Ein Werkzeug für die Systemprogrammierung von Realzeitsystemen auf der Basis von MODULA-2

Dipl.-Ing. Jürgen Stoll, München

Zusammenfassung

MODULA-2 ist eine Programmiersprache die auf Grund ihrer Konzeption auch sehr gut für die Systemprogrammierung geeignet ist. Allerdings stellt das in MODULA-2 verwendete Coroutinen-Konzept für die Programmierung von Realzeitsystemen keine befriedigende Lösung für die Steuerung paralleler Rechenprozesse dar. In diesem Bericht wird die Ersetzung des Coroutinen-Konzepts durch ein Rechenprozeßmodell vorgestellt, wobei das Rechenprozeßmodell mit einem Realzeitbetriebssystemkern realisiert wird. Nach der Vorstellung des Coroutinen-Konzepts werden die Schnittstellen im MODULA-2 Cross-Kompilierer und im Realzeitbetriebssystemkern analysiert und dann eine Realisierung ausführlich besprochen. Das Ergebnis stellt sich als ein Satz von Prozeduren dar, wobei die Prozedurdefinitionen (Definition-Part in MODULA-2) so zweckmäßig entworfen werden, daß es möglich ist, ohne weiteres andere Betriebssysteme einzusetzen und zwar "nur" durch Schreiben eines neuen Implementation - Parts. Es eröffnet sich somit - unterstützt durch MODULA-2 - die Möglichkeit, einheitliche Betriebssystemschnittstellen (Syntax und Semantik) ohne Beteiligung der verschiedenen Betriebssystemhersteller zu bekommen.

Schlüsselworte

Real- bzw. Echtzeitprogrammierung, MODULA-2, Coroutinen, Rechenprozeßmodell, Systemprogrammierung, Realzeit-Betriebssystemkern, hierarchisches verteiltes Prozeßrechnernetz.

Summary

MODULA-2 is a programming language which is also suitable for system level programming. However for programming embedded realtime systems, the coroutines as a concept for synchronization do not fulfill our requirements. In this paper the replacement of the coroutine model by a task model is described. After the introduction of the model of coroutines we are going to analyse the interfaces of the MODULA-2 cross-compiler and the realtime operating system kernel. A detailed description of the realization follows. The result is a set of procedures. The definitions of the procedure headers (Definition-Part in MODULA-2 terms) are constructed so that it is possible to adapt another operating system by merely writing a new Implementation-Part. With this approach we have a chance to get with the support of MODULA-2 a uniform operating system interface (syntax and semantics) without the participation of the various producers of operating systems.

Keywords

realtime programming, MODULA-2, Coroutines, task model, system programming, realtime operating system kernel, hierarchical distributed industrial control net.

1. EINLEITUNG

1.1. Motivation

Am Institut für Systemorientierte Informatik an der Universität der Bundeswehr München beschäftigen wir uns mit dem Thema "Gewährleistung einer eingeschränkten Funktionsfähigkeit von verteilten Realzeitsystemen beim Auftreten von Fehlern im Betrieb". Dazu wurde ein hierarchisch verteiltes Realzeitsystem <Obenhuber, Rzehak 83> auf der Basis des MC 68000 aufgebaut.

Bei der Auswahl der Sprache für die Systemprogrammierung dieses experimentellen Realzeitsystems war folgende Randbedingung zu beachten: Die Programmentwicklung für das verteilte Realzeitsystem soll auf einem Host-System (Perkin-Elmer 3240, Sprachen: FORTRAN, PASCAL, PEARL in Vorbereitung) mit Cross-Software durchgeführt werden.

1.2. Systemprogrammierung

Die Systemprogrammierung war bislang die Domäne der Assemblerprogrammierung mit all ihren Nachteilen.

Die Charakteristika bei der Systemprogrammierung sind:

(C1) Zugriff auf physikalische Adressen
(C2) Einzelbitverarbeitung
(C3) Aufbau von Datenstrukturen
(C4) Algorithmen für die Bearbeitung bzw. Verwaltung dieser Datenstrukturen

In Assembler kann man die Punkte (C1) und (C2) leicht beherrschen. Die Punkte (C3) und (C4) sind in Assembler sehr schwerfällig zu handhaben. Umgekehrt verhält es sich bei der Verwendung von Hochsprachen (z.B. PASCAL)

wobei die Punkte (C1) und (C2) manchmal gar nicht zu erfüllen sind.

Genau diese Lücke schließt MODULA-2. Um den häufig beobachteten Effizienzverlust beim Verwenden von Hochsprachen zu vermeiden, werden im Rahmen des LILITH-Projektes <Wirth 84> Vorschläge zur besseren Anpassung der Hardware an Hochsprachen gemacht.

1.3. Das Projekt LILITH

Die Programmausführungszeit für einen Algorithmus der in PASCAL kodiert ist, kann gegenüber der Ausführungszeit für den gleichen Algorithmus in FORTRAN kodiert um ein Mehrfaches größer sein. Gegenüber der Kodierung in Assembler kann dieser Faktor noch größer sein. Gleichzeitig ist der vom Kompilierer für eine Hochsprache erzeugte Kode wesentlich länger als der von einem geübten Assembler-Programmierer.

Diese Zahlen, die man messen und prüfen kann, müssen nun interpretiert werden. Man muß klar unterscheiden zwischen einer Programmiersprache und ihrer Implementierung.

Eine Programmiersprache ist ein Formalismus, der auf Grund der kompakten und formalen Definition seiner Konstrukte und deren eindeutigen Bedeutung sehr geeignet ist um Algorithmen und Datenstrukturen zu beschreiben. Von diesem Standpunkt aus ist der Begriff 'Sprache' sehr unglücklich und führt zu Mißinterpretationen. Eine Implementierung ist ein Mechanismus für die Interpretation von Algorithmen, welche mit einem Formalismus beschrieben sind. Daraus zog Wirth den Schluß, daß der Verlust an Effektivität nicht inhärent im Konzept der strukturierten Hochsprache, sondern in deren inadäquaten Implementierung begründet liegt. Die Zielsetzungen beim LILITH Projekt waren:

Es sollte ein Arbeitsplatzrechner mit einem geschlossenen Hardware- und Software - Ansatz entwickelt werden <Ohran 84>.

(A1) Auf dem Rechner soll nur eine Sprache (Formalismus) implementiert werden. Alle Programme (Algorithmen) sollen in dieser Sprache formuliert werden, ohne Ausnahme!

(A2) Das Betriebssystem soll für den Single-User-Betrieb konzipiert werden.

Durch diese Entscheidung umgeht man solche Probleme wie

- Rechenkernvergabe
- Schutzmechanismen
- Betriebsmittelverwaltung
- Abrechnung

(A3) Für den Arbeitsplatzrechner soll ein Prozessor verwendet werden der leistungsfähig genug ist, um die Rasteroperationen auf einem graphikfähigen Bildschirm und die Benutzerprogramme auszuführen.

1.4. MODULA-2

Aus der ersten Bedingung (A1) ergab sich folgendes Anforderungsprofil an die zu entwickelnde Sprache (MODULA-2).

Die Sprache muß sowohl die Formulierung von Algorithmen auf einem hohen Abstraktionsniveau als auch Operationen der Hardware direkt unterstützen.

MODULA-2 ist eine strukturierte blockorientierte Hochsprache, mit der man auf einem hohen Abstraktionsniveau Algorithmen und Datenstrukturen beschreiben kann. Mit den sogenannten 'Low-Level-Facilities' ist es in MODULA-2 möglich auf physikalische Adressen zuzugreifen und Einzelbitverarbeitung zu betreiben <Paul 84>, <Gutknecht 84>.

Durch das Modulkonzept mit kontrollierbaren Import- und Export-Möglichkeiten (Information-Hidding) und die getrennt übersetzbaren Moduln (Entwicklung größerer Programmpakete) wird der Systemprogrammierer aus der Sicht des Software-Engineering wirklich sehr gut unterstützt <Pomberger 84>.
Zusammenfassend gilt:

MODULA-2 ist im wesentlichen eine Kombination der hervorragenden Merkmale der drei Sprachen

(M1) PASCAL (Syntax weiter systematisiert) <McCormack, Gleaves 83>, <Summer, Gleaves>

(M2) MODULA-1 (schachtelbare Module mit kontrollierbaren Import/Export-Möglichkeiten) <Wirth 77>

(M3) MESA (getrennt übersetzbare Moduln mit Implementation- und Definition-Part) <Mitchell, Mayburg, Sweet 78>

1.5. Echtzeit-Systemprogrammierung

Neben der Programmierung von hardwarenahen Elementarfunktionen stellen die Echtzeit - oder Realzeitsysteme (Bild 1) eine weitere Aufgabenklasse im Bereich der Systemprogrammierung dar.

Die zusätzlichen Charakteristiken für die Systemprogrammierung von Einprozessor-Realzeitsystemen sind:

(C5) Verwaltung und Quasi-simultane Ausführung von parallel ablauffähigen Rechenprozessen.

(C6) Reaktion auf asynchrone Ereignisse in einer vorgegebenen anwendungsabhängigen Zeitspanne.

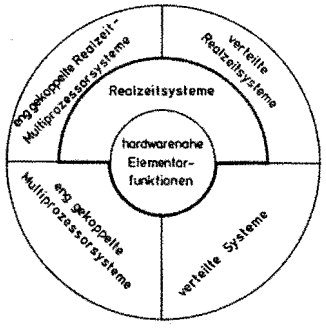

Bild 1: Klassen in der Systemprogrammierung

Für diese Punkte (C5) und (C6) stellt die Syntax von MODULA-2 keine Sprachkonstrukte zur Verfügung. Wegen (A2) war es auch nicht notwendig irgendwelche Annahmen über ein Betriebssystem - das normalerweise die in (A2) ausgeklammerten Punkte organisiert - zu machen <Wirth 83>.

Vielmehr wurden die Coroutinen als Werkzeug für die Lösung der Aufgaben der Punkte (C5) und (C6) im Laufzeitsystem (Modul SYSTEM) zur Verfügung gestellt.

2. Das Coroutinen-Modell

2.1 Allgemeine Vorstellung

Eines der Hauptmerkmale in der Prozeßautomatisierung ist die Steuerung und Regelung gleichzeitg ablaufender technischer Prozesse. Entsprechend benötigt man auf der Programmierseite adäquate Modelle und Prinzipien für die Abbildung dieser parallelen technischen Prozesse auf parallele Rechenprozesse.

Bekannte repräsentative Modelle sind:

- Coroutinen
- FORK- und JOIN-Anweisungen
- cobegin-Anweisungen
- PROCESS Deklarationen

Im folgenden wird das Coroutinen-Konzept (Kooperierende Routinen) näher erläutert <Andrews, Schneider 83>, <Dal Cin, Lutz, Risse 84>.

Coroutinen sind vergleichbar mit Unterprogrammen. Sie unterscheiden sich jedoch in der Art der Weitergabe der Kontrolle. Während bei Unterprogrammen der Kontrollfluß streng hierarchisch abläuft, ist die Kontrollvergabe bei Coroutinen symmetrisch organisiert <Conway 1963>. Beim Aufruf eines Unterprogramms beginnt dessen Ausführung immer mit der ersten Anweisung; es wird dann völlig abgearbeitet bis zur RETURN-Anweisung. Coroutinen können dagegen auch nur teilweise abgearbeitet werden. Ein erneuter Aufruf bewirkt, daß mit ihrer Ausführung am Unterbrechungspunkt fortgefahren wird.

Mit einer Anweisung, die hinsichtlich der Bedeutung der RESUME-Anweisung entspricht, wird die Kontrolle zwischen Coroutinen weitergegeben.

Wie bei einem Unterprogrammaufruf übergibt die Anweisung RESUME(Coroutine-name) die Kontrolle an die benannte Coroutine und speichert so viel Information der rufenden Coroutine ab, um bei einer späteren Kontrollübernahme an der Anweisung nach RESUME wieder fortzufahren. Diesen Vorgang nennt man Quasi-Nebenläufigkeit. Nach der Kontrollübergabe bleiben - anders als bei einem Unterprogramm - die Werte aller lokalen Objekte der Coroutine erhalten. Die Abarbeitung der Coroutine wird sozusagen am Unterbrechungspunkt eingefroren; man sagt auch, daß die Coroutine suspendiert wird.

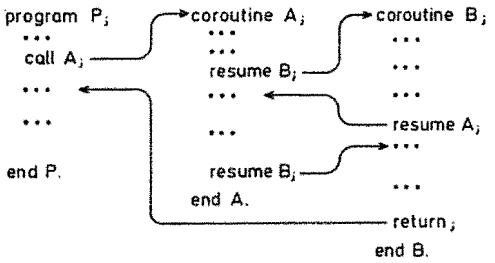

Bild 2:

Prinzip der Benutzung von Coroutinen aus <Andrews, Schneider 83>

Beim ersten Aufruf einer Coroutine wird mit der Abarbeitung am Beginn der Coroutine begonnen, d.h. Initialisierung der Speicherstruktur für die Statusinformation mit den Startwerten der Coroutine. Die Symmetrie der Kontrollübergabe oder die Gleichberechtigung der Coroutinen untereinander erkennt man daran, daß mit der RESUME-Anweisung jede Coroutine jede andere Coroutine aufrufen kann. Die Ko-operation der Routinen muß jedoch der Anwender festlegen.

Jede Coroutine kann als Implementierung eines Rechenprozesses betrachtet werden. Die Synchronisation der Rechenprozesse erfolgt mit der Ausführung von RESUME-Anweisungen. Für echte Parallelität in der Abarbeitung von Programmen (mehr als 1 Prozessor) sind Coroutinen nicht geeignet, da die Semantik der Coroutinen nur die Abarbeitung von einer Routine zur gleichen Zeit erlaubt.

Zusammengefaßt kann man sagen: Coroutinen sind Rechenprozesse bei denen die Kontrollvergabe fest vom Anwender vorgegeben wird und nicht dem zufälligen Eintreffen externer Ereignisse überlassen ist. Es ist möglich die einem Problem inhärente Parallelität auf der Programmebene zu modellieren, die Abarbeitung muß jedoch quasi-simultan erfolgen.

2.2. Umsetzung und Verwendung von Coroutinen in MODULA-2

Da in MODULA-2 die Coroutinen zu den 'Low-Level Facilities' gezählt werden, müssen die zugehörigen Typen (ADDRESS, PROCESS) und die entsprechenden Prozeduren (NEWPROCESS, TRANSFER, IOTRANSFER) vom Modul SYSTEM importiert werden. Mißverständlicherweise wird mit PROCESS eine Coroutine bezeichnet.

Mit der Prozedur NEWPROCESS werden Coroutinen kreiert. Der Aufruf hat die Form:

```
PROCEDURE NEWPROCESS
    (procid: PROC; a: ADDRESS; n: CARDINAL;
     VAR processid: PROCESS; ip: CARDINAL);
```

procid — steht für den Bezeichner einer Prozedur vom Typ PROC. Es sind nur parameterlose Prozeduren, die nicht in einer anderen Prozedur enthalten sein dürfen, zugelassen. Diese Prozedur enthält den Kode für die neu erzeugte Coroutine.

a — steht für die Startadresse des Arbeitsspeichers (workspace)

n — steht für die Größe des lokalen Speichers einschließlich der Kontext-Information.

processid steht für den Bezeichner der Coroutine. Durch den Aufruf wird dieser Variable ein Zeiger in den Arbeitsspeicher der neu kreierten Coroutine zugewiesen. Gleichzeitig wird der Status der Coroutine so initialisiert, daß bei einer ersten Kontrollübergabe die Ausführung am Anfang der Coroutine beginnt.

ip steht für die Interruptebene der Coroutine (vgl. Kapitel 3.2.)

Durch die Trennung von Kode und Daten (workspace) ist es möglich mehrere Coroutinen mit demselben Kode zu kreieren.

Mit einem Aufruf der Prozedur TRANSFER wird die Kontrolle von einer Coroutine zu einer anderen übergeben.

```
PROCEDURE TRANSFER
(VAR form-processid, to-processid: PROCESS);
```

from-processid steht für den Bezeichner der Coroutine, die die Kontrolle abgibt.

to-processid steht für den Bezeichner der Coroutine, die die Kontrolle erhält.

Die rufende Coroutine wird angehalten. Für eine spätere Fortführung - unmittelbar nach der TRANSFER-Anweisung - wird die dazu notwendige Statusinformation im Arbeitsbereich der rufenden Coroutine abgelegt. Die gerufene Coroutine wird entsprechend der abgelegten Statusinformation in ihrem Arbeitsbereich fortgeführt. Dies ist entweder die erste Anweisung nach der zuletzt ausgeführten TRANSFER-Anweisung oder aber bei einer erstmaligen Kontrollübernahme die erste Anweisung innerhalb der Coroutine. In MODULA-2 ist das Hauptprogramm als Coroutine anzusehen.

Da Coroutinen explizit mit einer TRANSFER-Anweisung gestartet werden, müssen sie auch durch solch eine TRANFER-Anweisung beendet werden. Wird bei der Programmausführung auf das Ende einer Coroutine gelaufen, so wird das ganze Programm mit einer Fehlermeldung abgebrochen.

Ein weiteres charakteristisches Merkmal in der Echtzeitprogrammierung ist die Reaktion auf unerwartete Ereignisse (Interrupt). Um in der MDOULA-2 Terminologie zu sprechen; der Punkt der Kontrollübergabe ist bei der Verarbeitung von Interrupts a priori nicht bekannt.

Mit Hilfe der Prozedur IOTRANSFER wird eine 'unprogrammierte' Kontrollübergabe (Interrupt) zwischen Coroutinen vorgenommen.

```
PROCEDURE IOTRANSFER
          (VAR from-processid, to-processid:
           PROCESS; iva: ADDRESS);
```

from-processid steht für den Bezeichner der Coroutine, die die Kontrolle abgibt.

to-processid steht für den Bezeichner der Coroutine, die die Kontrolle erhält.

iva steht für die Interrupt - Vektor-Adresse die einem Gerät zugeordnet ist.

Der Aufbau einer Interrupt-Service-Routine als Coroutine sieht dann folgendermaßen aus:

```
Procedure InterruptHandler;
...
Beginn  (* InterruptHandler *)
  Loop
    IOTRANSFER
        (from-processid, to-processid, iva);
      .
      .
      .
  END
END InterruptHandler .
```

Durch den Aufruf von IOTRANSFER wird die Startadresse des Interrupt-Handler (Geräte - Prozeß in MODULA-2 Terminologie) in die Interrupt-Vektor-Tabelle eingetragen. Danach wird die Kontrolle an die in der Parameterliste angegebene Coroutine übergeben. Löst nun ein Gerät einen Interrupt aus, so wird die gerade aktive Coroutine angehalten und die dem Interrupt zugeordnete Coroutine (Interrupt-Handler) nach der IOTRANSFER-Anweisung fortgeführt. Bedingung ist allerdings, daß die Geräte-Priorität höher ist als die der vor Auslösung des Interrupts aktiven Coroutine. Der Interrupt-Handler ist in einer Endlosschleife eingeschlossen, d.h. IOTRANSFER wird wieder ausgeführt. Offen bleibt nun ob die unterbrochene Coroutine weiter abgearbeitet wird, oder ob in die Coroutine gesprungen wird, deren Bezeichner (to-processid) in der Parameterliste von IOTRANSFER steht.

Um den Programmablauf zu verstehen ist es wichtig zu wissen, daß nur beim ersten Aufruf von IOTRANSFER zur Coroutine to-processid verzweigt wird. Bei allen weiteren Aufrufen von IOTRANSFER welche durch Interrupts initiiert werden, wird in die unterbrochene Coroutine zurückgekehrt.

2.3. Coroutinen-Konzept und Rechenprozeß-modell im Vergleich

Im folgenden werden virtuelle Maschinen bestehend aus den Komponenten "Speicher" und "Prozessor" betrachtet. Die Prozessoren der virtuellen Maschine können reale Prozessoren oder auch Rechenprozesse des unterliegenden Betriebssystems sein <Persch, et.al. 84>.

Für Einprozessorsysteme werden drei virtuelle Maschinen unterschieden:

(VM1) Synchrone Einprozessormaschine:

Alle Rechenprozeßaktivitäten werden in einer Coroutinenumgebung ausgeführt. Die Übergabe der Kontrolle wird nur durch explizite Synchronisationsanweisungen erreicht. (Bsp. aus PEARL: ACTIVATE, TERMINATE, SUSPEND, ...) .

(VM2) Zeitscheibengesteuerte Einprozessormaschine:

Die Rechenkernvergabe wird durch das Ende einer Zeitscheibe oder durch explizite Synchronisationsanweisungen angestossen.

(VM3) Asynchrone Einprozessormaschine:

Die Rechenkernvergabe wird durch asynchrone Ereignisse oder durch explizite Synchronisationsanweisungen angestossen.

Die Ereignisse können sein:

- Geräteunterbrechungen (Interrupts)
- Zeitgeberunterbrechungen bei Betriebssystemen mit Time-Share-Betrieb, d.h. (VM2) wird durch (VM3) mit erfaßt
- Aufruf von Funktionen des Betriebssystems (Traps) (Ein-/Ausgabe, Änderung der Priorität eines Rechenprozesses, warten auf ein bestimmtes Ereignis, ...)

Es findet also eine asynchrone Rechenprozeßverdrängung entsprechend den momentanen Prioritätsverhältnissen und den Stati der Rechenprozesse statt. Im allgemeinen weiß der Benutzer nicht, welcher Rechenprozeß gerade aktiv ist.

Unser Ziel ist also eine virtuelle Maschine vom Typ (VM3). Was wir in MODULA-2 vorfinden ist zunächst eine virtuelle Maschine vom Typ (VM1). Durch IOTRANSFER kann jedoch das Eintreffen eines Interrupts zum Synchronisationsereignis erklärt werden, an dem die Kontrolle an eine andere Coroutine abgegeben wird.

Ein (sequentieller) Rechenprozeß besteht aus privaten Daten und einem sequentiellen Programm. Ein paralleles Programm besteht aus mehreren sequentiellen Rechenprozessen, die zeitlich (quasi) parallel ablaufen können und sich gegenseitig in einer definierten Weise beeinflussen. Die Beeinflussungsmöglichkeiten bestehen im wesentlichen im geordneten Datenaustausch untereinander und in der Möglichkeit, den zeitlichen Ablauf zu beeinflussen <Hansen 77>.

Coroutinen sind ein Hilfsmittel für die Beschreibung und Modellierung der einem Problem inhärenten Parallelität durch Sequentialisierung. D.h. Coroutinen als Beschreibungsmittel für parallele Rechenprozesse eignen sich ausschließlich für Einprozessorsysteme, eine Einschränkung die im Rechenprozeßmodell nicht vorhanden ist.

Das Coroutinen-Konzept ist auf einer sehr niedrigen Ebene angesiedelt. Daraus ergibt sich eine gewisse Schwerfälligkeit in der Handhabung (Beispiel in <Dal Cin 84>, S. 282-287). Der Programmierer muß sich für jede Applikation die Synchronisation neu überlegen, auch wenn eine Synchronisation gar nicht notwendig wäre, weil die Rechenprozesse unabhängig sind. Jeder Kontrollwechsel muß explizit zur Programmerstellungszeit statisch angegeben werden. Beim Rechenprozeßmodell wird die "Verzahnung" der parallel ablauffähigen Rechenprozesse implizit durch das Betriebssystem vorgenommen, nur bei einer notwendigen Synchronisation (wechselseitiger Ausschluß, Kooperation) muß man Angaben machen. Dies ist ein wichtiger Punkt der bei der Synchronisation der Coroutinen beachtet werden muß, denn dadurch können sehr leicht "Deadlocks" programmiert werden.

Aufbauend auf dem Coroutinen-Konzept ist die Implementierung höherer Konzepte zur Verwaltung paralleler Rechenprozesse möglich. Die dazu erforderlichen Prozeduren müssen vom Programmierer selbst geschrieben werden (Beispiel in <Wirth 83>). Die Einschränkung des möglichen Gewinns durch Mehrprogrammbetrieb wird aber auch durch die höheren Konzepte nicht beseitigt, denn die explizite Kontrollübergabe bleibt bestehen. Durch die Verdrängungsmöglichkeit in (VM3) ergibt sich eine bessere Auslastung des Prozesses.

Prioritäten in MODULA-2 werden nicht im Sinne von "bevorzugt zu bedienen" verstanden. Die Angabe einer Priorität bei der Modul-Deklaration dient vielmehr zur Deklaration eines Monitors.
Dadurch kann man für Geräte-Rechenprozesse "wechselseitigen Ausschluß" garantieren. Es wird also die Interruptebene festgelegt, bis zu der (einschließlich) Interrupts mit niedriger Priorität ausgesperrt werden.

Da jetzt Interrupts auf gleicher Ebene sich nicht gegenseitig unterbrechen können, wurde als Ausweg die Prozedur LISTEN eingeführt (<Dal Cin 84; S. 282-287>, <Spector>). Durch den Aufruf LISTEN wird die Interruptebene einen Befehl lang auf 0 gesetzt, dadurch können auch Interrupts mit niedriger Priorität abgearbeitet werden.

Mit der Prozedur IOTRANSFER wird gewissermaßen die Zuordnung einer Interrupt-Quelle zu einer Interrupt-Service-Routine hergestellt (CONNECT-Funktion). Die komplementäre Prozedur (DISCONNECT-Funktion) gibt es nicht.

Wie bereits unter Kapitel 2.2 ausgeführt wurde, wird nur beim ersten Aufruf von IOTRANSFER in einem Interrupt-Handler zur Coroutine to-processid verzeigt, während bei allen nachfolgenden Aufrufen zur unterbrochenen Coroutine zurückgekehrt wird. Der dynamische Ablauf entspricht an dieser Stelle nicht der statischen Programmbeschreibung.

Dies erklärt sich aus der Tatsache, daß die Abbildung der Interrupt-Verarbeitung auf Coroutinen im Prinzip ein Verstoß gegen das Coroutinen-Konzept ist.

Die Nachteile des Coroutinen-Konzepts für Realzeitprogrammierung sind also

(N1) Beschränkung auf Einprozessorsysteme

(N2) Schwerfälligkeit in der Handhabung

(N3) Einschränkung des möglichen Gewinns bei Mehrprogrammbetrieb

(N4) Fehlende Möglichkeit der Vergabe von Prioritäten im Sinne von "bevorzugt zu bedienen"

(N5) Mangelnde Flexibilität bei der Interrupt-Verarbeitung

(N6) Diskrepanz bei der Interrupt-Verarbeitung zwischen Programmbeschreibung und Programmausführung

3. Verfügbare MODULA-2 (Cross-)Kompilierer

3.1. Eine Liste mit Adressen

MODULA-2 ist inzwischen für mehrere Prozessoren und Betriebssysteme verfügbar.

Eine Liste (Stand Oktober 1985) kann vom Institut für Systemorientierte Informatik der Universität der Bundeswehr bezogen werden. Diese Liste kann natürlich nicht vollständig sein, da ständig neue Implementierungen vorgestellt werden <Anderson 84>, <Vergleich 85>.

3.2. Das Programmpaket SMILER-2

SMILER-2 steht als Synonym für die drei Cross-Kompilierer-Systeme

- SMILER (PDP-11 / LSI-11)
- SMILERM (MC 6809)
- SMILERX (MC 68000)

Bei allen drei Cross-Kompilierern ist das Hauptprogramm SM2 und die Pässe PASS1, PASS2, PASS3, SYMPASS, LSTPASS und ERRPASS identisch. Entsprechend den Zielmaschinen unterscheiden sich die Pässe PASS4 und PASS5 (Kodegenerierung). Die Originalversion ist auf den CDC-Rechensystemen lauffähig.

Im folgenden wird nur die Version für den MC 68000 - bezeichnet als SMILERX - betrachtet.

Die Portierung auf unseren 32-Bit Rechner war auf Grund

- der unterschiedlichen Hardware
- den verschiedenen Betriebssystemen
- und der unterschiedlichen Mächtigkeit des implementierten PASCAL-Sprachumfangs

nicht ganz einfach. Besondere Probleme waren:

- Real-Zahlen Darstellung
- Abbildung des SET-Konstrukts
- Verwendung von mehr als 32 Bit in einem CDC-60 Bit Maschinenwort
- File-Handling

Auf Grund der gemachten Erfahrungen sollte die Wirtsmaschine für eine erfolgreiche Portierung folgende Forderungen erfüllen:

- minimale Maschinenwortlänge 32 Bit
- Hauptspeicherausbau > = 1 MByte
- Externspeicher > = 10 MByte
- PASCAL-Kompilierer sollte/muß getrennt übersetzbare Moduln unterstützen

4. Ersetzung der Coroutinen durch ein Betriebsystem mit Prozeßmodell

Um den Implementierungsaufwand niedrig zu halten, wird das Prozeßmodell mit Hilfe des auf dem Markt käuflich erhältlichen Realzeitbetriebssystemkern MTOS-68K[1] (Multi-Tasking/Multi-Processor-Operating-System für den Prozessor MC 68000) realisiert <User's Guide>, <Installations Guide>.

4.1. MTOS - Ein Realzeit-Betriebssystemkern

MTOS ist ein Realzeitbetriebssystemkern, der gedacht ist für den Einsatz in sogenannten "Embedded Systems" ist und für mehrere Prozessoren verfügbar:

- Intel 8086
- Intel 8080/85
- Motorola 6800
- Motorola 6809
- Motorola 68000

Wichtige Eigenschaften von MTOS-68K sind:

- Echtzeit-Multi-Tasking/Multi-Prozessor Betriebssystemkern
- Prozessorvergabe gemäß Prioritäten/ Round-Robin
- Interruptgesteuerte Ein-/Ausgabe
- ROM-fähig
- Ein-/Ausgabetreiber, standard- und benutzerdefiniert
- Dynamische Debugger
- Anwendungsschnittstelle im RAM

Diese Anwendungsschnittstelle wird im nächsten Punkt näher untersucht.

4.2. Die Schnittstellen

4.2.1. Anwendungs- und Systemschnittstelle in MTOS

Da MTOS ein ROM-fähiger Realzeitbetriebssystemkern ist, muß die Verbindung zwischen Kern und Anwendung (Anwenderrechenprozesse) über einen Schreib-/Lese-Speicher geschehen (Bild 3). Die Systemschnittstelle hat dabei die Form

```
Marke 1   DS x
Marke 2   DS y
   .
   .
   .
```

also eine Platzhalterfunktion.

Die Anwenderschnittstelle stellt ein Konfigurationsprogramm dar. Es ist bisher in MTOS nicht möglich Rechenprozesse dynamisch zu kreieren. Entsprechend muß der Anwender in einem Konfigurationslauf die Anzahl der Rechenprozesse, die Anzahl der Gerätetreiber, die dazugehörigen Startadressen, usw. angeben. Mit Hilfe von Makros werden dann die entsprechenden Kontrollstrukturen aufgebaut.

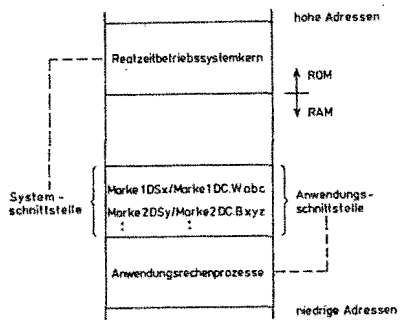

Bild 3: Schnittstelle zwischen dem MTOS-Kern und den Anwendungsrechenprozessen

---

[1] MTOS-68K ist ein eingetragenes Warenzeichen der Industrial Programming Inc.

Das uns interessierende Teilergebnis eines Konfigurationslaufes hat die Form

```
Marke 1   DC.W abc
Marke 2   DC.B xyz
   .
   .
   .
```

Diese beiden Bereiche müssen adressgleich im Speicher liegen. So werden also die Anwendungsparameter dem Systemkern übergeben.

4.2.2. Das Modul SYSTEMX im SMILER

Im Pseudo Modul SYSTEM sind die zielprozessor abhängigen Dinge für die "Low-Level-Facilities", die Routinen zur Unterstützung des Kompilierers und die Laufzeitroutinen enthalten. Im original MODULA-2 Kompilierer <Wirth 83> muß dieses Modul für jeden Übersetzungslauf dem Kompilierer zur Verfügung stehen. Durch die Transkription von MODULA-2 nach PASCAL gab es da Probleme. Die Lösung sah dann so aus: In Pass 1 werden die reservierten Worte für das Pseudo Modul SYSTEM ins Namensbuch des Kompilierers eingetragen und die Realisierung wird im Modul SYSTEMX vorgenommen (X für 68000). Das Modul SYSTEMX wird zur Bindezeit wie ein normales getrennt übersetzbares Modul dazugebunden. Durch diese Lösung hat man hier die Möglichkeit in das Laufzeitsystem (NEWPROCESS, TRANSFER, IOTRANSFER) von MODULA-2 einzugreifen, ohne daß man den ganzen Kompilierer neu übersetzen muß.

4.3. Realisierung

Zunächst könnte man im Modul SYSTEMX alle Kontrollstrukturen für MTOS entsprechend einer MODULA-2 Applikation aufbauen und die Anwendungsschnittstelle entsprechend ausfüllen. Diese Lösung ist wohl sehr elegant aber dafür sehr aufwendig. Da es nicht unser ursprüngliches Ziel war Coroutinen durch ein Prozeßmodell zu ersetzen, entschieden wir uns für folgende Lösung.

Es wird in MTOS eine Dummy Applikation implementiert (z.B. 10 Rechenprozesse und 2 oder 3 Standard-Gerätetreiber). Benutzerspezifische Treiber müssen allerdings vorher in MTOS integriert werden. Da der Benutzer die Startadresse der Schnittstelle (Kern-Anwendung) in MTOS selbst festlegen muß, ist die Startadresse bekannt, und somit die Adressen der Rechenprozesskontrollblöcke (Startadresse des Rechenprozesses) und die Stelle wo die aktuelle Anzahl der Rechenprozesse stehen muß.

Beim Aufruf von NEWPROCESS zur Laufzeit, wird die Bezugsadresse des Arbeitsspeichers einer Variable vom Typ PROCESS zugewiesen. In diesem Arbeitsspeicher steht unter anderem auch die Startadresse der Prozedur, die zum Rechenprozeß umgewandelt wurde. Die Reservierung des Speicherplatzes und die Vorbelegung mit den Startwerten wurde bereits zur Übersetzungs- und Bindezeit erledigt.

Die Anzahl der Aufrufe von NEWPROCESS entspricht der aktuellen Anzahl der Rechenprozesse. NEWPROCESS muß so abgeändert werden, daß bei einem Aufruf dem Prozeßdeskriptor eine MTOS-Rechenprozeßnummer zugeordnet wird (Tabelle) und die Startadresse des nächsten freien Dummy-Rechenprozesses durch die Startadresse des MODULA-2-Rechenprozesses ersetzt wird.

Mit den beiden Zugriffsfunktionen Gib-Rechenprozeßnummer(Prozeßname) und Gib-Prozeßname(Rechenprozeßnummer) wird die Konsistenz der Tabelle gewährleistet. Daß bei einer tatsächlichen Anwendung nicht alle Dummy-Rechenprozesse einen entsprechenden MODULA-2 Prozeß zugeordnet bekommen, hat keine weiteren Auswirkungen. Die restlichen Dummy-Rechenprozesse sind im Zustand dormant.

Die Prozedur IOTRANSFER wird gestrichen, denn es darf kein Interrupt am Betriebssystem "vorbei" abgearbeitet werden <Wirth 83>. TRANSFER entfällt ebenfalls und wird durch entsprechende MTOS-Dienste ersetzt (Tabelle 1).

MTOS-68K USER'S GUIDE

APPENDIX 1: SUMMARY OF MTOS SUPERVISOR SERVICES

SVC	SDB MACRO	FUNCTION	DESCRIPTION
1	STRSDB	START	start task
2	GIASDB	GETIA	get initial argument
3	TRMSDB	TRMT	terminate task without timed restart
4	TRRSDB	TRMR	terminate task with timed restart
5	CPRSDB	CPTY	change task priority
6	PAUSDB	PAUSE	pause for given interval
7	CANSDB	RESUME	resume (cancel pause of) another task
8	EIOSDB	EIO	reset/immediately set/test event flags
9	LRSSDB	LRS	*reset/immediately set EFs of given task
$A	CEFSDB	CEF	*copy event flags
$B	SEFSDB	SEF	set event flags after given time
$C	WEFSDB	WEF	wait until event flag are set
$D	MBSSDB	MBS	send a message to a mailbox
$E	MBRSDB	MBR	receive a message from a mailbox
$F	ALOSDB	ALOC	allocate a block of memory from a pool
$10	DALSDB	DALOC	return (de-allocate) memory to a pool
$11	STISDB	STIME	set ASCII date and time
$12	GTISDB	TIME	get (read) ASCII date and time
$13	SYNSDB	SYN	synchronize to ASCII ("wall") time
$14	DIOSDB	DIO	perform discrete I/O
$15	PIOSDB	PIO	perform peripheral I/O
	RSVSDB	RESERVE	reserve unit
	RLSSDB	RELEASE	release unit
	REDSDB	READ	read with default prompt
	WRISDB	WRITE	write
	REPSDB	READPT	read with given prompt
	ROCSDB	READ1C	read one character w/o prompt
$16	RSFSDB	RSF	release a semaphore
$17	WSFSDB	WSF	wait for $F to be free
$18	.	.	[spare]
$19	MIOSDB	MIO	* perform (local) memory I/O
$1A	CITSDB	CIT	* convert task number to TCB address
$1B	CTISDB	CTI	* convert TCB address to task number
$1C	.	.	** perform Debugger support
$1D	CDISDB	CDI	* copy discretes

* = introduced for Debugger, but may be used by any task
** = may be used ONLY by Debugger tasks

Tabelle 1: MTOS - Dienste aus <User's Guide>

Es ist nun zu klären, wie MTOS-Dienste in MODULA-2-Programmen benutzt werden können.

Wie heute allgemein üblich, werden auch in MTOS die Systemdienste mit Hilfe eines TRAPS aufgerufen. Die Adresse des Parameterblokkes, der genaue Spezifikationen des Dienstes und den Rückkehrstatus enthält, wird im Stack übergeben.

Die Ausführung eines TRAPS kann mit Hilfe der im Cross-Kompilierer schon vorhandenen Prozedur SYSCALL erledigt werden.

```
FROM SYSTEM IMPORT SYSCALL;
SYSCALL (Service: CARDINAL;
         StartAdrParaBlock: ADDRESS;
         VAR Reply: CARDINAL);
```

Die Aufbereitung des Parameterblockes muß in

dienstorientierten Prozeduren erledigt werden.

Einige Details bleiben noch zu klären.

Der Arbeitsspeicher (Workspace) eines Rechenprozesses (Bild 4) muß nach wie vor angelegt werden. Denn zum einen verwaltet der Kompilierer selbst Informationen im Arbeitsspeicher (Dijkstra Display's, statischer Link für Zugriff auf globale Variable aus einer Prozedur) und zum anderen benötigen wir Informationen (Initialwerte der Register) für den Anschluß an MTOS.

```
(*  THE STRUCTURE OF A PROCESS WORKSPACE :                          *)
(*  ------------------------------------                           *)
(*                                                                  *)
(*  THE VALUE OF A VARIABLE OF TYPE 'PROCESS' (CALLED               *)
(*  PROCESS-VARIABLE) IS A POINTER TO THE REFERENCE                 *)
(*  LOCATION INDICATED BY --->>> (PD).                              *)
(*  ALL OFFSETS (IN BYTES) RELATIVE TO THE PROCESS BASE             *)
(*  --->>> OF THISPD.                                               *)
(*  THE PICTURE SHOWS A PROCESS WORKSPACE GENERATED                 *)
(*  AT NEWPROCESS TIME.                                             *)
(*                                                                  *)
(*    HIGH MEMORY : LWA + 1 OF WSP IS AT 52(PD).                    *)
(*                                                                  *)
(*                                                                  *)
(*         52(PD) LWA + 1 OF WSP                                    *)
(*                ==========================                        *)
(*         48(PD) I  RESERVE                I   1 LONGWORD         *)
(*         44(PD) I      ...                I   1 LONGWORD         *)
(*         40(PD) I      ...                I   1 LONGWORD         *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*         16(PD) I  INTERRUPTCODE          I   8 LONGWORDS        *)
(*         12(PD) I  FIRSTELEMENTSIZE       I   1 LONGWORD         *)
(*          8(PD) I  FIRSTFREEELEMENT       I   1 LONGWORD         *)
(*          4(PD) I  STACKLIMIT = FWA WSP   I   1 LONGWORD         *)
(*    --->>> (PD) I  PROCESS BASE THISPD    I   1 LONGWORD         *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*        -36(PD) I  DIJKSTRA DISPLAY'S     I   9 LONGWORDS        *)
(*        -40(PD) I  INITIAL A7 = SP        I                      *)
(*        -44(PD) I  INITIAL A6 = MP        I                      *)
(*        -48(PD) I  INITIAL A5 = PD        I                      *)
(*        -52(PD) I  INITIAL A4             I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I                      *)
(*                I      ...                I   TOTAL 16           *)
(*                I      ...                I   LONGWORDS          *)
(*                I      ...                I   FOR ALL            *)
(*       -100(PD) I  REGISTERS D0 - A7       I   REGISTERS         *)
(*       -104(PD) I  PROCESS TERMINATES      I   1 LONGWORD        *)
(*      FOPOS(PD) I  FORMAT/OFFSET(MC68010)  I   1 WORD            *)
(*      PCPOS(PD) I  INITIAL PC              I   1 LONGWORD        *)
(*      SRPOS(PD) I  INITIAL SR              I   1 WORD            *)
(*                I                          I                     *)
(*                I                          I                     *)
(*                I                          I                     *)
(*                I                          I                     *)
(*         FWA WSP I                          I                    *)
(*                ==========================                        *)
(*                                                                  *)
(*                                                                  *)
(*    LOW MEMORY  : THE VALUE OF STACKLIMIT IS                      *)
(*                    = FWA OF WSP.                                 *)
```

Bild 4: Struktur des Arbeitsspeichers einer Coroutine

In SYSTEMX haben drei Adressregister eine feste Bedeutung

A7 = SP aktiver Stack Pointer

A6 = MP Mark Pointer (Stack Marker)
Zeiger auf die lokalen Variablen und Parameter einer Coroutine (bzw. Rechenprozess)

A5 = PD Process Description
Zeiger auf den aktiven Prozeßbezeichner

Diese Register müssen bei jedem Rechenprozesswechsel gerettet werden. Dies wird in MTOS durch Retten aller Register erledigt.

Ein Sonderfall ist noch zu klären. In MODULA-2 wird das Hauptprogramm wie eine Coroutine behandelt. Analog gibt es in MTOS einen INITIAL-Rechenprozeß (Rechenprozeß Nummer 0) der vom MTOS-System gestartet wird und der alle anderen Rechenprozesse startet. Die Verbindung zwischen MTOS-INITIAL-Rechenprozeß und MODULA-2 MAIN-Programm muß im Modul-Rumpf von SYSTEMX hergestellt werden.

5. Stand und Ausblick

Die Portierung vom SMILERX auf unsere Host - Maschine (PE 3240) ist erfolgreich abgeschlossen <Bosse, Heine, Kotschi 84>, <Westenkirchner 85>. Ein Bibliotheksverwaltungsverwaltungsprogramm <Hafner 85> ist ebenfalls verfügbar.

Nach der vorgestellten Analyse scheint die Ersetzung des Coroutinen-Konzepts durch ein Prozeßmodell in Form einer Diplomarbeit durchführbar. Die Arbeiten könnten bis Ende 1986 abgeschlossen sein.

Da in MTOS die Parameterstruktur nicht einheitlich ist, scheint es nicht sinnvoll die verschiedenen Parameterblockstrukturen als Typen zu vereinbaren und dann zu exportieren. Vielmehr wollen wir einen Satz von parametrisierten Prozeduren erarbeiten und diese in Form von einem getrennt übersetzbaren Modul (Definition- und Implementation-Part) bereitstellen.

Bei der Entwicklung des Definition-Parts soll nicht nur der vorhandene Echtzeitbetriebssystemkern betrachtet werden. Ziel ist vielmehr zu versuchen, die Prozedurvereinbarungen so universell zu gestalten, daß auch andere Betriebssysteme duch einen entsprechenden Implementation-Part angeschlossen werden können.

In diesem Zusammenhang sollen insbesondere die nachfolgenden Entwicklungen betrachtet werden:

(E1) RTOS (realtime operating system) vom EWICS TC 8

(E2) MODEB (MODULA-2 Echtzeitbetriebssystem) basierend auf dem Vorschlag des EWICS TC 8

(E3) MOSI (Microprocessor Operating Systems Interface) vom IEEE 825, Revision 4.0

Jeder, der schon größere Programmpakete portiert hat, weiß, daß die unterschiedlichen Betriebssystemschnittstellen und -dienste große Hindernisse bei der Portierung darstellen. MODULA-2 bietet hier die Möglichkeit durch einen Satz von Prozedurdefinitionen (Definition-Part) eine Anpassung an die unterschiedlichen Betriebssysteme zu ermöglichen. Hierdurch könnte eine Vereinheitlichung geschaffen werden, ohne daß vorhandene Betriebssysteme geändert werden müssen.

6. Literatur

<Anderson 84>

Anderson, T.L.: Seven MODULA-2 Compilers Reviewed.
Journal of PASCAL, ADA & MODULA-2, März/April 84

<Andrews, Schneider 83>

Andrew, G.R.; Schneider, F.B.: Concepts and Notation for Concurrent Programming.
Computing Surveys, Vol. 15, No. 1, March 83

<Bosse, Heine, Kotschi 84>

Bosse, J.; Heine, P.; Kotschi, R.: Portierung eines MODULA-2 Cross-Systems auf dem Prozeßrechner PERKIN ELMER 3240.
Diplomarbeit an der Hochschule der Bundeswehr München, Fachbereich Informatik, 1984

<Conway 1963>

Conway, M.E.: Design of a separable transition-diagram comipler.
Communication of the ACM 6, 7 (July 1963), pp 396-408

<Dal Cin, Lutz, Risse 84>

Dal Cin, M.; Lutz, J.; Risse, T.:
Programmierung in MODULA-2.
Teubner Verlag, Stuttgart, 1984

<Gutknecht 84>

Gutknecht, J.: Tutorial on MODULA-2.
Byte, August 1984, pp 157-176

<Hafner 85>

Hafner, U.: Portierung und Erweiterung eines Bibliotheksverwaltungsprogramms für ein MODULA-2 Programmiersystem.
Diplomarbeit an der Universität der Bundeswehr München, Fakultät für Informatik, 1985

<Hansen 77>

Hansen, P.B.: The Architecture of Concurrent Programs.
Prentice-Hall, INC. Englewood Cliffs, New Jersey 07632

<Installation Guide>

MTOS-68K Installation Guide.
Industrial Programming, Inc. Jericho, New York 11755

<Levi 81>

Levi, P.: Betriebssysteme für Realzeitanwendungen.
Datakontext-Verlag, Köln 1981

<McCormack, Gleaves 83>

McCormack, J.; Gleaves, R.: MODULA-2 - A Worthy Successor to PASCAL.
Byte 1983, pp 385-395

<Mitchell, Mayburg, Sweet 78>

Mitchell, J.G.; Mayburg, W.; Sweet, R.: MESA Language Manual.
Report CSL-78-1, Xerox, Palo Alto, California, 1978

<Obenhuber, Rzehak 83>

Obenhuber, H.; Rzehak, H.: Die Anpassung eines Mikroprozessorsystems an den CAMAC-Standard zum Aufbau eines hierarchischen lokalen Rechnernetzes.
Bericht Nr. 8309 der Hochschule der Bundeswehr München, Fachbereich Informatik, Oktober 1983

<Ohran 84>

Ohran, R.: LILITH and MODULA-2.
Byte, August 1984, pp 181-192

<Paul 84>

Paul, R.J.: An Introduction to MODULA-2.
Byte, August 1984, pp 195-210

<Persch, et.al. 84>

Persch, G.; Jansohn, H.-S.; Landwehr, R.; Uhl, J.; Dausmann, M.; Kirchgässner, W.:
A Protable Ada Tasking System for Single Processors
German Chapter of the ACM - Berichte, Teubner Verlag, Stuttgart, 1984

<Pomberger 84>

Pomberger, G.: Softwaretechnik und MODULA-2.
Hauser-Verlag, München, Wien, 1984

<Spector>

Spector, D.: Ambiguities and Insecurities in MODULA-2.
MS 108-17-3, Prime Computer, Inc. 500 Old Connecticut Path Framingham, Massachusetts 01701

<Summer, Gleaves>

Summer, R.T.; Gleaves, R.E.: MODULA-2 - A Solution to PASCAL's Problems.
Volition Systems, P.O. Box 1236, Del Mar, CA 92014

<User's Guide>

MTOS-68K User's Guide.
Industrial Programming, Inc. Jericho, New York 11753

<Vergleich 85>

Drei MODULA-Compiler im Vergleich.
Computer Persönlich, Ausgabe 14, 26.6.85, pp 50-54

<Westenkrichner 85>

Westenkrichner, H.: Überarbeiten der Ein-/Ausgabe und testen spezieller Sprachkonstrukte eines MODULA-2 Cross-Kompilierers.
Diplomarbeit an der Universität der Bundeswehr, Fakultät für Informatik, Institut für Systemorientierte Informatik, 1985

<Wirth 77>

Wirth, N.: "MODULA: A Language for Modular Multiprogramming Language."
Software-Practice and Experience, Vol. 7, pp 3-35, 1977

<Wirth 83>

Wirth, N.: Programming in MODULA-2.
Second Edition, Springer-Verlag, Berlin, Heidelberg, New York 1983

<Wirth 84>

Wirth, N.: History and Goals of MODULA-2.
Byte, August 1984, pp 145-152

Verfasser:

Jürgen Stoll
Universität der Bundeswehr München
Fakultät für Informatik
Institut für Systemorientierte Informatik
Werner-Heisenberg-Weg 39
D-8014 Neubiberg
Tel. (089) 6004-2404