein Loader 25
Kein Filesystem, Aufruf beliebiger C-Bibliotheksfunktionen 25
PORTIERUNG AUF OBERON-2 25
Umsetzung von Mehrfachvererbung auf Einfachvererbung 25
Laufzeittypinformation 26
Konstruktoren/Destruktoren 26
Sichtbarkeits- und Zugreifbarkeitsstrukturen 27
Weglassen nicht benötigter Klassen 27
AKTUELLER STAND 27
BESTEHENDE KLASSEN(BEZIEHUNGEN) 27
Beschreibung des Objektdiagramms 28

SCHEDULING UND PROZEßWECHSEL 29

GRUNDLEGENDES ÜBER SCHEDULING 29
ÄNDERUNG DES PROZEßZUSTANDES 30
Wie kann nun ein Anwenderprogramm die Möglichkeiten des Scheduling ausnutzen? 31
PROZEßWECHSEL 33
Freiwilliger Prozeßwechsel 33
Unfreiwilliger Prozeßwechsel (automatisches Scheduling) 33
PROZEßWECHSEL AUF DEM I960 IM DETAIL 34
Prozeßwechsel durch StartCurrentProcess 34
Prozeßwechsel durch einen Interrupt 39
Prozeßwechsel durch ImmediateSwitchUnlock 45

IMPLEMENTIERUNG IM DETAIL 50

Init (nrOfProcesses: INTEGER; maxElemsQES: LONGINT): BOOLEAN 50
Start (code: OCT.Code; ticksPerSecond: INTEGER; scheduler: Scheduler) 50
InitSchedulerD (VAR s: SchedulerDesc; maxPriority: OCT.Priority) 51

InstallScheduler (scheduler: Scheduler; ticksPerSecond: INTEGER) *51*
(VAR s: SchedulerDesc) InstallProcess (proc: Process; state: OCT.State; *51*
prio: OCT.Priority) *51*
Oberon Timer-ISR *51*
AUFRUFHIERARCHIE 51
ALARME 52
SEMAPHORE 56

ANWENDUNGEN 58

EINFACHE TESTFÄLLE ZUM ANFANGEN 58
PRODUCER / CONSUMER-PROBLEM 65

BEWERTUNG UND AUSBLICK 68

ZIELE 68
Objektorientiertheit, Erweiterbarkeit *68*
Echtzeitfähigkeit *68*
Lesbarkeit, Verständlichkeit *68*
Einfachheit *69*
Portabilität *69*
VORTEILE GEGENÜBER BISHERIGEN ENTWICKLUNGSUNTERSTÜTZUNGEN 69
AUSBLICK 69
Priority inheritance protocol *69*
Deadlock prevention, deadlock detection *70*
Browser für den Crosscompiler *70*
Oberon-to-C Translator *70*

USER MANUAL 71

ALLGEMEINES 71
KERNEL ALLGEMEIN 72
PROCESS 76
SCHEDULER 79
SEMAPHORE 82
MUTUAL EXCLUSION 83
ALARME 83
GESAMT-OBJEKTDIAGRAMM 85

KOMPLETTE PROGRAMMIERSCHNITTSTELLE DES MIKROKERNELS 86

DEFINITION OBJIX; 86
MODULE OBOARDSPECIFIC; 91
MODULE OTIMER; 91
MODULE OASMINTERFACE; 92
MODULE OCOMMONCONSTANTS; 93
MODULE OCOMMONTYPES; 93

SYNTAX VON OBERON-2 98

LITERATURVERZEICHNIS 98

INDEX 102

ABBILDUNGSVERZEICHNIS 104

TABELLENVERZEICHNIS 105

INHALTSVERZEICHNIS 106