Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

SharkOberon

Entwicklung eines Oberon System 3 für den DNARD Network Computer

Diplomarbeit
am Institut für Computersysteme
der ETH Zürich

bei
Prof. J. Gutknecht
Assistent: P. Muller

vorgelegt von Andreas Signer
Brüttisellen, im März 1999

Abstract

Dieser Bericht beschreibt die Implementierung von SharkOberon, einer Portierung des Oberon System 3 auf den DNARD Network Computer. Als Grundlage für das System wurde der Quellcode von Native Oberon V2.3.2 verwendet. Der Compiler basiert auf dem Oberon-1-Compiler für den NS32000. SharkOberon wird unter Verwendung von DHCP oder TFPT vom Netzwerk gebootet und verwendet als Massenspeicher eine RAM-Disk. Neben einer Übersicht über die Hardware und Firmware der Zielplattform werden einige Aspekte der Implementierung des Compilers und des Systems detailliert erläutert. Auf eine umfangreiche Beschreibung von Oberon System 3 sowie des Oberon-Compilers wird dabei verzichtet, da dazu ausreichend Literatur existiert.

Inhaltsverzeichnis

Abstract iii

1 Einleitung 1

2 Der DNARD Network Computer 3
2.1 Der StrongARM SA-110 . . . 3
2.1.1 Die Prozessormodi . . . 3
2.1.2 Datentypen . . . 4
2.1.3 Register . . . 4
2.1.4 Der Instruktionssatz . . . 5
2.2 Open Firmware . . . 7

3 Der Compiler 9
3.1 Struktur des Compilers . . . 10
3.2 Repräsentation eines Moduls im Speicher . . . 11
3.3 Codeerzeugung . . . 11
3.3.1 Verwendung der Register . . . 11
3.3.2 Adressierung und Alignment von Variablen und Parametern 11
3.3.3 Laden von Konstanten . . . 12
3.3.4 Das `FOR`-Statement . . . 15
3.3.5 Aufruf von Prozeduren und Parameterübergabe . . . 20
3.3.6 Prolog und Epilog . . . 21
3.3.7 Das Procedure Activation Frame . . . 21
3.3.8 `DIV` und `MOD` . . . 21
3.3.9 Gleitkommaunterstützung . . . 24
3.4 Der integrierte Assembler . . . 25
3.4.1 Verwendung des Assemblers . . . 25
3.4.2 Implementation des Assemblers . . . 25

4 Das System 29
4.1 Laden des Boot Images . . . 29
4.2 Verwaltung des physikalischen Speichers . . . 30
4.3 Verwendung des virtuellen Adressraums . . . 30
4.3.1 Die Seite 0 . . . 30
4.3.2 Der Modul-Bereich . . . 32
4.3.3 Der Heap-Bereich . . . 32
4.4 Initialisierung des Systems . . . 32

4.4.1 Initialisierung der RAM-Disk 33
4.4.2 Implementation der RAM-Disk 33
4.4.3 Netzwerkunterstützung bei der System-Initialisierung . . 34
4.5 Struktur des Inner Core 36
4.6 Garbage Collection und Type Descriptors 38
4.6.1 Type Descriptor 38
4.6.2 Array Block 39
4.7 Änderungen im Netzwerk-Subsystem 39
4.8 Konfigurationsvariablen 40
4.9 Gleitkommaunterstützung 41

5 Performance-Messungen 43
5.1 Dhrystone Benchmark 43
5.2 Hennessy Benchmark 45
5.3 Display Benchmark 45

6 Notwendige Änderungen am Native Quellcode 49

7 Adaption des Systems an eine 2-Tasten-Maus 51
7.1 Die bestehende Lösung 51
7.2 Das neue Verfahren und seine Implementation 51
7.3 Kritik 52

8 Schlussbemerkungen und Ausblick 55

A Cross-Development-Tools 57
A.1 Compiler, Decoder und Browser 57
A.2 Boot Linker 57
A.3 Automatische Generierung von `RAMDisk.Content` 57

B Syntax von Symboldateien 59

C Syntax von Objektdateien 61

D Das Modul SYSTEM 63

E Konfigurationsvariablen 65

F Portierte Module 67

Abbildungsverzeichnis

2.1 Die Register des SA-110 5

3.1 Struktur des Compilers 10
3.2 Repräsentation eines Moduls im Speicher 12
3.3 Programmfragment „Laden von Konstanten“ 14
3.4 Programmfragment „Leeren des Konstantenbuffers“ 17
3.5 Beispiel „`FOR i:=0 TO j DO ... END`“ 18
3.6 Beispiel „Unstrukturierter Kontrollfluss aufgrund von `EXIT`“ . . . 19
3.7 Programmfragment: „Codeerzeugung für `EXIT`“ 20
3.8 Beispiel „Branch über eine Distanz > 32 MByte“ 20
3.9 Beispiel „Prolog und Epilog einer Prozedur“ 21
3.10 Das Procedure Activation Frame 22
3.11 Beispiel „Verwendung von `DIV` und `MOD`“ 23
3.12 Beispiel „Verwendung des Assemblers“ 26
3.13 Beispiel „Verwendung von `ADDR`“ 26
3.14 Programmfragment „Kern des Assemblers“ 27

4.1 Verwendung des virtuellen Adressraums 31
4.2 Programmfragment „Start der Netzwerkdienste zur RAM-Disk-Initialisierung“ 35
4.3 Module des Inner Core 37
4.4 Struktur des Type Descriptors 38
4.5 Struktur des Array Blocks 39
4.6 Programmfragment „Auslesen der Netzwerkeinstellugnen“ 40
4.7 Beispiel einer NetSystem.Hosts.HostNames-Sektion 41
4.8 Das Interface von SoftFloat 42

5.1 Dhrystone Benchmark 44
5.2 Hennessy Benchmark (ohne FFT) 46
5.3 Display Benchmark 47

7.1 Programmfragment „Emulation der mittleren Maustaste“ 53

A.1 Beispiel eines ARMCompiler.Prefix-Eintrags 57
A.2 Beispiel eines `ContentMaker`-Scripts 58

Tabellenverzeichnis

2.1 Datentypen des SA-110 4

3.1 Verwendung der Register 12
3.2 Adressierung von Variablen und Parametern 13
3.3 Übersicht über eingefügte Konstantenblöcke ausgewählter Module. 16
3.4 Vom Compiler aufgerufene Prozeduren in „SoftFloat“ 24

4.1 Prozeduren zur Verwaltung des physikalischen Speichers 30

5.1 Hennessy-Benchmark (nur FFT) 45

6.1 bei der Portierung notwendige Änderungen an portablen Modulen 50

E.1 Konfigurationsvariablen von SharkOberon 66

Kapitel 1

Einleitung

Aufgabenstellung

Das Ziel der Diplomarbeit war die Portierung der Basis eines Oberon System 3 auf den DNARD Network Computer (im folgenden kurz NC genannt). Das System soll die Entwicklung von Oberon-Programmen unterstützen sowie grundlegende Internet-Tools zur Verfügung stellen. Das Aufstarten soll via Standard-Internet-Protokollen (DHCP, TFPT, etc.) erfolgen. Als lokaler Massenspeicher soll eine RAM-Disk eingesetzt werden.

Das Projekt wurde in folgende Arbeitsschritte aufgeteilt:

1. Entwicklung von Cross-Development-Tools unter Native Oberon
2. Portierung der Native Oberon Kernel-Module
3. Entwicklung der Netzwerk-Tools
4. Portierung der Cross-Development-Tools

Optional sollte noch Gleitkommaunterstützung implementiert werden sowie eine Adaption des Systems an eine 2-Tasten-Maus durchgeführt werden.

Übersicht

Im Kapitel 2 werden die Hardware und Firmware der Zielmaschine vorgestellt. Danach folgt eine Beschreibung des Compilers. Kapitel 4 enthält eine Beschreibung des eigentlichen Systems. Dabei werden jedoch nicht alle Einzelheiten von Oberon System 3 erläutert, sondern nur auf einzelne Implementationsdetails eingegangen, die zum Verständnis und zur Erweiterung nötig sind. Zum Schluss wird in Kapitel 6 eine Übersicht über die notwendigen Anpassungen eigentlich portabler Module gegeben sowie in Kapitel 7 eine mögliche Adaption des Systems an eine 2-Tasten-Maus vorgestellt.

Kapitel 2

Der DNARD Network Computer

Das DIGITAL Network Appliance Reference Design (DNARD) besteht im grossen und ganzen aus folgenden Komponenten:

- StrongARM SA-110 CPU [Dig96]
- 32 MByte RAM
- IGS CyberPro 2010 Graphics Adapter [IGS97]
- Crystal Semiconductor CS8900 Ethernet Controller [Cry95]
- American Megatrends Keyboard Controller [Ame94]
- ESS ES1887 Soundchip [ESS97]

Auf der Softwareseite kommt als Firmware eine Open Firmware-Implementation [Fir98, Fir97] der Firma FirmWorks hinzu. Nachfolgend werden nur die CPU und die Firmware genauer betrachtet, da diese als einzige Komponente des DNARDs nicht IBM-PC-kompatibel ist. Für eine genaue Beschreibung der anderen Bausteine sei auf die Literatur verwiesen.

2.1 Der StrongARM SA-110

Der StrongARM SA-110 ist ein 32bit-RISC-Prozessor mit einer load-/store-Architektur. Er besitzt eine 5-stufige Integer-Pipeline und jeweils 16 KBytes Daten- und Instruktionscache. Ausserdem enthält er eine integrierte MMU und kann entweder *little endian* oder *big endian* betrieben werden. Jede Instruktion umfasst 4 Bytes und ist bedingt ausführbar.

2.1.1 Die Prozessormodi

Der SA-110 verfügt über einen *User Mode* sowie über mehrere privilegierte *Supervisor Modes*. Diese werden wie folgt verwendet:

IRQ Der IRQ-Mode wird aktiviert, wenn ein Interrupt Request (IRQ) auftritt.

FIQ Der FIQ-Mode wird aktiviert, wenn ein Fast Interrupt Request (FIQ) auftritt.

Svc Der Svc-Mode wird durch die Ausführung eines Software Interrupts (`SWI`-Instruktion) aktiviert.

Undef Der Undef-Mode wird aktiviert, wenn eine undefinierte Instruktion ausgeführt werden soll.

Abt Der Abt-Mode wird durch die MMU aktiviert, wenn auf eine ungültige Adresse zugegriffen wird. Erfolgt der Zugriff durch den Fetch-Zyklus (d.h. der *program counter* zeigt auf eine ungültige Adresse), so handelt es sich um einen *prefetch abort*, ansonsten um einen *data abort*.

2.1.2 Datentypen

Die vom SA-110 unterstützten Datentypen sowie ihre Grösse in Bytes sind in Tabelle 2.1 aufgeführt.

Datentyp	Grösse
Byte	1
Halfword	2
Word	4

Tabelle 2.1: Datentypen des SA-110

2.1.3 Register

Der SA-110 verfügt über 30 allgemeine Register, 6 Statusregister sowie einen PC (*program counter*), wobei nicht alle Register ständig sichtbar sind. Je nach Prozessormodus sind jeweils 15 allgemeine Register R0 - R14, ein oder zwei Statusregister sowie der PC sichtbar. Die Register sind in teilweise überlappenden Bänken mit verschiedenen Registern für jeden Prozessormodus angeordnet. Der Abbildung 2.1 auf der nächsten Seite ist zu entnehmen, welche Register in welchem Modus sichtbar sind.

Die Register R0 – R12 sind immer allgemein verwendbar. R13 und R14 können ebenfalls allgemein verwendet werden, haben aber ausserdem noch spezielle Rollen:

R13 R13 (der *stack pointer* oder SP) ist in allen Prozessormodi *banked*, um einen privaten Stack Pointer zur Verfügung zu stellen.

R14 In R14 (das *link register* oder LR) wird die Rücksprungadresse von Subroutinen gespeichert. Das LR ist ebenfalls in allen Modi *banked*.

Usr	Svc	Abort	Undef	IRQ	FIQ
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABT	R13_UND	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABT	R13_UND	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABT	SPSR_UND	SPSR_IRQ	SPSR_FIQ

Abbildung 2.1: Die Register des SA-110

Auf das Statusregister CPSR kann in allen Prozessormodi zugegriffen werden. Es enthält die *Condition Code Flags*, die *Interrupt Enable Flags* sowie den aktuellen Prozessormodus. In nicht privilegierten Modi können nur die *Condition Code Flags* verändert werden.

2.1.4 Der Instruktionssatz

Der Instruktionssatz lässt sich grob in folgende Gruppen unterteilen:

- Data Processing
- Single Data Transfer
- Halfword Data Transfer
- Block Data Transfer
- Multiply und Multiply-and-add

- Sprungbefehle
- Software Interrupt
- Instruktionen zur Koprozessorsteuerung
- Instruktion zur Manipulation der Statusregister

In den folgenden Abschnitten werden die einzelnen Gruppen kurz erläutert. Es wird dabei jedoch nicht auf Software Interrupts sowie die Instruktionen zur Koprozessorsteuerung und Statusregistermanipulationen eingegangen, da diese nicht für das Verständnis des vom Compiler erzeugten Codes notwendig sind. Eine detaillierte Beschreibung aller Instruktionen findet sich in [Adv96, Dig96].

Data Processing

Unter *Data Processing* werden Vergleiche, arithmetische Operationen (mit Ausnahme der Multiplikation) und Bitoperationen zusammengefasst. Ausserdem gehört eine `MOV`- Instruktion dazu, die „kleine“ Konstanten in ein Register lädt. Jede Operation hat ein Zielregister sowie zwei Operanden. Der 1. Operand ist dabei immer ein Register, der 2. Operand kann ein konstanter Wert sein (8 Bit vorzeichenlos, kann um eine gerade Anzahl Bits rotiert werden), oder ein Register, auf welches zusätzlich eine Schiebeoperation angewendet werden kann (wobei der Operand für die Schiebeoperation konstant ist oder aus einem Register stammt). Bei Vergleichsoperationen wird das Zielregister ignoriert, bei einer `MOV`-Instruktion wird der 1. Operand ignoriert. Soll eine Instruktion die *Condition Code Flags* beeinflussen, so muss dies explizit angegeben werden.

Single Data Transfer

Zu den *Single Data Transfer* Instruktionen gehören eine load-Instruktion `LDR` sowie eine store-Instruktion `STR`. Die Adresse des zu lesenden/schreibenden Datums wird register-indirekt gebildet. Der Offset zur Basisadresse stammt entweder aus einem Register (auf welches ebenfalls eine Schiebeoperation angewendet werden kann) oder ist eine Konstante k mit $0 \leq k \leq 4095$. In der Instruktion muss ausserdem angegeben werden, ob der Offset addiert oder subtrahiert werden soll.

Es besteht die Möglichkeit eines *Base Register Writebacks*. Dabei wird die effektive Adresse in das Basisregister zurückgeschrieben. Wird *Base Register Writeback* verwendet, so kann ausserdem zwischen *preindexed* und *postindexed* gewählt werden. Bei einem *preindexed* load oder store wird der Offset vor dem Lesen/Schreiben addiert, während bei einem *postindexed* load/store der Offset erst nach dem Lesen/Schreiben addiert wird.

`LDR` und `STR` verarbeiten entweder Words oder Bytes. Bei Words muss die Adresse ein Vielfaches von 4 sein. Bytes werden in die Bits 0 – 7 des Registers geladen, die Bits 8 – 31 werden mit 0 gefüllt. Soll eine Vorzeichenerweiterung durchgeführt werden, so muss ein *Halfword Data Transfer* verwendet werden.

Halfword Data Transfer

Zu den *Halfword Data Transfer* Instruktionen gehören wie beim *Single Data Transfer* eine load-Instruktion `LDRH` sowie eine store-Instruktion `STRH`. Die Adresse des zu lesenden/schreibenden Datums wird ebenfalls register-indirekt gebildet, wobei der Offset aber nicht skaliert werden kann. Bei `LDRH` kann der gelesene Wert (ein Byte oder ein Halfword) vorzeichenerweitert werden.

Block Data Transfer

Mittels *Block Data Transfer* Instruktionen lassen sich mehrere Register gleichzeitig (d.h. in einer Instruktion) lesen oder schreiben. Die load- und store-Instruktionen `LDM` und `STM` haben zwei Argumente: ein Basisregister, in welchem die Adresse steht, ab der gelesen bzw. geschrieben werden soll, sowie eine Liste der zu lesenden/schreibenden Register. Es besteht wie bei *Single Data Transfer* Instruktionen die Möglichkeit eines *Base Register Writebacks*. Die Änderung der Adresse im Basisregister kann ebenfalls spezifiziert werden. Dem Programmierer stehen folgende Möglichkeiten zur Verfügung:

- increment before
- increment after
- decrement before
- decrement after

Multiply und Multiply-and-add

Multiply und Multiply-and-add arbeiten nur mit Registern. Beide Instruktionen enthalten ein Zielregister sowie 2 (multiply) oder 3 (multiply-and-add) Operanden.

Sprungbefehle

Die Sprungbefehle `B` und `BL` berechnen die Zieladresse PC-relativ. Die maximale Sprungdistanz beträgt $\pm$ 32 MByte. Bei `BL` wird zudem die Adresse der auf den Sprung folgenden Instruktion im LR-Register (R14) gespeichert.

2.2 Open Firmware

Die im DNARD NC verwendete Firmware wird als Open Firmware bezeichnet. Darunter versteht man Software, die den IEEE Standard 1275 [IEE94] implementiert. Solche Software muss folgende Dienste zur Verfügung stellen:

- einen Mechanismus zum Laden und Ausführen von Programmen (wie z.B. Betriebssysteme) von Disk, Tape, dem Netzwerk, etc.,
- eine prozessorunabhängige Methode zur Identifizierung von Erweiterungsgeräten sowie Firmware und Treiber für diese Geräte,

- eine erweiterbare und programmierbare Kommandosprache basierend auf Forth,
- Mechnismen zum Management von Konfigurationsvariablen, welche in NVRAM gespeichert werden,
- ein *Call Back Interface*, welches anderen Programmen erlaubt, die von der Open Firmware angebotenen Dienste zu benutzen,
- sowie Debugging Tools für Hardware, Firmware, Treiber und Systemsoftware.

Normalerweise benötigt ein Betriebssystem die Open Firmware-Treiber nur solange, bis es seine eigenen Treiber initialisiert hat. Dies geschieht aus dem einfachen Grund, dass die Open Firmware-Treiber typischerweise so einfach und klein wie möglich gehalten werden (was auch durchaus sinnvoll ist), während bei den Treibern des Betriebssystems mehr Wert auf Vollständigkeit und Effizienz gelegt wird.

SharkOberon verwendet nach der Initialisierung ebenfalls eigene Treiber (für Netzwerk, Tastatur, Maus, Grafikkarte, etc.), ist aber in den Bereichen Memory Managament und Konfigrationsvariablen (siehe Kapitel 4.2 auf Seite 30 und Kapitel 4.8 auf Seite 40) weiterhin auf die Open Firmware angewiesen.

Kapitel 3

Der Compiler

Obwohl der Compiler eigentlich nicht zum Basissystem gehört, spielt er doch eine wichtige Rolle, da ohne ihn eine Portierung des Systems unmöglich wäre. Deshalb wird er in diesem Kapitel – noch vor dem System selbst – beschrieben.

Der Compiler basiert auf der aktuellsten Version des Oberon-Compilers für den NS32000, welcher in [WG92] beschrieben wird und eine um das `FOR`-Statement erweiterte Variante der in [Wir88] definierten Sprache Oberon-1 implementiert. Zur Auswahl standen ausserdem noch der portable OP2 [Cre91] sowie ein Compiler für die Sprache Oberon-SA [Wir97], welcher Code für den SA-110 erzeugt.

Der Oberon-SA-Compiler wurde nicht verwendet, da Oberon-SA eine speziell für eingebettete Systeme ausgelegte Sprache ist, die nur eine sehr eingeschränkte Untermenge von Oberon-1 implementiert. Eine Erweiterung auf Oberon-1 wäre mindestens ebenso aufwendig gewesen wie eine Portierung des bestehenden Oberon-1-Compilers, vor allem da der Instruktionssatz des SA-110 überschaubar und einfach codiert ist.

OP2 wäre wahrscheinlich die bessere Wahl gewesen, vor allem was die Qualität des erzeugten Codes betrifft. Durch die Verwendung eines Multiple-Pass-Compilers kann oft besserer Code erzeugt werden, da zum Zeitpunkt der Codeerzeugung mehr Information verfügbar ist als bei einem Single-Pass-Compiler. Nur schon eine bessere Registerverwendung hätte sich bezahlt gemacht. Trotzdem wurde davon abgesehen, OP2 zu verwenden, da er komplexer als ein Single-Pass-Compiler ist und eine Portierung in weniger als 2 Monaten für mich nicht machbar gewesen wäre.

In der nun folgenden Beschreibung des Compilers wird nicht auf jedes Detail eingegangen. Dazu sei auf [WG92] verwiesen. Auch auf die Darstellung von Code-Patterns wird weitestgehend verzichtet, da sie sich nicht gross von denjenigen in [WG92] unterscheiden. Eine Ausnahme bilden dabei das Laden von Konstanten (siehe Kapitel 3.3.3 auf Seite 12), das durch die Einschränkungen des SA-110 bei einem Single-Pass-Compiler relativ kompliziert ist, sowie die Übersetzung des `FOR`-Statements (Kapitel 3.3.4 auf Seite 15), das im Zusammenspiel mit `EXIT` Probleme birgt, die im Oberon-Compiler aus [WG92] nicht berücksichtigt wurden und in einigen Fällen zur Erzeugung von fehlerhaftem Code führen. Ansonsten werden diejenigen Aspekte hervorgehoben, in welchen sich der SharkOberon-Compiler vom „Original-Compiler“ unterscheidet.

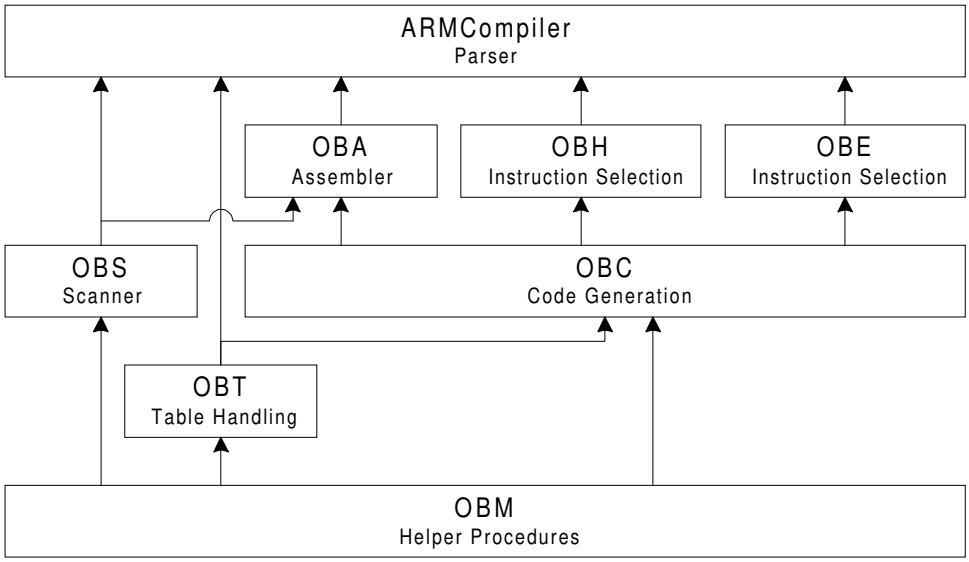


Abbildung 3.1: Struktur des Compilers

3.1 Struktur des Compilers

Der Compiler umfasst folgende Module:

- ARMCompiler (Parser)
- OBE und OBH (Instruktionsauswahl)
- OBA (Assembler)
- OBC (Codeerzeugung)
- OBS (Scanner)
- OBT (Symboltabellenverwaltung)
- OBM (Hilfsprozeduren)

Die Abhängigkeiten sind in Abbildung 3.1 dargestellt. Durch die Trennung des *Front End* (Parser, Symboltabellenverwaltung und Scanner) vom *Back End* (Codegenerator) durch die Verwendung der Module OBE und OBH musste nur das *Back End* neu geschrieben werden. Das *Front End* konnte fast unverändert übernommen werden. Es wurden einige Features hinzugefügt, die zur Compilierung des bestehenden Oberon System 3 Quellcodes nötig sind, so zum Beispiel der Export von String-Konstanten oder die `ASSERT`-Prozedur.

Der Assembler (siehe auch Kapitel 3.4 auf Seite 25) wurde nicht in *Front End* und *Back End* unterteilt, da dies aufgrund der engen Verbindung zwischen der „Sprache“ (Mnemoniks) und dem erzeugten Code keinen Nutzen bringen würde ein. Die einzige exportierte Prozedur ist `Assemble`, und somit kann bei einer Portierung des Compilers der Assembler als ganzes ersetzt werden.

3.2 Repräsentation eines Moduls im Speicher

Ein Modul besteht typischerweise aus verschiedenen Bereichen oder *Segmenten*. Dies sind

1. ein Datensegment,
2. ein Codesegment und
3. ein Konstantensegment.

Im Datensegment werden nicht (bzw. mit 0) initialisierte Daten abgelegt, d.h. es bietet Platz für die globalen Variablen des Moduls. Im Konstantensegment werden prinzipiell alle Konstanten abgelegt. Es wird sich jedoch zeigen, dass es (für den SharkOberon-Compiler) vorteilhaft ist, dort nur String-Konstanten abzulegen, während alle anderen Konstanten in den Code eingebettet werden (siehe Kapitel 3.3.3 auf der nächsten Seite).

Daten-, Code- und Konstantensegment liegen bei SharkOberon direkt hintereinander, wie in Abbildung 3.2 auf der nächsten Seite verdeutlicht wird. Prinzipiell können die Segmente völlig unabhängig voneinander im Speicher angeordnet werden, solange sie vom Code aus erreichbar bzw. adressierbar sind. Falls Daten- und Codesegment aber nicht direkt aufeinanderfolgen, so muss ein Register immer die Adresse des Datensegments enthalten (wie das *Static Base Register* des NS32000). Ohne Hardwareunterstützung führt dies zu einem nicht unerheblichen Overhead bei exportierten Prozeduren, da dort immer damit gerechnet werden muss, dass sie aus einem anderen Modul heraus aufgerufen worden sind und sie deshalb die Adresse ihres Datensegments neu laden müssen. Analoge Überlegungen treffen auch auf das Konstantensegment zu.

3.3 Codeerzeugung

3.3.1 Verwendung der Register

Die Verwendung der Register wird in Tabelle 3.1 auf der nächsten Seite gezeigt. Eine Prozedur kann davon ausgehen, dass ihr beim Eintritt alle allgemeinen Register zur Verfügung stehen. Für eine Speicherung von gegebenenfalls noch benötigten Zwischenresulaten ist also der *Caller* und nicht der *Callee* verantwortlich. Es hat sich gezeigt, dass diese Lösung besser ist, da bei den meisten Prozeduraufrufen keine oder nur wenige Zwischenresultate in Registern gehalten werden. Bei einer Speicherung durch den *Callee* müssten immer alle allgemeinen Register gespeichert werden, was auch mit einem *Block Data Transfer* einen Zyklus pro Register benötigt. Parameter werden auf dem Stack und nicht in Registern übergeben (siehe dazu auch Kapitel 3.3.5 auf Seite 20).

3.3.2 Adressierung und Alignment von Variablen und Parametern

Variablen und Parameter werden wie in Tabelle 3.2 auf Seite 13 gezeigt adressiert. Korrektes Alignment ist beim SA-110 besonders wichtig. Während zum

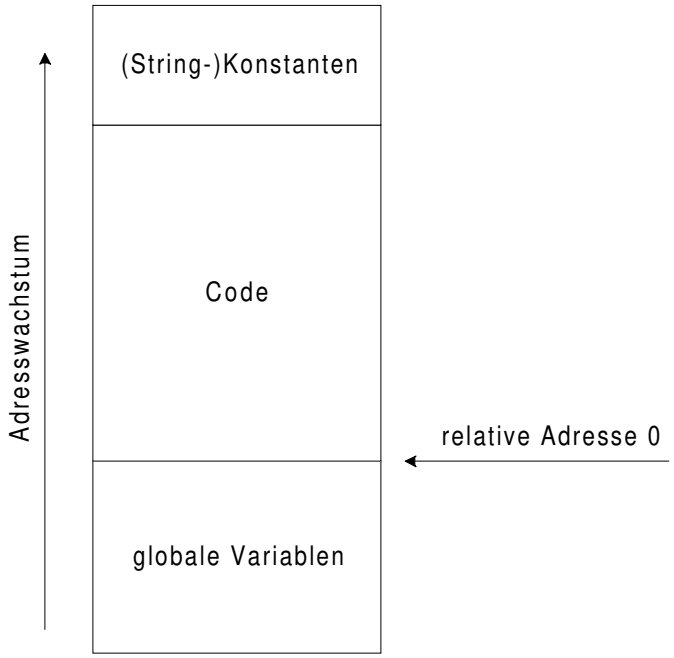


Abbildung 3.2: Repräsentation eines Moduls im Speicher

Register	Verwendung
R15	Program Counter (PC)
R14	Link Register (LR)
R13	Stack Pointer (SP)
R12	Frame Pointer (FP)
R11 – R0	Allgemeine Verwendung für Berechnungen. R0 wird ausserdem für Rückgabewerte verwendet. Falls ein `LONGREAL` zurückgegeben wird, so wird das Registerpaar (R0,R1) verwendet

Tabelle 3.1: Verwendung der Register

Beispiel bei Intel-Prozessoren nicht-alignierte Daten nur langsamer gelesen werden, führen solche Zugriffe beim SA-110 zu einem *Data Abort*. Dieses Verhalten kann zwar ausgeschaltet werden, aber dann werden nicht-alignierte Daten nicht korrekt gelesen. Aus diesem Grund verwendet SharkOberon Exceptions.

Variablen werden entsprechend ihrer Grösse auf Word-, Halfword- oder Byte-Grenzen ausgerichtet. Parameter hingegen werden immer auf Word-Grenzen ausgerichtet.

3.3.3 Laden von Konstanten

Der Bereich der Konstanten, die mit einem `MOV` geladen werden können, ist sehr beschränkt (siehe Kapitel 2.1.4 auf Seite 6). Es muss also ein Weg gefunden werden, der es erlaubt, beliebige Konstanten k mit `MIN(LONGINT)` $\leq k \leq$ `MAX(LONGINT)` zu laden. Dazu bieten sich zwei Möglichkeiten an:

Typ	Adressierung	Offset
globale Variablen	PC-relativ	< 0
lokale Variablen	FP-relativ	< 0
Parameter	FP-relativ	> 0

Tabelle 3.2: Adressierung von Variablen und Parametern

(i) Die zu ladende Konstante wird berechnet, entweder durch arithmetische oder durch bitweise Funktionen. Dies ist für jede Konstante mit maximal 4 Befehlen möglich.

(ii) Die zu ladende Konstante wird im Speicher abgelegt, und es wird Code erzeugt, der sie mittels `LDR` lädt. Dies benötigt immer eine Instruktion (mit Speicherzugriff) sowie ein weiteres Word für die Konstante.

Auch Adressen von importierten Variablen können als Konstanten betrachtet werden, da sie während der Laufzeit eines Programmes nicht ändern. Aus dieser Tatsache folgt, dass der Wert einer zu ladenden Konstante zur Compilezeit nicht notwendigerweise bekannt ist. Somit ist ein Berechnen der Konstante unmöglich.

Natürlich wäre es möglich, Adressen und Konstanten getrennt zu behandeln. Der Einfachheit halber wurde jedoch nur die zweite Variante implementiert (wobei Konstanten, die mit einem `MOV` geladen werden können, natürlich nicht im Speicher abgelegt werden), da für die erste Variante die optimale Instruktionsfolge für die Berechnung der Konstante gefunden werden müsste. Ausserdem ist es fraglich, ob Variante (i) Vorteile bietet, da zwar nicht auf den Speicher zugegriffen werden muss, aber meistens zwei oder mehr Instruktionen verarbeitet werden müssen.

Der Konstantenbuffer

Im Speicher abgelegte Konstanten werden PC-relativ adressiert. Daraus folgt, dass eine Konstante maximal 4 KByte nach dem entsprechenden `LDR` ausgegeben werden muss. Eine Möglichkeit wäre, jede Konstante direkt nach ihrem `LDR` auszugeben. Dabei müsste jedoch jedesmal ein unbedingter Sprung über die Konstante hinweg dem `LDR` folgen, was sicher nicht wünschenswert ist. Deshalb werden Konstanten zuerst in einem *Konstantenbuffer* gesammelt. Da der Konstantenbuffer wegen der Single-Pass-Natur des Compilers zwangsläufig nach den einzelnen Zugriffen ausgegeben wird, kann seine Grösse auf 4 KByte beschränkt werden. Zum Zeitpunkt der Ausgabe der `LDR`-Instruktion kann nun jedoch noch nichts über die endgültige Adresse (bzw. den Offset zum PC) der Konstante ausgesagt werden. Um dieses Problem zu lösen, werden die `LDR`-Instruktionen in einer Fixup-Liste verkettet. Anstelle des `LDR` wird dabei nur das Zielregister (4 Bit), der Offset in den Konstantenbuffer (12 Bit) sowie der Abstand der vorhergehenden `LDR`-Instruktion gespeichert (16 Bit, in Words, die `LDR`s können somit maximal 256 KByte auseinanderliegen). Nachdem die Konstanten ausgegeben worden sind, wird diese Fixup-Liste aufgelöst. Die Abbildungen 3.3 und 3.4 auf der nächsten Seite und auf Seite 17 zeigen die Implementation.

```
PROCEDURE EnterConst(c : LONGINT) : LONGINT;
VAR i : LONGINT;
BEGIN
  (* Check if const is already entered *)
  i := 0;
  WHILE (i < constCnt) & ((constBlock[i].a # c) OR
        (constBlock[i].type # Constant)) DO INC(i) END;
  IF i < constCnt THEN
    (* Constant already in memory *)
    RETURN i*4
  END;
  constBlock[constCnt].a := c;
  constBlock[constCnt].type := Constant;
  IF constCnt > constDist THEN constDist := constCnt END;
  INC(constCnt);
  RETURN (constCnt-1) * 4
END EnterConst;

PROCEDURE LoadConst*(reg, const : LONGINT);
VAR ofs, imm, rot : LONGINT;
BEGIN
  IF DecomposeConst(const, imm, rot) THEN
    PutDP1(MOV, reg, 0, const)
  ELSE
    ofs := EnterConst(const)
    code[pc] := (pc-constRefChain)*10000H + reg*1000H + ofs;
    constRefChain := pc;  INC(pc)
  END
END LoadConst;
```

Abbildung 3.3: Programmfragment „Laden von Konstanten“

Ausgabe der Konstanten

Der Konstantenbuffer kann nur dann ausgegeben werden, wenn er vom Kontrollfluss nicht erreicht werden kann, d.h. immer nach einem unbedingten Sprung, wobei natürlich die Sprunginstruktionen zu dieser Stelle angepasst werden müssen. Solche Stellen sind unter anderem:

- Das Ende einer Prozedur
- Nach dem Sprung am Ende einer `WHILE`-Schleife
- Nach dem Code eines `THEN`-Teiles, wenn darauf ein `ELSE` oder `ELSIF` folgt

Um den Ort der Ausgabe optimal zu bestimmen, muss der gesamte erzeugte Code einer Prozedur betrachtet werden. Da dies mit einem Single-Pass-Compiler nicht möglich ist, geht das implementierte Verfahren *greedy* vor, d.h. es werden alle angesammelten Konstanten so früh wie möglich ausgegeben.

Es kann natürlich vorkommen, dass der Konstantenbuffer voll ist, ohne dass eine geeignete Stelle auftritt, an welcher er geleert werden kann. In diesem Fall wird künstlich eine Stelle geschaffen, die vom Kontrollfluss nicht erreicht wird. Dazu wird einfach ein unbedingter Sprung ausgegeben und danach der Konstantenbuffer geleert. Wie aus Tabelle 3.3 auf der nächsten Seite ersichtlich ist, tritt dieser Fall jedoch nur sehr selten ein.

Bei der Ausgabe der Konstanten müssen alle `LDR`-Answeisungen der Fixup-Liste angepasst werden. Ausserdem müssen gewisse Konstanten (z.B. Adressen von importierten Variablen) in andere Fixup-Listen eingetragen werden. Abbildung 3.4 auf Seite 17 zeigt die Implementation.

3.3.4 Das `FOR`-Statement

Gemäss Definition in [MW95] wird das `FOR`-Statement als eine While-Schleife übersetzt, wobei der Endwert nur einmal zu Beginn ausgewertet wird. Ist die obere Grenze eine Variable, so muss der Wert, den die Variable beim Eintritt in das `FOR`-Statement hatte, in einer temporären Variable gespeichert werden. Platz für diese Variable kann an folgenden Orten alloziert werden:

1. bei den globale Variablen
2. bei den lokalen Variablen
3. in einem Register
4. auf dem Stack

Der globale Variablenbereich ist keine gute Wahl, da Speicher verschwendet wird, wenn nicht ein Verfahren implementiert wird, welches das *Recycling* temporärer Variablen zulässt.

In einem Multiple-Pass-Compiler wird die temporäre Variable typischerweise bei den lokalen Variablen abgelegt. In einem Single-Pass-Compiler ist dies aufwendiger, da in diesem nachträglich die `SUB`-Instruktion geändert werden

Modul	Anzahl Blöcke	Anzahl Sprünge	Min. Blockgr.	Max. Blockgr.	Durchschnittl. Blockgrösse
OFW	23	0	1	3	2.17
Kernel	87	0	1	34	6.28
Disk	1	0	2	2	2.00
FileDir	45	0	1	19	6.44
Files	18	0	1	8	2.61
CS8900	17	0	1	42	10.53
NetBase	11	0	1	8	3.18
NetPorts	5	0	1	3	1.60
NetIP	35	0	1	35	4.86
NetUDP	9	0	1	4	2.00
Modules	38	0	1	48	5.26
SoftFloat	14	0	1	5	2.43
Input	83	0	1	35	5.41
Objects	38	0	1	49	7.18
Display	47	0	1	73	7.26
Fonts	10	0	2	27	6.50
Texts	84	0	1	28	3.81
Viewers	14	0	1	30	5.50
Oberon	54	0	1	56	4.43
System	87	0	1	113	9.59
MenuViewers	28	0	1	10	3.21
TextFrames	76	1	1	71	6.13
Display3	62	0	1	53	4.16
Effects	62	0	1	59	4.40
NetSystem	55	0	1	22	5.35
Gadgets	177	0	1	31	4.27
TextGadgets	159	0	1	56	3.35
TextGadgets0	154	0	1	20	3.51
TextDocs	86	0	1	26	5.06
Views	76	0	1	25	3.37
BasicGadgets	219	0	1	12	2.79
OBS	17	0	1	47	5.24
OBT	34	0	1	68	5.94
OBC	65	0	1	70	7.12
OBE	181	0	1	11	1.97
OBH	33	0	1	13	3.09
ARMCompiler	155	0	1	32	3.98
OBA	69	2	1	175	10.91

Tabelle 3.3: Übersicht über eingefügte Konstantenblöcke ausgewählter Module. Die Grössen sind in Words angegeben.

```
PROCEDURE FlushConsts*;
(* Writes out the constants and fixes all offsets *)
VAR i, j, next, destreg, ofs, cnt : LONGINT;
BEGIN
  flushing := TRUE;
  constDist := -1; (* reset here so that we dont get recursive
                      calls (because of PutSDT later on), if we
                      are already in the "safety range" *)

  (* Fix all references to a constant *)
  i := constRefChain;
  IF i # -1 THEN INC(nofConstBlocks) END;
  cnt := 0;
  WHILE i # -1 DO
    INC(cnt);
    (* Extract dest reg, ofs and next pos that needs fixup *)
    next := i - SYSTEM.LSH(code[i], -16);
    destreg := SYSTEM.LSH(code[i], -12) MOD 10H;
    ofs := code[i] MOD 1000H;

    (* Adjust ofs and generate an LDR *)
    j := pc; pc := i;
    PutSDT(LDR, destreg, PC, j*4 + ofs);
    pc := j;
    i := next
  END;

  (* Emit all constants *)
  FOR i := 0 TO constCnt-1 DO
    code[pc] := constBlock[i].a;
    CASE constBlock[i].type OF
      Constant:    (* nothing to do *)
      |ProcAddr:   AddAddrFixup(pc, unknownProcTab[constBlock[i].b].a2)
      |StringAddr: AddAddrFixup(pc, strfixlist)
      |ExtAddr:    AddAddrFixup(pc, extaddrfixlist)
    END;
    INC(pc)
  END;
  constCnt := 0;  constRefChain := -1;
  constDist := -1;  flushing := FALSE;
END FlushConsts;
```

Abbildung 3.4: Programmfragment „Leeren des Konstantenbuffers“

```
     MOV   RB,0H
     STR   RB,[FP,-4H]  (* i:=0 *)
     LDR   RB,[FP,-8H]
     STR   RB,[SP,-4H]! (* store upper limit (j) *)
loop LDR   RB,[FP,-4H]
     LDR   RA,[SP,+0H]
     CMPS  RB,RA        (* i <= tmp var?   *)
     BGT   done         (* no -> leave FOR *)
     ... FOR Body ...
     LDR   RB,[FP,-4H]
     ADDS  RB,RB,1H
     STR   RB,[FP,-4H]  (* INC(i) *)
     B     loop
done ADDS  SP,SP,4H     (* remove upper limit *)
```

Abbildung 3.5: Beispiel „`FOR i:=0 TO j DO ... END`“

müsste, welche den Platz für die lokalen Variablen schafft. Auf dem SA-110 kommt noch hinzu, dass eine `SUB`-Instruktion nicht jede beliebige Konstante verwenden kann, was dazu führt, dass die Grösse der lokalen Variablen immer zuerst in ein Register geladen werden muss, bevor sie vom SP subtrahiert werden kann.

Eine Speicherung in einem Register würde dieses Register für den ganzen `FOR`-Body blockieren und wäre somit „Verschwendung“, vor allem da der Endwert ja nur zur Entscheidung verwendet wird, ob die Schleife weiter durchlaufen werden soll.

Die natürlichste und einfachste Lösung für einen Single-Pass-Compiler ist eine Speicherung auf dem Stack. Der Endwert wird auf den Stack gelegt und nach dem `FOR`-Statement wieder vom Stack entfernt. Dies ist in Abbildung 3.5 verdeutlicht. Es ist wichtig, dass temporäre Variablen am Ende des `FOR`s entfernt werden und nicht etwa erst am Ende der Prozedur, da sonst bei verschachtelten `FOR`-Statements alle Zugriffe der übergeordneten `FOR`s angepasst werden müssten.

Das soeben beschriebene Verfahren setzt voraus, dass der Kontrollfluss immer strukturiert ist, d.h. dass keine Sprünge aus dem Body eines `FOR`-Statements heraus stattfinden. Leider gibt es aber in Oberon zwei Statements, die unstrukturierten Kontrollfluss zur Folge haben: `EXIT` und `RETURN`.

Das `EXIT`-Statement erlaubt nahezu beliebige Sprünge, unter anderem auch aus einem `FOR`-Body heraus, wie in Abbildung 3.6 auf der nächsten Seite dargestellt wird. Wird das Entfernen von temporären Variablen wie beschrieben implementiert, so wird fehlerhafter Code erzeugt. Da `EXIT` das Ende des inneren `FOR`-Statement überspringt, wird dessen temporäre Variable nicht entfernt. Dies führt dazu, dass das äussere `FOR` den falschen Endwert verwendet und somit für

```
MODULE UnstructuredCF;

IMPORT Out;

PROCEDURE P;
VAR a, b, i, j : LONGINT;
BEGIN
  a := 3; b := 5;
  FOR i := 0 TO a DO
    Out.String("i="); Out.Int(i, 1); Out.Ln;
    LOOP
      FOR j := 0 TO b DO EXIT END
    END
  END
END P;

END UnstructuredCF.
```

Abbildung 3.6: Beispiel „Unstrukturierter Kontrollfluss aufgrund von `EXIT`“

die Werte 0 bis und mit 5 (anstelle von 3) durchlaufen wird[1].

Eine Konstruktion wie in Abbildung 3.6 tritt in der Praxis zugegebenermassen sehr selten auf. Dennoch ist es keineswegs akzeptabel, dass in einem solchen Fall fehlerhafter Code erzeugt wird. Es muss deshalb eine Lösung gefunden werden, die (a) das Problem löst und (b) die Effizienz von unproblematischen Fällen nicht verringert.

Die Lösung des Problems gestaltet sich äussert einfach: Für jedes `LOOP`-Statement wird jeweils die Anzahl der „offenen“ `FOR`-Statements gespeichert, die eine temporäre Variable benötigen. Sobald ein `EXIT` auftritt, kann aufgrund des Zählers bestimmt werden, wieviele temporäre Variablen vom Stack entfernt werden müssen, und es wird entsprechender Code erzeugt.

Diese Anpassungen benötigen nur wenige Zeilen Quellcode und haben keinen Einfluss auf die Effizienz der problemlosen Fälle. Als Beispiel für die Einfachheit wird die Implementation der Codeerzeugung für `EXIT` in Abbildung 3.7 auf der nächsten Seite dargestellt. Nur die mit `(**)` markierten Zeilen sind nötig, um die Entfernung von temporären Variablen zu ermöglichen.

Aus den obigen Betrachtungen geht hervor, dass unstruktierter Kontrollfluss in einem `FOR`-Statement nur dann ein Problem darstellt, wenn der Schleifenkopf eines übergeordneten `FOR`s ein weiteres Mal ausgeführt werden kann (da nur im Schleifenkopf der Endwert zum Vergleich herangezogen wird). Aus diesem Grund bereitet das `RETURN`-Statement keine Probleme, da ein Rücksprung zum *Caller* erfolgt und deshalb kein übergeordnetes `FOR` erneut ausgeführt werden kann.

[1] Es sei hier die Anmerkung erlaubt, dass der aktuelle Oberon-Compiler für die Ceres-3 fehlerhaften Code erzeugt, der genau das geschilderte Verhalten zeigt.

```
...
ELSIF sym = OC.exit THEN
  IF (LoopLevel > 0) & (LoopLevel <= MaxLoopNesting) THEN  (**)
    OBH.RemoveTempVars(ForDepth[LoopLevel-1])              (**)
  END;                                                     (**)
  OBS.Get(sym);  OBE.FJ(L0);
  IF LoopLevel = 0 THEN OBS.Mark(45)
  ELSIF ExitNo < MaxExits THEN LoopExit[ExitNo] := L0; INC(ExitNo)
  ELSE OBS.Mark(214)
  END
...
```

Abbildung 3.7: Programmfragment: „Codeerzeugung für `EXIT`“

```
    LDR   R0,[PC,xx]    (* load address of procedure *)
    MOV   LR,PC         (* store return address *)
    MOV   PC,R0         (* call procedure *)
```

Abbildung 3.8: Beispiel „Branch über eine Distanz > 32 MByte“

3.3.5 Aufruf von Prozeduren und Parameterübergabe

Die Parameterübergabe erfolgt auf dem Stack. Die Parameter werden dabei wie gewohnt von links nach rechts ausgewertet. Der *Static Link* wird nur dann erzeugt, wenn eine lokale Prozedur aufgerufen wird. Er wird nach dem letzen Parameter auf den Stack gelegt. Für die Entfernung der Parameter und des *Static Links* vom Stack ist der *Callee* verantwortlich. Wird eine Funktionsprozedur aufgerufen, so ist der *Caller* dafür verantwortlich, die nach dem Aufruf noch benutzten Register zu speichern. Das Resultat wird von der Funktionsprozedur in R0 zurückgegeben, oder im Registerpaar (R0, R1), falls es sich um einen `LONGREAL` handelt.

Zum Aufruf der Prozedur wird der Sprungbefehl `BL` verwendet. Dies bedingt, dass alle Module innerhalb eines 32 MByte grossen Bereiches des virtuellen Adressraums liegen (was vom *Module Loader* garantiert wird). Es wäre zwar möglich, Sprünge zu implementieren, die den gesamten Adressraum abdecken können, aber diese würden für importierte Prozeduren 3 Instruktionen benötigen (ein Beispiel ist in Abbildung 3.8 dargestellt. Der PC enthält immer den Wert (Adresse der Instr.)+8, das vielleicht verwirrende `MOV LR,PC` ist also korrekt). Da solche Aufrufe aber relativ häufig vorkommen, wurde darauf verzichtet. Ausserdem hat sich gezeigt, dass 32 MByte für Code mehr als ausreichend sind, da die meisten Module nach dem Laden nicht mehr aus dem Speicher entfernt werden.

```
STMDB SP!, {FP LR }      PROCEDURE P(a : LONGINT);
MOV   FP,SP

MOV   RB,0H              VAR b : LONGINT;
STR   RB,[SP,-4H]!

                         BEGIN
LDR   RB,[FP,+8H]          b := a
STR   RB,[FP,-4H]

MOV   SP,FP              END P;
LDR   FP,[SP],+4H
LDR   PC,[SP],+8H
```

Abbildung 3.9: Beispiel „Prolog und Epilog einer Prozedur“

3.3.6 Prolog und Epilog

Der Prolog einer Prozedur sichert zuerst den FP (*Dynamic Link*). Ausserdem muss auch die Rücksprungadresse im LR gesichert werden. Danach wird auf dem Stack Platz für die lokalen Variablen geschaffen und zum Schluss noch gegebenenfalls Kopien von dynamischen Array-Parametern gemacht. Ausserdem werden die lokalen Variablen mit 0 initialisiert, falls die entsprechende Compileroption angegeben worden ist.

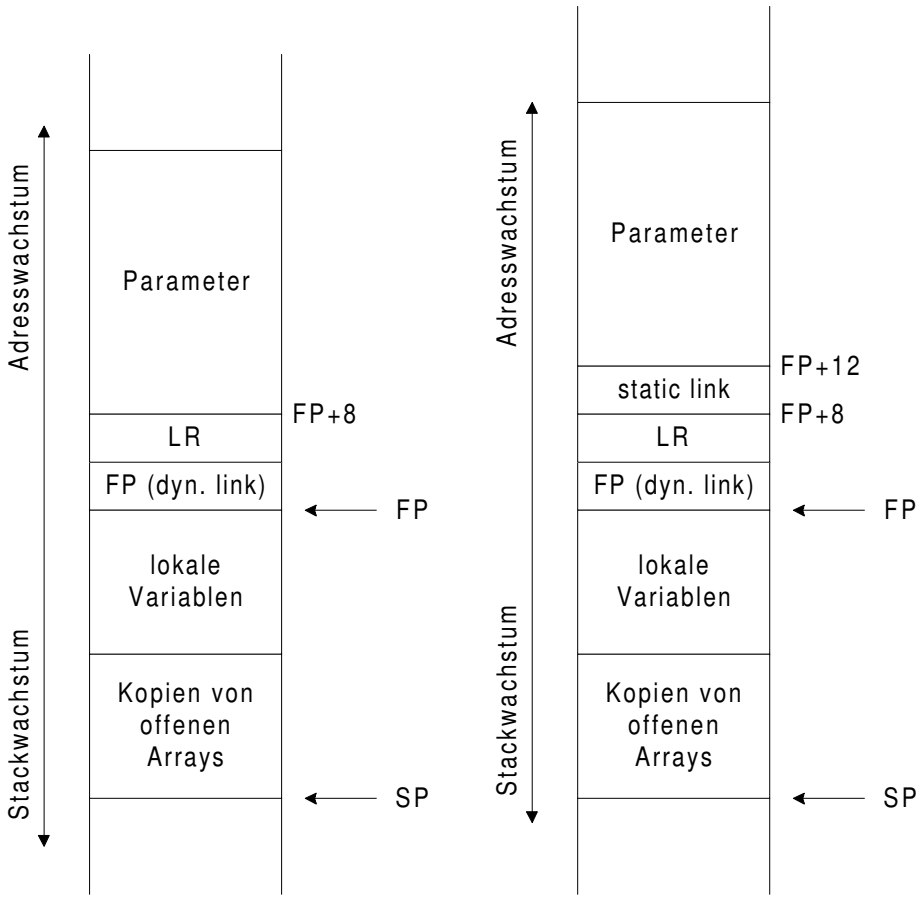
Der Epilog stellt den alten FP wieder her und entfernt die Rücksprungadresse und die Parameter vom Stack. Dazu wird, wie in der Abbildung 3.9 ersichtlich, Gebrauch von einem *postindexed* `LDR` (mit implizitem *Base Register Writeback*) gemacht. Falls keine Parameter entfernt werden müssen, so wird ein *Block Data Transfer* verwendet, um den FP und PC mit nur einer Instruktion zu laden.

3.3.7 Das Procedure Activation Frame

Das sich aus den obigen Betrachtungen ergebende *Procedure Activation Frame* ist in Abbildung 3.10 auf der nächsten Seite dargestellt.

3.3.8 `DIV` und `MOD`

Da der SA-110 über keine Divisionsinstruktion verfügt, müssen Division und Modulo ausprogrammiert werden. Als Basis diente dabei die entsprechende Prozedur von NetBSD/arm32. Die Prozedur heisst `DivMod` und wird vom Modul „Kernel“ exportiert. Sie erwartet die Parameter in den Registern R0 und R1. Nach dem Aufruf enthält R0 den Wert (R0 `DIV` R1) und R1 den Wert (R0 `MOD` R1). Das Verwenden von Division/Modulo führt *nicht* zum impliziten Import des Moduls „Kernel“. Statt dessen wird wie in Native Oberon Code für den Aufruf einer Prozedur des importierten Moduls 0 generiert. Da die Numerierung



(a) Ohne Static Link (b) Mit Static Link

Abbildung 3.10: Das Procedure Activation Frame

```
Extern Call Fixlist:
  0000002C  DivMod
  0000001C  DivMod

Procedure P:                                  PROCEDURE P;
          (* Prolog weggelassen *)           VAR a : LONGINT;
                                              BEGIN
00000014  LDR   R0,[FP,-8H]                     a := a DIV b;
00000018  LDR   R1,[FP,-4H]
0000001C  BL    30H
          (* Sprungziel wird vom Loader gefixt *)
00000020  STR   R0,[FP,-4H]

00000024  LDR   R0,[FP,-8H]                     a := a MOD b
00000028  LDR   R1,[FP,-4H]
0000002C  BL    40H
          (* Sprungziel wird vom Loader gefixt *)
00000030  STR   R1,[FP,-4H]
          (* Epilog weggelassen *)           END P;
```

Abbildung 3.11: Beispiel „Verwendung von `DIV` und `MOD`“

von importierten Modulen bei 1 beginnt, erkennt der *Module Loader* den speziellen Aufruf und kann die entsprechende Adresse einsetzen. In Abbildung 3.11 ist ein Beispiel für den Aufruf von `DivMod` dargestellt.

Es wurde darauf verzichtet, „Kernel“ implizit zu importieren, da `DivMod` dann exportiert werden müsste und somit auch „von Hand“ aufrufbar wäre. Dies ist aber nicht gewünscht, da `DivMod` nicht die *Calling Conventions* von SharkOberon einhält (die Parameter werden in Registers übergeben).

Prozedurname	Operation
RealAdd	Addition
RealSub	Subtraktion
RealMul	Multiplikation
RealDiv	Division
RealEq	$=$
RealNe	$\#$
RealLt	$<$
RealLe	$\leq$
RealGt	$>$
RealGe	$\geq$
RealNeg	unäres $-$
RealEntier	`ENTIER`
RealAbs	`ABS`

(a) REAL-Operationen

Prozedurname	Operation
LRealAdd	Addition
LRealSub	Subtraktion
LRealMul	Multiplikation
LRealDiv	Division
LRealEq	$=$
LRealNe	$\#$
LRealLt	$<$
LRealLe	$\leq$
LRealGt	$>$
LRealGe	$\geq$
LRealNeg	unäres $-$
LRealEntier	`ENTIER`
LRealAbs	`ABS`

(b) LONGREAL-Operationen

Prozedurname	Operation
LIntToReal	`INTEGER` $\rightarrow$ `REAL`
LIntToLReal	`LONGINT` $\rightarrow$ `LONGREAL`
RealToLReal	`REAL` $\rightarrow$ `LONGREAL`
LRealToReal	`LONGREAL` $\rightarrow$ `REAL`

(c) Konversionsprozeduren

Tabelle 3.4: Vom Compiler aufgerufene Prozeduren in „SoftFloat“

3.3.9 Gleitkommaunterstützung

Der SA-110 verfügt weder über einen Gleitkommakoprozessor noch über die Möglichkeit, um einen solchen erweitert zu werden [Dig96]. Deshalb wurde darauf verzichtet, einen Gleitkomma-Emulator zu schreiben. Statt dessen wurde die Gleitkommaunterstützung als gewöhnliches Modul implementiert, das vom Compiler bei Bedarf *implizit* importiert wird. Gleitkommaoperationen werden dann in Aufrufe der entsprechenden Prozeduren übersetzt. Ein impliziter Import ist hier problemlos, da sich die Gleitkommaprozeduren im Gegensatz zu `DivMod` an die *Calling Conventions* halten. Ausserdem ist es so einfacher, das Gleitkommamodul durch eine schnellere oder in Oberon geschriebene Version zu ersetzen. Das Modul, welches die Prozeduren zur Verfügung stellt, muss „SoftFloat“ heissen und mindestens die in Tabelle 3.4 aufgeführten Prozeduren exportieren.

Auf die Implementierung der Gleitkommaprozeduren wird in Kapitel 4.9 auf Seite 41 eingegangen.

3.4 Der integrierte Assembler

Zur Unterstützung der Systemprogrammierung wurde ein vollständiger SA-110-Assembler in den Compiler integriert. Er unterstützt die in [Adv96] benutzten Instruktionen sowie die Syntax mit folgenden Ausnahmen:

- Vor Integer-Konstanten darf kein # stehen
- `SPSR_flags` und `CPSR_flags` wird nicht erkannt. Statt dessen wird `SPSR, flags` und `CPSR, flags` verwendet.
- Koprozessorbefehle werden nicht unterstützt.
- Bei *Block Data Transfer* Instruktionen können in der Registerliste keine Bereiche angegeben werden. Anstelle von „R0 – R3“ muss also „R0, R1, R2, R3'“ verwendet werden.

Zusätzlich wird eine Direktive `ADDR` zur Verfügung gestellt, welche die Adresse einer Variablen, eines Parameters oder einer Prozedur in ein Register lädt.

3.4.1 Verwendung des Assemblers

Der Assembler wird wie unter Native Oberon verwendet: Wenn `CODE` anstelle von `BEGIN` in einer Prozedur verwendet wird, so wird der restliche Quellcode bis `END` als Assembler-Quellcode betrachtet. Abbildungen 3.12 und 3.13 verdeutlichen die Verwendung des Assemblers. Achtung: Bei `CODE`-Prozeduren wird weder ein Prolog noch ein Epilog erzeugt!

3.4.2 Implementation des Assemblers

Dank der einfachen Codierung der SA-110-Instruktionen konnte ein einfacher, tabellengesteuerter Assembler implementiert werden. Jede erkannte Instruktion wird mit bis zu 6 *Argument-Handlern* in einer Tabelle eingetragen. Ein Identifier wird genau dann als Mnemonik oder Direktive erkannt, wenn ein Eintrag in der Tabelle existiert. In diesem Fall werden dann alle installierten Argument-Handler aufgerufen und die zurückgelieferten Werte bitweise-oder verknüpft. Abbildung 3.14 auf Seite 27 zeigt die Implementation des Assemblerkerns.

```
PROCEDURE Move(src, dest, len : LONGINT);
CODE
  LDMFD SP!,{R0,R1,R2} (* R0 = len   R1 = dest   R2 = src *)
  CMP   R0,0
  MOVEQ PC,LR          (* if len=0 then return *)
loop
  SUBS  R0,R0,1
  LDRB  R3,[R2,R0]
  STRB  R3,[R1,R0]
  BNE   loop
  MOV   PC,LR
END Move;
```

Abbildung 3.12: Beispiel „Verwendung des Assemblers“

```
MODULE AsmDemo;

VAR a : LONGINT;

PROCEDURE P(b : LONGINT);
CODE
  STMFD SP!,{FP, LR}
  MOV   FP, SP
  ADDR  R0, a       (* load addr of a *)
  LDR   R1, [FP,8]  (* load b *)
  STR   R1, [R0,0]  (* a := b *)
  LDR   FP, [SP], 4
  LDR   PC, [SP], 8
END P;

END AsmDemo.
```

Abbildung 3.13: Beispiel „Verwendung von `ADDR`“

```
PROCEDURE Assemble*;
VAR i : INTEGER; op,cw : LONGINT;
BEGIN
  labelCnt := 0;
  Get(sym);
  LOOP
    IF sym = OC.end THEN EXIT END;
    WHILE sym = OC.ident DO AddLabel; Get(sym) END;
    IF sym = OC.end THEN EXIT END;
    IF sym # mnemo THEN
      OBS.Mark(400);
      WHILE (sym # mnemo) & (sym # OC.end) & (sym # OC.eof) DO
        Get(sym)
      END;
      IF (sym = OC.end) OR (sym = OC.eof) THEN RETURN END
    END;
    cw := mnemoTab[OBS.intval].opCode;
    op := OBS.intval;
    Get(sym);
    IF cw = -1 THEN (* Directive *)
      IF mnemoTab[op].arg[0] # NIL THEN
        cw := mnemoTab[op].arg[0]()
      END
    ELSE (* mnemonic *)
      i := 0;
      LOOP
        cw := cw + mnemoTab[op].arg[i]();  INC(i);
        IF mnemoTab[op].arg[i] # NIL THEN Match(OC.comma)
        ELSE EXIT
        END
      END;
      OBC.PutWord(cw)
    END
  END;
  IF sym # OC.end THEN OBS.Mark(OC.end) END;
  CheckUndefLabels
END Assemble;
```

Abbildung 3.14: Programmfragment „Kern des Assemblers“

Kapitel 4

Das System

In diesem Kapitel wird die Implementierung des Systems beschrieben. Sie basiert weitestgehend auf dem Quellcode von Native Oberon V2.3.2 (`http://www.oberon.ethz.ch/native/`). Ausser einer Änderung im Netzwerk-Subsystem (die in Kapitel 4.7 auf Seite 39 beschrieben wird) mussten nur die hardwareabhängigen Teile neu programmiert werden. Darunter fallen insbesondere:

- Initialisierung des Displays
- Zugriff auf den Ethernet-Controller
- Allokation und Freigabe von physikalischen Seiten sowie Bereichen im virtuellen Adressraum
- Interrupt- und Exception-Handling

4.1 Laden des Boot Images

Open Firmware versucht nach dem Einschalten des Computers, das *Boot Image* vom voreingestellten Gerät zu lesen. Auf dem NC ist dies standardmässig das Gerät „net“. Beim Booten vom Netz wird entweder BOOTP [CG85] bzw. DHCP [Dro97] oder TFTP [Sol92] verwendet. Im zweiten Fall muss die IP-Adresse des Servers spezifiziert werden, die IP-Adresse des Clients und der Name des *Boot Images* können optional angegeben werden.

Das *Boot Image* kann in einem der folgenden Formate vorliegen:

- a.out
- Forth
- FCode
- raw binary

Ist das *Boot Image* weder ein FCode- noch ein Forth-Programm, dann wird es an die Adresse geladen, die in der Konfigurationsvariable **load-base** angegeben ist (Defaulteinstellung F0000000H) und mit der Ausführung begonnen. Der *Boot Linker* von SharkOberon verwendet das a.out-Format.

Prozedur	Funktion
MapPages(n,virt, phys, mode)	n Seiten beginnend bei der physikalischen Adresse phys werden im Mode mode an die virtuelle Adresse virt gemapped.
UnmapPages(n, virt)	Das Mapping von n Seiten beginnend bei der virtuellen Adresse virt wird augehoben.
AllocAndMap(n,virt,mode)	Es werden n Seiten alloziert und beginnend bei der virtuellen Adresse virt im Mode mode gemapped.
ModifyMappingMode(n,virt,mode)	Der Mode von n Seiten beginnend bei der virtuellen Adresse virt wird auf mode gesetzt.
FreeAndUnmap(n,virt)	Es werden n Seiten beginnend bei der virtuellen Adresse virt freigegeben.

Tabelle 4.1: Prozeduren zur Verwaltung des physikalischen Speichers

4.2 Verwaltung des physikalischen Speichers

Die Verwaltung des physikalischen Speichers sowie des virtuellen Adressraums wird von der Open Firmware übernommen. Die Open Firmware Services zur Allokation von physikalischen Speicher schlagen fehl, wenn nicht genügend *direkt aufeinanderfolgende* Seiten verfügbar sind, um eine Speicheranforderung zu erfüllen. Da es jedoch in den meisten Fällen unwichtig ist, ob die Seiten aufeinanderfolgend sind, exportiert der Kernel die in Tabelle 4.1 aufgelisteten Prozeduren, welche versuchen, Speicheranforderungen durch „Zusammensetzen“ zu erfüllen. Die Prozeduren `MapPages` und `UnmapPages` werden meist von Gerätetreibern verwendet, um auf den Geräten vorhandenen Speicher in den Adressraum einzublenden (zum Beispiel das Video-RAM). Die übrigen Prozeduren werden vom *Module Loader* benötigt.

4.3 Verwendung des virtuellen Adressraums

Abbildung 4.1 auf der nächsten Seite gibt eine Übersicht über die Verwendung des virtuellen Adressraums. Die wichtigsten Bereiche werden im folgenden besprochen.

4.3.1 Die Seite 0

Die Seite 0 wird normalerweise als *not valid* gekennzeichnet, damit `NIL`-Zugriffe einfach und effizient erkannt werden können. Auf dem Network Computer ist dies nicht möglich, da ab der Adresse 0 die Adressen der Exception-Handler gespeichert werden. Um `NIL`-Zugriffe dennoch effizient abzufangen, erhalten pri-

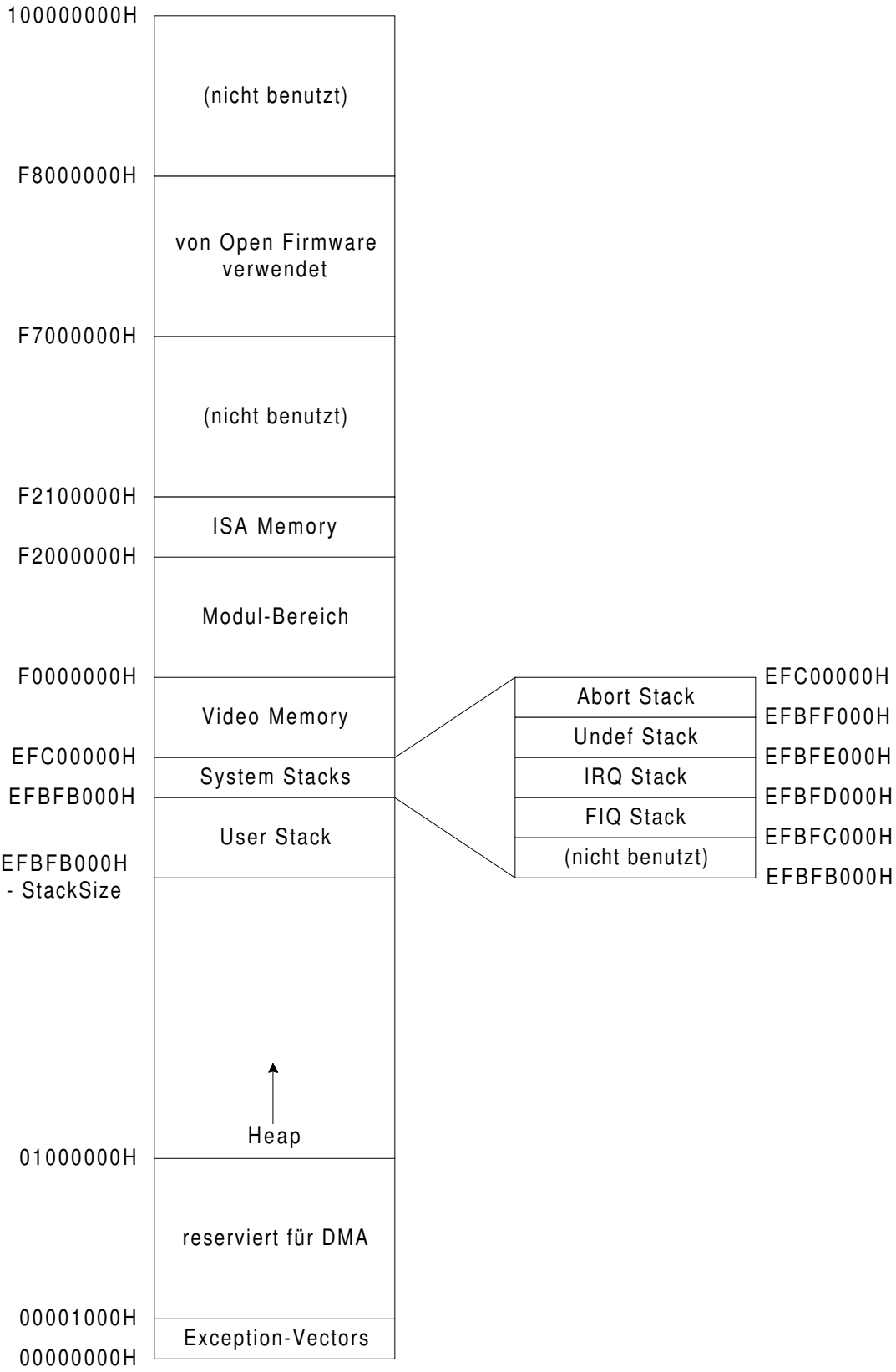


Abbildung 4.1: Verwendung des virtuellen Adressraums

vilegierte Modi Lesezugriff. Schreibzugriff wird keinem Modus gewährt. Somit werden nur noch `NIL`-Zugriffe vor Beendigung der Initialisierung des Kernels nicht erkannt, was aber vernachlässigbar ist, da in einem solchen Fall das System kaum stabil wäre.

4.3.2 Der Modul-Bereich

Für die Allokation von Adressraum für Module wird das in [WG89] beschriebene Verfahren verwendet: Adressraum wird sequentiell alloziert. Diese Adressen werden auch nach einer Freigabe des Moduls nicht mehr verwendet. Dadurch können zwar „nur“ 32 MByte verwendet werden (siehe Kapitel 3.3.5 auf Seite 20), aber dafür wird die Stabilität des Systems erhöht. Würde Adressraum wiederverwendet, so könnte eine Prozedurvariable p, die auf eine Prozedur M.P zeigt, auch nach der Freigabe von M immer noch auf eine gültige Adresse zeigen. Ein Aufruf via p würde dann zu undefiniertem Verhalten führen.

4.3.3 Der Heap-Bereich

Der Heap-Bereich wird beim Systemstart als ein grosser Block alloziert. Die Grösse wird dabei durch eine Konfigurationsvariable gesteuert. Diese Lösung ist leider nicht optimal, da dadurch der sowohl dem Heap als auch den Modulen zur Verfügung stehende Speicher von vornherein festgelegt wird und sich das System nicht den eigentlichen Anforderungen anpassen kann. Ideal wäre *allocation on demand*: Sobald eine Speicheranforderung durch `NEW` nicht mehr erfüllt werden kann, versucht das System, aus einem *Page Pool* neue Seiten zu allozieren und in den Adressraum einzublenden. Bei einer Speicherfreigabe könnten am Ende des Heaps liegende Seiten wieder freigegeben werden.

Wie bereits erwähnt wurde, wird für die Verwaltung des Speichers und des Adressraums Open Firmware eingesetzt. Da sich herausstellte, dass Open Firmware Probleme mit sehr vielen Allokationen von nur je einer Seite hat, wurde das eben beschriebene System *nicht* implementiert.

4.4 Initialisierung des Systems

Die Initialisierung verläuft im grossen und ganzen wie bei Native Oberon. Zuerst wird der Kernel initialisert. Dazu sind folgende Schritte nötig:

1. Die Exception-Vektoren auf die einzelnen Handler-Prozeduren setzen
2. Maske des Interrupt-Controllers setzen
3. Management des physikalischen Speichers initialisieren
4. Stacks und Heap initialisieren
5. Real-Time-Clock initialisieren (die RTC wird als System-Timer verwendet)
6. In den User-Mode wechseln

Nach der Initialisierung des Kernels wird der Body des Moduls „Modules“ abgearbeitet. Hier treten auch die einzigen Unterschiede zur Initialisierung von Native Oberon auf. Oberon System 3 ist ein disk-basiertes System. Da der NC im Normalfall aber über keine Disk verfügt und über ein Netz gebootet wird, muss eine RAM-Disk zur Verfügung gestellt werden, damit das System korrekt aufgestartet werden kann.

4.4.1 Initialisierung der RAM-Disk

Die RAM-Disk wird via TFTP [Sol92] initialisiert. Dabei wird ein sehr einfaches Protokoll verwendet: Zuerst wird vom Fileserver (dessen IP-Adresse von Open Firmware geliefert wird) eine Datei angefordert, welche die Namen aller Dateien enthält, die in der RAM-Disk gespeichert werden müssen. Optional kann zu jeder Datei noch eine CRC32-Prüfsumme und der Name angebenen werden, unter welchem sie in der RAM-Disk abgelegt wird. Ausserdem besteht die Möglichkeit, Oberon-Archive anzugeben, die beim Laden *on the fly* dekomprimiert werden. Dies wird durch die Option „[decompress]“ angezeigt. Fehlt „[decompress]“, so wird ein Archiv wie eine gewöhnliche Datei behandelt und einfach in der RAM-Disk gespeichert.

Der Name der „Inhaltsdatei“ ist im allgemeinen „RAMDisk.Content“, kann aber mittels einer Konfigurationsvariablen geändert werden (siehe Anhang E auf Seite 65). Das Format der Datei wird durch folgende Syntax beschrieben:

```
ContentFile      =  { Line }.
Line             =  RemoteFileName
                    [ LocalFileName | "[decompress]" ]
                    [ Crc ].
Crc              =  digit { hexdigit } "H".
RemoteFileName   =  { char }.
LocalFileName    =  { char }.
```

Sobald alle Dateien heruntergeladen worden sind, kann mit der Initialisierung des Systems wie von Native Oberon gewohnt weitergefahren werden.

4.4.2 Implementation der RAM-Disk

Die Implementation der RAM-Disk basiert auf dem Modul „Disk“. Die Anzahl der zur Verfügung stehenden Sektoren wird beim Systemstart festgelegt und kann zur Laufzeit nicht mehr geändert werden. Speicher für die einzelnen Sektoren wird jedoch dynamisch alloziert. Dadurch wird nur soviel Speicher verwendet, wie nötig ist.

Die Sektorgrösse wurde nicht geändert und beträgt 2048 Bytes. Dadurch konnten die Module „FileDir“ und „Files“ ohne Änderungen übernommen werden. Bei einer physikalischen Disk können so grosse Sektoren durchaus die Zugriffszeit verringern, da pro Zugriff mehr Daten gelesen werden und somit die Anzahl Zugriffe auf die (im Vergleich zu RAM) langsame Disk gesenkt wird. Bei einer RAM-Disk ist aber die Zugriffszeit auf einzelne Sektoren vernachlässigbar. Vielmehr sollte darauf geachtet werden, dass möglichst wenig Speicher

verschwendet wird. Unter SharkOberon werden etwa 10% des von der RAM-Disk benötigten Speichers nicht für Nutzdaten verwendet, wenn zur Berechnung alle Dateien des portierten Systems ausser den Quellcode-Dateien herangezogen werden. Ohne Symboldateien sinkt der unbenutze Speicher auf ca. 7%.

4.4.3 Netzwerkunterstützung bei der System-Initialisierung

Während der System-Initialisierung werden nur die benötigten Module zur Netzwerkunterstützung geladen. Dazu gehören:

- CS8900 (der Ethernet-Treiber)
- NetBase
- NetPorts
- NetIP
- NetUDP

Nachdem die RAM-Disk initialisiert worden ist, wird die Netzwerk-Unterstützung wieder entfernt (d.h. der Treiber wird deinstalliert). Sie wird dann im weiteren Verlauf der System-Initialisierung erneut geladen. Obwohl die Vorgehensweise auf den ersten Blick umständlich erscheint, ergeben sich damit folgende Vorteile:

- Im *Inner Core* befinden sich nur die nötigen Module.
- Der Code für Initialisierung des Systems kann weitestgehend ohne Veränderungen von Native Oberon übernommen werden.

Zur Initialisierung der Netzwerkdienste wird wie in `NetSystem.Start` vorgegangen. Die benötigten Informationen (IP-Adresse des Clients, des Servers und des Gateways sowie die Subnet-Maske) können jedoch nicht aus der Datei `Oberon.Text` gelesen werden, da diese ja noch nicht zur Verfügung steht, sondern werden von Open Firmware abgefragt. Ausserdem werden nur die benötigten Protokolle gestartet, d.h. IP und UDP. Die Implementierung ist in Abbildung 4.2 auf der nächsten Seite dargestellt.

```
PROCEDURE StartNet;
VAR route : NetIP.Route; chosen : LONGINT;
BEGIN
    (* CS8900-Device already installed in CS8900-Body, just create route *)
    NEW(route);  route.dev := NetBase.FindDevice(0);
    INCL(route.options, NetIP.arpopt);
    chosen := OFW.FindDevice("/chosen");
    IF OFW.GetProp(chosen, "server-ip", TFTPserver) # NetIP.AdrLen THEN
      Kernel.WriteString("can't get server ip");  HALT(99)
    END;
    IF OFW.GetProp(chosen, "client-ip", route.adr) # NetIP.AdrLen THEN
      Kernel.WriteString("can't get client ip");  HALT(99)
    END;
    IF OFW.GetProp(chosen, "netmask-ip", route.subnet) # NetIP.AdrLen THEN
      Kernel.WriteString("can't get client ip");  HALT(99)
    END;
    IF OFW.GetProp(chosen, "gateway-ip", route.gway) # NetIP.AdrLen THEN
      Kernel.WriteString("can't get gateway ip");  HALT(99)
    END;
    NetIP.InstallRoute(route);
    NetIP.SetDirectedCast(route);
    NetBase.Start;  NetIP.StartIP;  NetPorts.Init;  NetUDP.Start
END StartNet;
```

Abbildung 4.2: Programmfragment „Start der Netzwerkdienste zur RAM-Disk-Initialisierung“

4.5 Struktur des Inner Core

Aufgrund der über das Netz zu initialisierenden RAM-Disk sind neue Module zum *Inner Core* hinzugekommen. Er umfasst nun

- OFW
- Kernel
- Files
- FileDir
- Disk
- NetBase
- NetPorts
- NetIP
- NetUDP
- CS8900
- CRC32
- StreamCompress
- CompressUtil
- SoftFloat (implizit importiert von StreamCompress)
- Modules

Abbildung 4.3 auf der nächsten Seite gibt eine Übersicht über die Abhängigkeiten der Module des *Inner Core*.

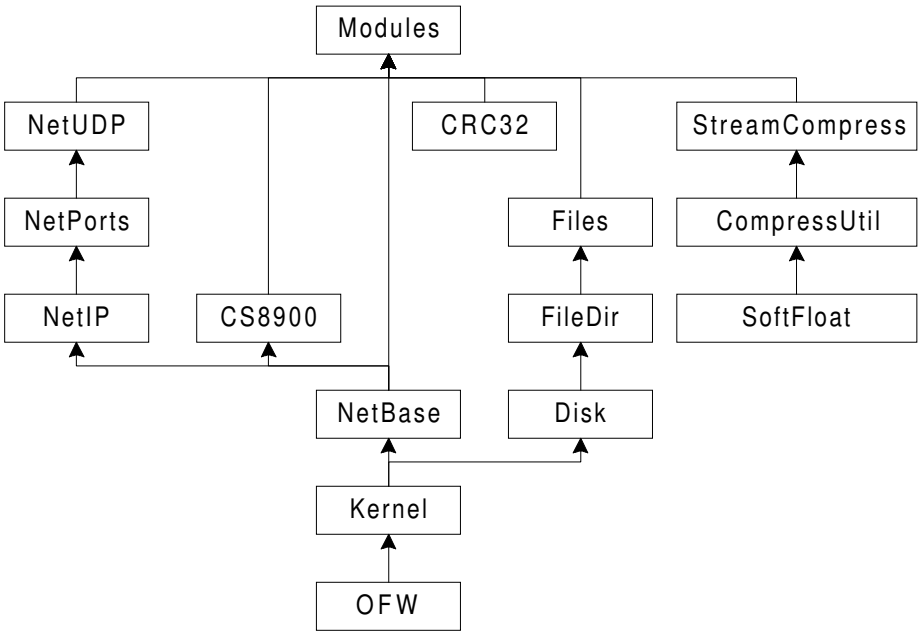


Abbildung 4.3: Module des Inner Core

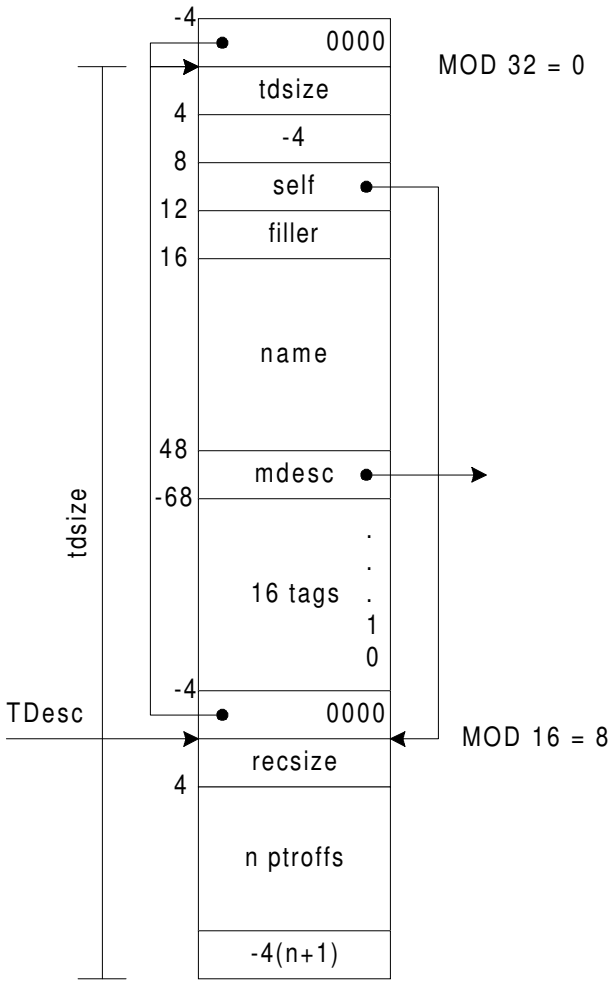


Abbildung 4.4: Struktur des Type Descriptors

4.6 Garbage Collection und Type Descriptors

Der *Garbage Collector* wurde unverändert übernommen. Nur bei den *Type Descriptors* und den *Array Blocks* wurden kleine Änderungen durchgeführt.

4.6.1 Type Descriptor

Beim *Type Descriptor* wurden die Zeiger auf die einzelnen Methoden weggelassen, da diese weder vom Compiler noch vom *Garbage Collector* verwendet werden. Der *Extension Level* wurde ebenfalls weggelassen und durch das neue Feld „filler“ ersetzt, um die Alignment-Bedingung „MOD 16 = 8“ einzuhalten. Die übrigen Felder haben die gleiche Bedeutung wie bei Native Oberon. Die neue Struktur des *Type Descriptors* ist in Abbildung 4.4 dargestellt.

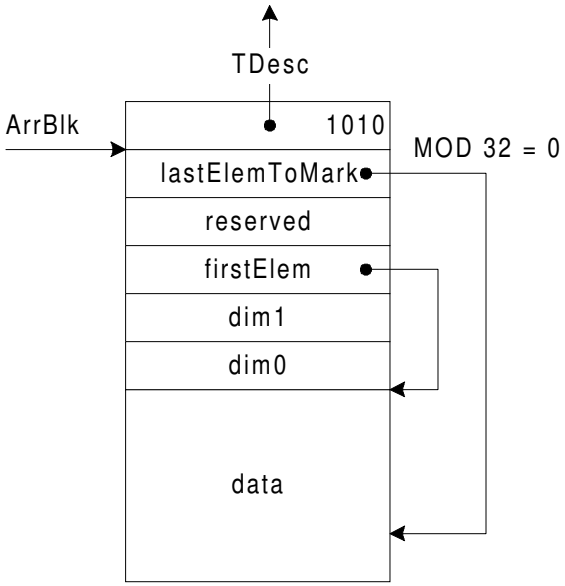


Abbildung 4.5: Struktur des Array Blocks

4.6.2 Array Block

Beim *Array Block* wurde die Bedingung fallengelassen, dass die Daten an einer durch 8 teilbaren Adresse beginnen müssen. Diese Bedingung wurde ursprünglich nur eingeführt, damit auch `LONGREALs` korrekt aligned gespeichert werden können. Da SharkOberon aber `LONGREALs` immer als zwei Words betrachtet, reicht eine durch 4 teilbare Adresse, was immer garantiert ist. Abbildung 4.5 zeigt die Struktur des neuen *Array Blocks*.

4.7 Änderungen im Netzwerk-Subsystem

Es wurde versucht, das Netzwerk-Subsystem möglichst unverändert zu lassen. Es waren jedoch Anpassungen im Verfahren notwendig, das angewendet wird, um die IP-Einstellungen des Systems in Erfahrung zu bringen.

Native Oberon liest die IP-Einstellungen (IP-Adresse, Hostname, Subnet-Maske und Gateway-IP) aus der Datei `Oberon.Text`. Dieses Verfahren kann unter SharkOberon im allgemeinen nicht verwendet werden, da die IP-Adresse des Rechners nicht notwendigerweise bei jedem Start gleich ist (DHCP verteilt IP-Adressen dynamisch). Ausserdem verwendet in den meisten Fällen jeder Rechner im Netz die gleiche Datei `Oberon.Text`. Es wäre zwar möglich, jedem Rechner eine eigene Version zur Verfügung zu stellen, aber das würde eine erheblich umständlichere Administration zur Folge haben, da dann für jeden Rechner eine spezielle `RAMDisk.Content`-Datei erstellt werden müsste.

Aus diesen Gründen verwendet SharkOberon ein anderes Verfahren: Zuerst wird versucht, alle benötigten Einstellungen aus der Sektion Net-System.Hosts.Route0 zu lesen. Sind sie dort nicht vorhanden, so verwendet SharkOberon für die IP-Adresse, die Subnet-Maske sowie den Gateway die

```
GetEntry(key,"Host",hostname,num); ToHost0(num,route.adr,done);
IF ~done THEN
  (* get IP from OpenFirmware and look up hostname in HostNames *)
  IF OFW.GetProp(chosen,"client-ip",route.adr)#NetIP.AdrLen THEN
    HALT(99)
  END;
  ToNum(SYSTEM.VAL(LONGINT,route.adr),num);
  FindHostName("NetSystem.Hosts.HostNames", num, hostname);
  IF hostname[0]#0X THEN done:=TRUE END;
  log := log OR done;
END;
...
GetEntry(key,"Netmask",dmy,num); ToHost0(num,route.subnet,done);
IF ~done THEN (* ok if not arp, e.g. SLIP or PPP *)
  IF NetIP.arpopt IN route.options THEN
    (* Get netmask from Open Firmware *)
    IF OFW.GetProp(chosen,"netmask-ip",route.subnet)#NetIP.AdrLen THEN
      NetBase.Copy(anyIP, route.subnet, NetIP.AdrLen)
    END
  ELSE
    (* all destinations local *)
    NetBase.Copy(anyIP, route.subnet, NetIP.AdrLen)
  END
END;
```

Abbildung 4.6: Programmfragment „Auslesen der Netzwerkeinstellugnen“

von Open Firmware gelieferten Einstellungen. Ist der Hostname nicht angegeben, so wird er in NetSystem.Hosts.HostNames gesucht, wobei die IP-Adresse des Rechners als Schlüssel dient. Abbildung 4.6 verdeutlicht dieses Vorgehen.

Einträge in NetSystem.HostNames bestehen immer aus String-Paaren, welche durch ein Komma getrennt werden. Der erste String ist dabei die IP-Adresse, der zweite String der Hostname des entsprechenden Rechners. Ein Beispiel für eine solche HostNames-Sektion ist in Abbildung 4.7 auf der nächsten Seite dargestellt.

Im Normalfall sollten also in der Sektion NetSystem.Hosts.Route0 nur das Device und der Mode angegeben werden. Alle anderen Einstellungen sollten die Werte von Open Firmware und NetSystem.Hosts.HostNames verwenden.

4.8 Konfigurationsvariablen

Der Oberon-Kernel ist bis zu einem gewissen Grad konfigurierbar. So ist zum Beispiel die Grösse der RAM-Disk, des Heaps oder des User-Stacks nicht „fest verdrahtet“ (oder auch *hard coded*), sondern kann mittels Konfigurationsvariablen eingestellt werden.

```
NetSystem = {
  Hosts = {
    ...
    HostNames = {
      "10.69.1.5", "gandalf.ssp.ch"
      "10.69.1.4", "bilbo.ssp.ch"
    }
  }
}
```

Abbildung 4.7: Beispiel einer NetSystem.Hosts.HostNames-Sektion

Um einerseits das Setzen und Löschen von Konfigurationsvariablen für den Benutzer einfach zu halten und andererseits eine einfache Implementation zu ermöglichen, werden die von Open Firmware zur Verfügung gestellten Dienste genutzt (siehe auch Kapitel 2.2 auf Seite 7). Konfigurationvariablen werden dabei mit **setenv** gesetzt und mit **unsetenv** gelöscht. Eine ausführliche Beschreibung dieser Befehle findet sich in [Fir98]. Eine Übersicht über alle unterstützten Konfigurationsvariablen gibt Anhang E auf Seite 65.

4.9 Gleitkommaunterstützung

Zur Unterstützung von Gleitkommaoperationen wurde die frei verfügbare und portable Bibliothek „SoftFloat“ von J. Hauser verwendet [Hau]. Es wurde versucht, die in C geschriebene Bibliothek „von Hand“ nach Oberon zu portieren. Da sich dies als äussert aufwendig erwies (weil SoftFloat extensiven Gebrauch von Bitoperationen und dem schwachen Typsystem von C macht), wurde ein anderer Ansatz gewählt: SoftFloat wurde mit `gcc` unter NetBSD compiliert. Der daraus resultierende Assembler-Quellcode (ca. 10'000 Zeilen) wurde mit einem Perl-Script in eine Form gebracht, die der SharkOberon-Assembler akzeptiert. Danach mussten noch einige wenige Anpassungen von Hand durchgeführt werden, zum Beispiel die Programmierung von Wrapper-Prozeduren aufgrund der unterschiedlichen *Calling Conventions* des GNU-C-Compilers und des SharkOberon-Compilers. In Abbildung 4.8 auf der nächsten Seite ist das Interface von Oberon-SoftFloat dargestellt.

```
DEFINITION SoftFloat;

  CONST
    TininessAfterRounding = 0; TininessBeforeRounding = 1;
    RoundNearestEven = 0; RoundDown = 1; RoundUp = 2; RoundToZero = 3;
    Invalid = 1; DivByZero = 2; Overflow = 4; Underflow = 8; Inexact = 16;

  VAR
    RoundingMode, ExceptionFlags, DetectTininess: SHORTINT;
    OverflowExceptions: BOOLEAN;

  PROCEDURE LIntToReal (i: LONGINT): REAL;
  PROCEDURE LIntToLReal (i: LONGINT): LONGREAL;
  PROCEDURE RealToLInt (r: REAL): LONGINT;
  PROCEDURE RealToLReal (r: REAL): LONGREAL;
  PROCEDURE RealRoundToInt (r: REAL): REAL;
  PROCEDURE RealAdd (r1, r2: REAL): REAL;
  PROCEDURE RealSub (r1, r2: REAL): REAL;
  PROCEDURE RealMul (r1, r2: REAL): REAL;
  PROCEDURE RealDiv (r1, r2: REAL): REAL;
  PROCEDURE RealRem (r1, r2: REAL): REAL;
  PROCEDURE RealSqrt (r: REAL): REAL;
  PROCEDURE RealEq (r1, r2: REAL): BOOLEAN;
  PROCEDURE RealLe (r1, r2: REAL): BOOLEAN;
  PROCEDURE RealLt (r1, r2: REAL): BOOLEAN;
  PROCEDURE RealNe (r1, r2: REAL): BOOLEAN;
  PROCEDURE RealGe (r1, r2: REAL): BOOLEAN;
  PROCEDURE RealGt (r1, r2: REAL): BOOLEAN;
  PROCEDURE LRealToLInt (r: LONGREAL): LONGINT;
  PROCEDURE LRealToReal (r: LONGREAL): REAL;
  PROCEDURE LRealRoundToInt (r: LONGREAL): LONGREAL;
  PROCEDURE LRealAdd (r1, r2: LONGREAL): LONGREAL;
  PROCEDURE LRealSub (r1, r2: LONGREAL): LONGREAL;
  PROCEDURE LRealMul (r1, r2: LONGREAL): LONGREAL;
  PROCEDURE LRealDiv (r1, r2: LONGREAL): LONGREAL;
  PROCEDURE LRealRem (r1, r2: LONGREAL): LONGREAL;
  PROCEDURE LRealSqrt (r: LONGREAL): LONGREAL;
  PROCEDURE LRealEq (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE LRealLe (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE LRealLt (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE LRealNe (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE LRealGe (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE LRealGt (r1, r2: LONGREAL): BOOLEAN;
  PROCEDURE RealNeg (r: REAL): REAL;
  PROCEDURE LRealNeg (r: LONGREAL): LONGREAL;
  PROCEDURE RealEntier (r: REAL): LONGINT;
  PROCEDURE RealAbs(r : REAL) : REAL;
  PROCEDURE LRealEntier (r: LONGREAL): LONGINT;
  PROCEDURE LRealAbs(r : LONGREAL) : LONGREAL;

END SoftFloat.
```

Abbildung 4.8: Das Interface von SoftFloat

Kapitel 5

Performance-Messungen

Zur Performance-Messung wurden folgende Benchmarks durchgeführt:

- Dhrystone [Wei84]
- Hennessy
- DisplayBench (selbst geschrieben)

Vergleichsmessungen wurden auf einem PC unter Native Oberon V2.3.1 durchgeführt. Die CPU des PCs ist ein Intel Celeron 300A (300 MHz), als Grafikkarte kam eine Diamond Viper V550 (Riva TNT Chip) zum Einsatz.

Da die Benchmarks auf verschiedenen Maschinen ausgeführt wurden, sagen die hier präsentierten Ergebnisse (mit Ausnahme des Dhrystone Benchmark) nicht viel über die Qualität des vom SharkOberon-Compiler erzeugten Codes aus. Sie dienen vielmehr dazu, ein „Gefühl'" für die Geschwindigkeit von SharkOberon zu vermitteln. Für einen besseren Überblick sind neben den absoluten Werten jeweils auch die relativen angegeben.

5.1 Dhrystone Benchmark

Der Dhrystone-Benchmark misst die Geschwindigkeit mittels einer repräsentativen Mischung von Zuweisungen (58%), Control Statements (27%) und Prozeduraufrufen (15%). Die Ergebnisse müssen kritisch betrachtet werden, da sie nur einen kleinen Teil des Compilers und der CPU testen. Abbildung 5.1 auf der nächsten Seite zeigt die Resultate. Die C-Versionen des Benchmarks wurden auf dem NC unter NetBSD ausgeführt und mit `gcc` compiliert. Bei der optimierten Version wurde die Compileroption `-O2` verwendet. Auf dem PC wurden sie unter Windows 98 ausgeführt und mit Visual C++ 5 compiliert.

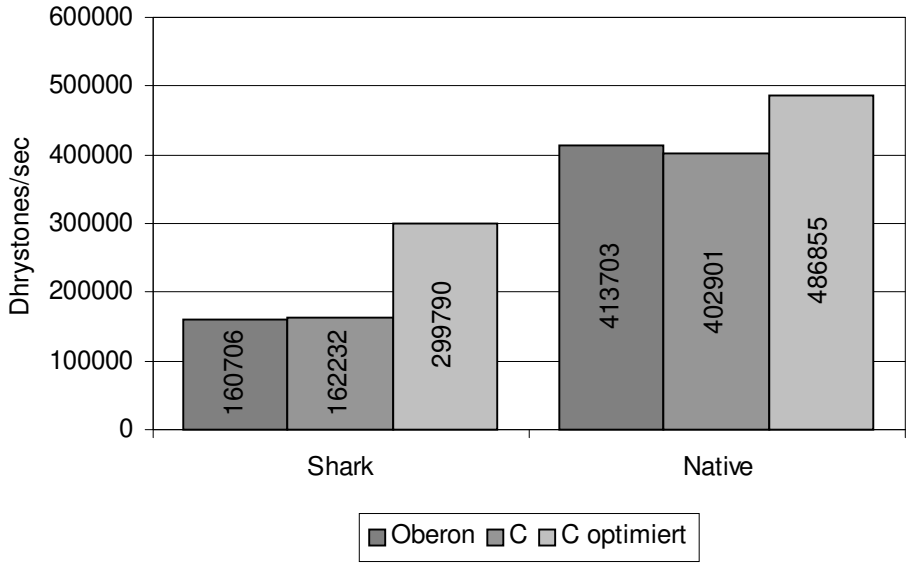


Abbildung 5.1: Dhrystone Benchmark
höhere Werte besser

5.2 Hennessy Benchmark

In der Abbildung 5.2 auf der nächsten Seite sind die Resultate des Hennessy-Benchmarks dargestellt. Die Ergebnisse der FFT wurden nicht in das Diagramm hineingenommen, da sich ihre Resultate so stark unterscheiden, dass die anderen Werte kaum mehr erkennbar waren. Die grossen Unterschiede im FFT-Test erklären sich durch die Tatsache, dass der Intel-Rechner über einen Gleitkommakoprozessor verfügt, während beim NC Gleitkommaoperationen durch Software implementiert sind. Der Vollständigkeit halber sind die Ergebnisse des FFT-Tests in Tabelle 5.1 aufgeführt.

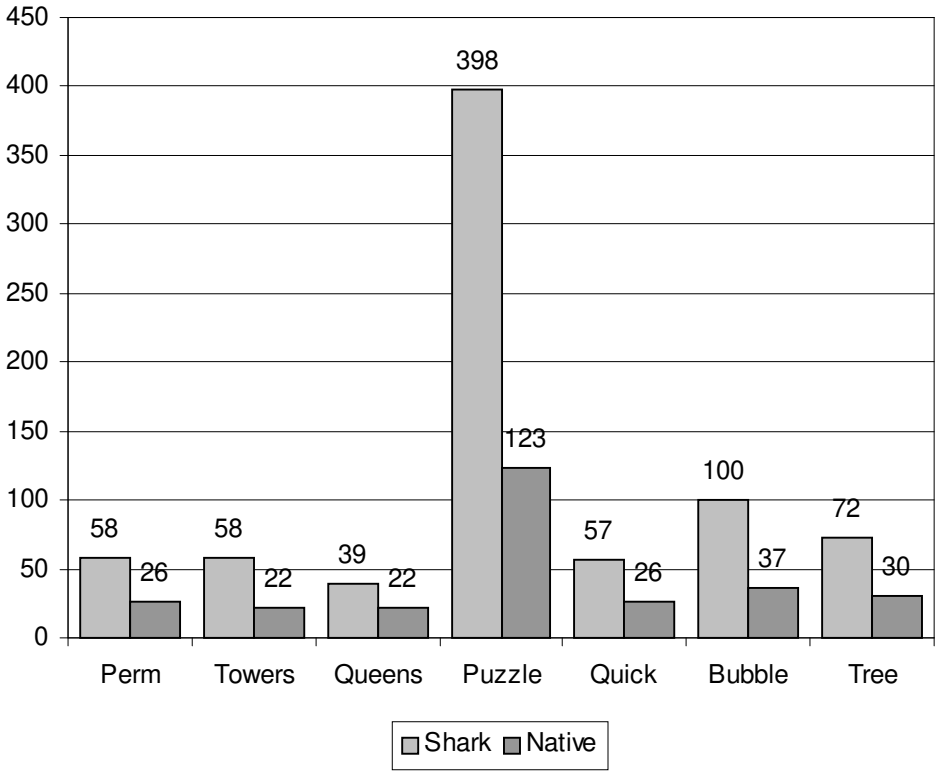
	Shark	Native
FFT	1526	40

Tabelle 5.1: Hennessy-Benchmark (nur FFT)

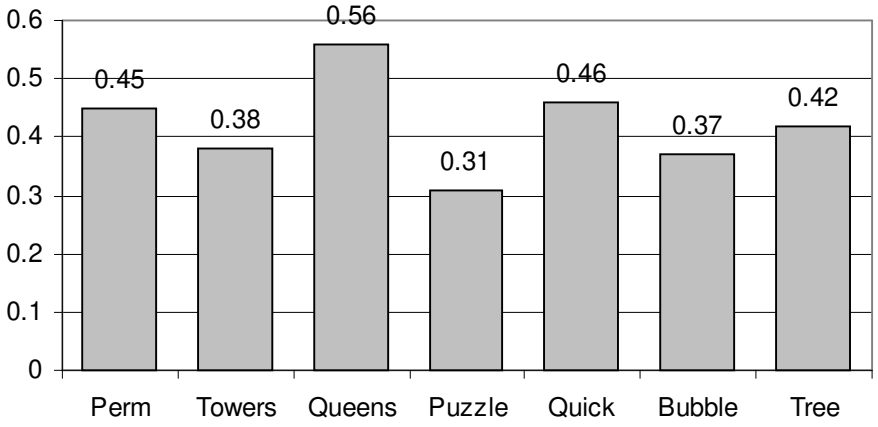
5.3 Display Benchmark

Der Display-Benchmark misst die Geschwindigkeit der Display-Prozeduren, die bei der Arbeit mit Oberon am häufigsten benutzt werden (Darstellung von Text, Verschieben von Viewers, etc.). Auf dem PC wurde dabei das generische SVGA-Display-Modul eingesetzt. Das SharkOberon-Display-Modul verwendet die auf dem Grafikadapter vorhandenen Beschleuniger-Hardware ebenfalls nicht. Die Auflösung betrug sowohl auf dem PC als auch dem NC 1024x768 Pixel bei 256 Farben. Die Ergebnisse sind in Abbildung 5.3 auf Seite 47 dargestellt.

Betrachtet man die Werte genauer, so macht man eine erstaunliche Entdeckung: Im Vergleich zum NC werden auf dem PC alle Operationen fast doppelt so schnell ausgeführt. Eine Ausnahme machen dabei Operationen im „invert“-Modus, die auf dem NC sogar schneller sind! Leider konnten weder ich noch mein Betreuer dafür eine Erklärung finden. Unterschiedliche Geschwindigkeiten beim Lesen und Schreiben ins Video-RAM (Pieter Muller stellte fest, dass das Lesen bei gewissen Karten bis zu 10x langsamer ist!) können ausgeschlossen werden, da dann auch CopyBlock davon betroffen wäre.

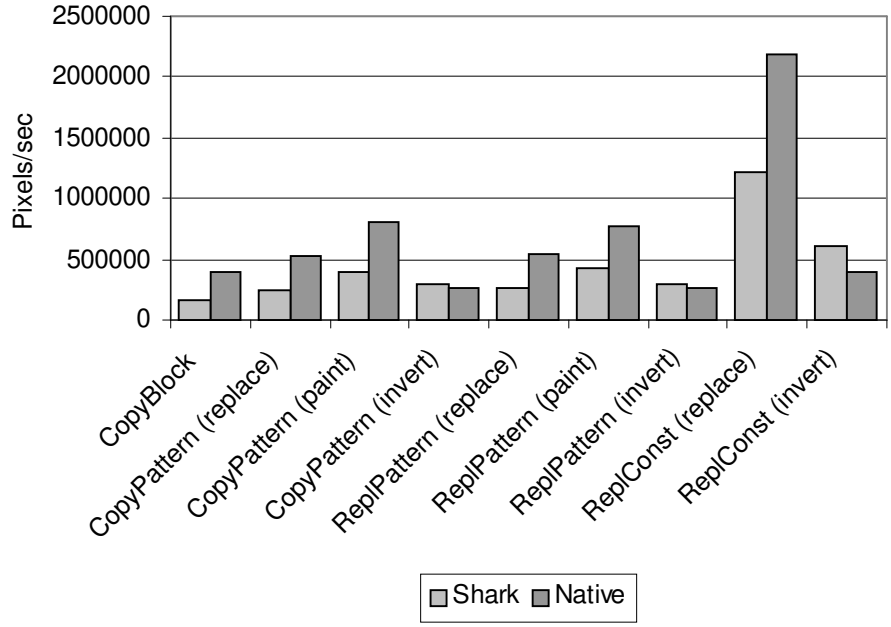


(a) Absolute Werte (tiefere Werte besser)

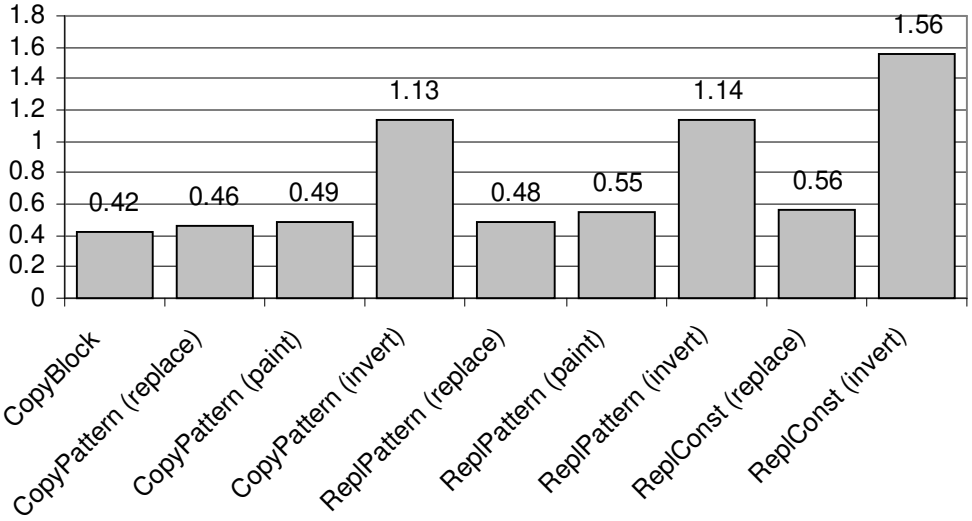


(b) relative Geschwindigkeit von SharkOberon (Native Oberon = 1)

Abbildung 5.2: Hennessy Benchmark (ohne FFT)



(a) Absolute Werte (höhere Werte besser)



(b) relative Geschwindigkeit von SharkOberon (Native Oberon = 1)

Abbildung 5.3: Display Benchmark

Kapitel 6

Notwendige Änderungen am Native Quellcode

In diesem Kapitel werden die Änderungen am Native Quellcode besprochen, die im Rahmen der Portierung durchgeführt werden mussten. Dabei geht es jedoch nicht um Module wie den Kernel oder Gerätetreiber, sondern um Module, die eigentlich portabel sein sollten. In den meisten Fällen war einer der folgenden Punkte der Grund für die Nicht-Portabilität:

Optimierungen
Oftmals wurden kleine Prozeduren aus Effizienzgründen in i386-Assembler geschrieben.

Annahmen über den Zielprozessor
In einigen Modulen wurde von der Annahme ausgegangen, dass Lese- oder Schreibzugriffe auf nicht-alignierte Adressen möglich sind.

Optimierungen mittels Verwendung von Assembler-Code sind problematisch. Einige wenige, einfache Prozeduren sind sicher vertretbar, aber es sollte nicht übertrieben werden. Das Modul „Bitmaps“ zum Beispiel bestand grösstenteils aus komplexen Assembler-Prozeduren. Es konnte deshalb nur mit grossem Aufwand portiert werden, obwohl es sonst plattformunabhängig ist. Auf solche ummfassenden Optimierungen sollte verzichtet werden, da die Portabilität darunter stark leidet.

Annahmen über den Zielprozessor sind zwar relativ leicht zu beheben, können aber nicht immer einfach gefunden werden. So wird zum Beispiel in den Netzwerk-Modulen an einigen Stellen ein Zeiger auf einen Record durch einen *Type Cast* gesetzt, wobei korrektes Alignment nicht garantiert ist. Solche Fehler können zur Compilierzeit nicht mit Sicherheit erkannt werden und führen zur Laufzeit zu einem *Data Abort*. Durch eine sorgfältigere Programmierung hätte dies vermieden werden können.

Zusammenfassend sind die Änderungen, die durchgeführt werden mussten, in der Tabelle 6.1 auf der nächsten Seite aufgelistet.

Modul	Änderungen
Fonts	Assembler-Prozeduren ersetzt/entfernt Alignment-Probleme behoben
Math MathL Bitmaps NetBase	Assembler-Prozeduren ersetzt/entfernt
Pictures	Assembler-Prozeduren ersetzt/entfernt
NetUDP NetTCP NetIP Dim3Base FontEditor CRC32	Alignment-Probleme behoben

Tabelle 6.1: bei der Portierung notwendige Änderungen an portablen Modulen

Kapitel 7

Adaption des Systems an eine 2-Tasten-Maus

Optional sollte im Rahmen dieser Diplomarbeit das Oberon System 3 an eine 2-Tasten-Maus adaptiert werden. Eine möglich Adaption wird in diesem Kapitel beschrieben.

7.1 Die bestehende Lösung

Das bestehende System verwendet als Eingabegerät eine 3-Tasten-Maus. Obwohl eine 2-Tasten-Maus als Eingabegerät im Prinzip nicht vorgesehen ist, kann sie trotzdem benutzt werden; die mittlere Maustaste wird in diesem Fall vom System durch die Control-Taste emuliert. Diese Lösung ist aber nicht befriedigend, da die mittlere Maustaste unter Oberon häufig benutzt wird. Durch den ständigen Wechsel zwischen Maus und Tastatur wird die Arbeit mit dem System erschwert.

7.2 Das neue Verfahren und seine Implementation

Es wurde ein Weg gesucht, das *Look and Feel* des Oberon-Systems beizubehalten und gleichzeitig den Wechsel zwischen Maus und Tastatur zu minimieren. Das neue Verfahren verwendet deshalb immer noch das Modell eines Eingabegeräts mit drei unabhängigen Tasten. Die mittlere Maustaste wird dabei allerdings durch einen *Doppelklick links* emuliert. Die Kombinationen „Mitte – Interklick links“, „Links – Interklick Mitte“ und „alle Maustasten“ sind somit aber nicht mehr nur durch Mausaktionen zu erreichen; deshalb muss in diesen Fällen weiterhin die Control-Taste der Tastatur verwendet werden, um die linke (bei „Mitte – Interklick links“) bzw. die mittlere Maustaste (bei „Links – Interklick Mitte“) zu emulieren. Die erwähnten Klickkombinationen werden bei der Arbeit mit Oberon nur selten benötigt, weshalb eine Verwendung der Tastatur den Arbeitsablauf nicht allzusehr beeinträchtigt.

Um das neue Verfahren zu aktivieren, muss die Konfigurationsvariable **DCT** auf einen Wert > 0 gesetzt werden. Der Wert gibt die Zeitspanne in Millisekunden an, innerhalb welcher zwei Linksklicks als Doppelklick erkannt werden,

Die Implementation gestaltet sich relativ einfach: Falls der Maustreiber das Ereignis „Linke Maustaste gedrückt“ meldet, wird während der Zeitspanne, innerhalb welcher ein Doppelklick möglich ist, die Maus weiterhin abgefragt (*polling*). Wird ein weiterer Linksklick erkannt, so wird dem Caller das Ereignis „Mittlere Maustaste gedrückt“ gemeldet. Tritt kein weiterer Linksklick auf, wird ein gewöhlicher Linksklick gemeldet. Die Implementation des neuen Verfahrens ist in Abbildung 7.1 auf der nächsten Seite gezeigt.

7.3 Kritik

Das obige Verfahren besitzt eine Schwachstelle: Bei einem gewöhnlichen Linksklick muss das System eine gewisse Zeit warten, bevor es das Ereignis behandeln kann, da ja eventuell ein Doppelklick erfolgen könnte. Obwohl diese Zeitspanne typischerweise kurz ist (0.2 sec sind ausreichend), wird sie vom Anwender deutlich wahrgenommen, vor allem beim Setzen des Carets.

Dieses Problem lässt sich meiner Meinung nach ohne eine grundlegende Überarbeitung des Systems nicht lösen, da der erste Linksklick eines Doppelklicks unter keinen Umständen gemeldet werden darf. Dies hätte nämlich zur Folge, dass zum Beispiel das Caret an eine neue Position gesetzt würde, und somit könnten auch die Funktionen zum Löschen und Kopieren von Text nicht mehr korrekt arbeiten.

```
PROCEDURE GetMouseEvent(VAR keys: SET;  VAR dx, dy: INTEGER): BOOLEAN;
VAR
  res, res2 : BOOLEAN;
  t : LONGINT;
  keys2 : SET;
  state, dx2, dy2 : INTEGER;
BEGIN
  IF buffered THEN
    keys:=bufKeys; dx:=0; dy:=0; buffered:=FALSE; res:=TRUE;
  ELSE
    res:=GetMouseEvent0(keys,dx,dy)
  END;
  IF (dblClickTime > 0) & res THEN
    IF 2 IN keys THEN
      IF wasDblClick THEN (* dbl click, not released -> keep middle key set *)
        INCL(keys,1);  EXCL(keys,2);
      ELSIF ~leftWasDown THEN
        (* Wait, may be double click *)
        leftWasDown := TRUE;  wasDblClick := FALSE;
        state := 0;
        t := Kernel.GetTimer() + dblClickTime;
        WHILE (Kernel.GetTimer() < t) & (state # 2) DO
          res2:=GetMouseEvent0(keys2, dx2, dy2);
          INC(dx,dx2);  INC(dy,dy2);
          IF state = 0 THEN (* not released *)
            IF res2 & ~(2 IN keys2) THEN
              leftWasDown : =FALSE;  state := 1
            END
          ELSIF state = 1 THEN (* released, not repressed *)
            IF res2 & (2 IN keys2) THEN
              wasDblClick := TRUE;  state := 2
            END
          END
        END;
        IF wasDblClick THEN INCL(keys,1);  EXCL(keys,2) END;
        IF ~(leftWasDown OR wasDblClick) THEN (* buffer "left-up" event *)
          bufKeys := keys - {2};  buffered := TRUE
        END;
        res := TRUE
      END
    ELSE
      leftWasDown := FALSE;
      wasDblClick := FALSE
    END
  END;
  IF (keys * {1,2} # {}) & (flags * {LCtrl, RCtrl} # {}) THEN
    keys := keys + {1,2}
  END;
  RETURN res
END GetMouseEvent;
```

Abbildung 7.1: Programmfragment „Emulation der mittleren Maustaste“

Kapitel 8

Schlussbemerkungen und Ausblick

Die aktuelle Version von SharkOberon erfüllt die geforderten Bedingungen der Aufgabenstellung. Die optionale Gleitkommaunterstützung wurde implementiert, und es wurde eine Adaption des Systems an eine 2-Tasten-Maus entwickelt. Das System läuft stabil und mit einer akzeptablen Geschwindigkeit. SharkOberon ist aber noch lange nicht fertig. Unter anderem sollten folgende Punkte noch in Angriff genommen werden:

- Implementierung eines optimierenden Compilers, vorzugsweise auf der Basis von Active Oberon.
- Verbesserung des Memory Managements. Aufgund der Probleme beim Allozieren vieler einzelner Seiten sollte auf die Open Firmware Services verzichtet werden.
- Optimierung diverser Module, vor allem „Display“ und „CS8900“.
- Portierung der restlichen Module, die zu Oberon System 3 gehören.

Diese Liste umfasst nur die meiner Meinung nach wichtigsten Punkte und könnte natürlich noch weiter fortgesetzt werden. Mögliche Anwendungsgebiete für NCs mit SharkOberon sind unter anderem:

- Workstations für Studentenlabors
- Spezialanwendungen (wie zum Beispiel Set-Top-Boxen)

Ein Einsatz in Studentenlabors (als Ersatz für die alten Ceres-Labors) ist mit der aktuellen SharkOberon-Version durchaus möglich. Die Administration der Rechner ist äusserst einfach, da jede Workstation mit dem gleichen RAM-Disk-Inhalt arbeiten kann.

In Set-Top-Boxen könnte eine Spezial-Version von SharkOberon zum Einsatz kommen, die über eine eingeschränkte GUI verfügt (nur ein Viewer, der den ganzen Bildschirm ausfüllt), dafür aber von einer ROM-Card booten kann. Dies sollte keine allzu grossen Probleme bereiten, da durch den Einsatz der

Open Firmware prinzipiell von jedem „block device“ gebootet werden kann. Die RAM-Disk könnte dabei beibehalten werden und nicht über das Netz, sondern von der ROM-Card initialisiert werden. Ausserdem sollte SharkOberon noch um einen Soundtreiber erweitert werden und den TV-Ausgang der Grafikkarte unterstützen. Zusammenfassend sind also folgende Erweiterungen notwendig:

- Unterstützung des integrierten Soundchips.
- Unterstützung des TV-Ausgangs des Grafikadapters.
- Änderung der System-Initialisierung (Booten von der ROM-Card)

Die Arbeit an SharkOberon war sehr interessant. Zum ersten Mal hatte ich die Möglichkeit, einen wirklich tiefen Einblick in die System-Programmierung zu erlangen. Mein besonderer Dank gilt meinem Betreuer Pieter Muller für seine hilfreichen Tips, Anregungen und Erklärungen zum System, Patrik Reali für seine Hilfestellungen bei der Portierung des Compilers, John Hauser für SoftFloat, sowie allen Programmierern des NetBSD/arm32-Projekts für ihren meistens hervorragend kommentierten Quellcode.

Anhang A

Cross-Development-Tools

A.1 Compiler, Decoder und Browser

Beim Cross-Compilieren von Modulen muss speziell darauf geachtet werden, dass bestehende Objekt- und Symboldateien nicht von den Versionen für SharkOberon überschrieben werden. Deshalb stellen der Compiler, Decoder und Browser vor den Namen jeder zu lesenden oder schreibenden Datei einen Präfix. Der Präfix wird aus `Oberon.Text` gelesen und ist dort unter ARM-Compiler.Prefix abgelegt. Somit können diese Tools ohne Änderungen sowohl zum Cross-Development als auch zur Entwicklung unter SharkOberon eingesetzt werden. Abbildung A.1 zeigt ein Beispiel für den Eintrag des Präfixes in `Oberon.Text`.

A.2 Boot Linker

Der *Boot Linker* wurde einfach gehalten und bietet nur das Kommando `Link` an. `Link` erwartet als Argument den Namen des Top-Level-Moduls und erzeugt ein *Boot Image* mit dem Namen `oberon`.

A.3 Automatische Generierung von `RAMDisk.Content`

Das Erstellen der Datei `RAMDisk.Content` ist von Hand zwar möglich, aber sehr aufwendig und fehlerträchtig. Deshalb wurde ein Tool „ContentMaker“ entwickelt, mit welchem `RAMDisk.Content` scriptgesteuert erzeugt werden kann. Es

```
ARMCompiler = {
  Prefix = "ARM."
}
```

Abbildung A.1: Beispiel eines ARMCompiler.Prefix-Eintrags

```
CONTENTFILE "Test.Content"
PREFIX "ARM."
FILES
  "ARM.Oberon.Text"
ARCHIVE "ARM.Everything.Arc"
    FILES
        "ARM.*.Obj" "ARM.*.Pict" "ARM.*.Text"  "ARM.*.Data"
        "ARM.*.Pal" "ARM.*.Tool" "ARM.*.Panel" "ARM.*.Desk"
        "*.Scn.Fnt" "ARM.*.Lib"  "ARM.*.Sym"   "ARM.*.poly"
    EXCEPT
        "ARM.Kernel.Obj" "ARM.*Test.Obj" "ARM.*Test.Sym"
        "ARM.Oberon.Text"
END CONTENTFILE
```

Abbildung A.2: Beispiel eines `ContentMaker`-Scripts

werden automatisch Umbennungen durchgeführt und auf Wunsch auch Archive erzeugt. Die Scriptsprache hat folgende Syntax:

ContentDescr	=	**CONTENTFILE** [FileName] **PREFIX** string [Files] [Archive { Archive }] **END CONTENTFILE**.
Archive	=	**ARCHIVE** FileName Files.
Files	=	**FILES** Mask { Mask } [**EXCEPT** Mask { Mask }].
FileName	=	string.
Mask	=	string.

Die erstellte Inhaltsdatei erhält den nach **CONTENTFILE** angegeben Namen. Fehlt dieser, so wird „RAMDisk.Content“ verwendet. Durch **PREFIX** wird der Präfix angegeben, der bei der Erzeugung der Einträge entfernt wird (falls er im Dateinamen vorkommt). Durch **FILES** wird die Liste aller Dateien gebildet, welche in der Inhaltsdatei abgelegt werden. Es werden alle Dateien genommen, welche einer der Suchmasken entsprechen. Die Suchmasken dürfen die von `System.Directory` akzeptierten *Wildcards* enthalten. Alle Dateien, auf die eine Maske nach **EXCEPT** passt, werden wieder aus der Liste entfernt. Archive können durch **ARCHIVE** erzeugt werden. Die angegeben Dateien werden dann im Archiv abgelegt (wobei ebenfalls der Präfix entfernt wird). Das Archiv wird mit der Option „[decompress]“ in der Inhaltsdatei eingetragen.

Die Entwicklung einer Inhaltsdatei wird durch dieses Tool deutlich vereinfacht, vor allem da *Wildcards* verwendet werden können. Ein Beispielscript ist in Abbildung A.2 dargestellt.

Anhang B

Syntax von Symboldateien

Con = 1, Typ = 2, Var = 4, Proc = 6, Pointer = 8, ProcTyp = 9,
Array = 10, DynArr = 11, Record = 12, ParList = 13, ValPar = 14,
VarPar = 15, FldList = 16, Fld = 17, HPtr = 18, Fixup = 20, Mod = 22
Byte = 1, Bool = 2, Char = 3, SInt = 4, Int = 5, LInt = 6,
Real = 7, LReal = 8, Set = 9, String = 10, Nil = 11, NoTyp = 12.

```
SymbolFile  =  0F7X modAnchor { element }.
modAnchor   =  Mod key:4 name.
element     =  Con constant
            |  Typ ref modno name
            |  (Var | Fld) ref offset:4 name
            |  (ValPar | VarPar) ref name offset:4
            |  ParList { element }
            |  Pointer ref mno
            |  Array ref mno size:4
            |  DynArr ref mno size:4 lenoff:4
            |  FldList { element } Record ref mno size:4 dscadr:4
            |  HPtr offset:4
            |  Fixup ref ref
            |  modAnchor.
constant    =  (Byte | Bool | Char | SInt) val name
            |  Int val:2 name
            |  (LInt | Real) val:4 name
            |  LReal val:8 name
            |  String name name
            |  Nil name.
name        =  { char } 0X.
```

Anhang C

Syntax von Objektdateien

```
ObjectFile    =  0F5X 030X refPos:4 modBody:4 nofImports:2
                 nofEntries:2 nofPtrs:2 nofCmds:2
                 stringSize:4 dataSize:4 codeSize:4
                 nofTDesc:2 key:4 name ImportBlock
                 EntryBlock PtrBlock CmdBlock CodeBlock
                 ConstBlock Fixups TDescBlock RefBlock.
ImportBlock   =  085X { key:4 name }.
EntryBlock    =  082X { ofs:4 }.
PtrBlock      =  084X { ptrofs:4 }.
CmdBlock      =  083X { name addr:4 }.
CodeBlock     =  088X { word:4 }.
ConstBlock    =  087X { word:4 }.
Fixups        =  086X extCallFixlist:4 extAddrFixlist:4 fixlist:4 .
TDescBlock    =  089X { TDesc }.
TDesc         =  tagaddr:4 nofPtrs:2 recSize:4 name extLev
                 { baseTyp:4 } { ptrOfs:4 }.
RefBlock      =  08BX { 0F8X POfs name { Vars } }.
Vars          =  Mode Type [ Dim ] VOfs name.
Mode          =  1X | 3X.
Type          =  1X .. 0FX | 81X .. 8EX .
POfs          =  num.
Dim           =  num.
VOfs          =  num.
name          =  { char } 0X.
```

Anhang D

Das Modul SYSTEM

Das Modul SYSTEM enthält alle Typen und Prozeduren, die in [MW95] aufgeführt werden. Ausserdem sind folgende neue Prozeduren hinzugekommen:

HALT(n)
`SYSTEM.HALT(n)` verhält sich wie das gewöhnliche `HALT`-Statement, mit dem Unterschied, dass der Halt-Code kleiner gleich 20 sein darf.

DISABLEINTERRUPTS
Setzt das I-Flag im *Current Processor State Register* (CPSR) und verbietet somit Interrupts. Es liefert das alte I-Flag zurück. Der Rückgabetyp ist `LONGINT`.

ENABLEINTERRUPTS
Löscht das I-Flag im CPSR und erlaubt somit Interrupts.

RESTOREINTERRUPTS(m)
Setzt das I-Flag des CPSR entsprechend dem Zustand in m. `RESTOREINTERRUPTS` wird typischerweise zusammen mit `DISABLEINTERRUPTS` verwendet, wenn nicht davon ausgegangen werden kann, dass die Interrupts vor `DISABLEINTERRUPTS` zugelassen waren.

Anhang E

Konfigurationsvariablen

Variable	Bedeutung	Default
Buttons	Anzahl Maus-Buttons	-3
Color	1 = Farbdisplay 0 = S/W-Display	1
ContentFile	Name des Content-Files	RAMDisk.Content
DCT	Time-Out für Doppelclicks in msec (aktiviert neue 2-Tasten-Maus-Adaption)	
DiskGC	Threshold für Disk-GC	10
DiskSize	Grösse der RAM-Disk in MBytes	8
EscCompat	1 = Neutralisierung via ESC 0 = keine Neutralisierung mit ESC	1
FontConv	1 = Ersetzung Syntax → Oberon 0 = keine Fontersetzung	1
HeapSize	Grösse des Heaps (MBytes)	max.
Keyboard	Name der Keyboard-Tabelle	
MP	Mausport (nur falls Maus # PS2)	1
MT	Maustyp	PS2
ModSize	für Module reservierter Speicher (KBytes)	2048
MouseBPS	Geschwindigkeit des Mausports	1200
MouseMap	Mapping der Buttons	012
MousePort	Alias für MP	
MouseRate	Report-Rate (wird nicht von allen Mäusen unterstützt)	100
MouseType	Alias für MT	
NumLock	1 = NumLock ein 0 = NumLock aus	0
Prefix	Präfix (wird vor jeden Dateinamen gesetzt)	
Speedup	Maus-Speedup	15

Fortsetzung auf der nächsten Seite...

Variable	Bedeutung	Default
StackSize	Grösse des User-Stacks (KBytes)	128
TFTPTimeOut	Time-Out-Zeit bei der Initialisierung der RAM-Disk (sec)	10
Threshold	Speedup Threshold	5
TraceBPS	Geschwindigkeit des Trace-Ports	0
TraceHeap	Traces von Heap-Operationen	0
TraceModules	1 = Modulname ausgeben beim Laden 0 = Modulname nicht ausgeben	0

Tabelle E.1: Konfigurationsvariablen von SharkOberon

Anhang F

Portierte Module

Nachfolgend sind alle portierten Module in alphabethischer Reihenfolge aufgeführt. Die Module, welche angepasst werden mussten, sind durch einen Stern (*) gekennzeichnet. Alle nicht gekennzeichneten Module mussten nur neu compiliert werden. Nicht berücksichtigt werden in dieser Übersicht einige Module des *Inner Cores*, Treiber, sowie die Entwicklungs-Tools (Compiler, Browser, Decoder), da entweder inhärent nicht portabel sind oder aber fast vollständig neu entwickelt wurden.

ASCIITab	Curves	Finger	ListGadgets
AsciiCoder	Dates	FontEditor *	ListModels
Asteroids *	Desktops	Fonts *	ListRiders
Attributes	DiffGadgets	Freecell	Lists *
BIT	Diff	GIF	MD5
BMP	Dim3Base *	Gadgets	MIME
BTrees	Dim3Engine	Gopher	Magnifier
Base64	Dim3Frames	GraphicFrames	Mail
BasicFigures	Dim3Paint	Graphics	Math *
BasicGadgets	Dim3Read	HPCalc	MathL *
BinHex	Directories	HTMLDocs	MenuViewers
Bitmaps *	Display3	HTMLForms	MineSweeper
CRC32 *	Documents	HTMLImages	Miscellaneous
Calc	Draw	HTMLTables	Modules *
CalculatorGadgets	ET	HTTPDocs0	NamePlates
Calculator	EditKeys	HTTPDocs	Navigators
Cards	EditTools	Hex	NetBase *
Clocks	Edit	HyperDocTools	NetDNS
ColorModels	Effects	HyperDocs	NetIP *
ColorTools	FTPTool	ICO	NetPorts
ColorWells	FTP	IFF	NetSystem
Columbus	FileDir	Icons	NetTCP *
CompressTools	Files	In	NetTools
CompressUtil	FindFile	JPEG	NetUDP *
Compress	Find	LPRPrinter	News
Conversions	Finder	Links	Oberon

Objects
Out
Outlines
PCX
PSPrinter
PanelDocs
Panels
Pictures *
Printer3
Printer
ProgressMeters
RXA
RX
RandomNumbers
Reals
Rectangles
RefGadgets
Rembrandt0
RembrandtDocs
RembrandtTools
Rembrandt
ScriptFrames
Script
ScrollViews
Scrollbars
Shanghai
Sisiphus
Snapshot *
Sokoban
Solitaire
Sort
Spider
Splines
Streams
Strings
Styles
System
TGA
TelnetGadgets
Telnet
TerminalFrames
TerminalGadgets
Terminals
Tetris
TextDocs
TextFields
TextFrames
TextGadgets0
TextGadgets
TextMail
TextPopups *
TextStreams
Texts
TimeStamps
UUDecoder
UnZip
V24Gadgets
V24Log
Viewers
Views
XBM

Literaturverzeichnis

[Adv96] Advanced RISC Machines Ltd. *ARM8 Data Sheet*, 1996.

[Ame94] American Megatrends, Inc. *Megakey Keyboard Controller BIOS Technical Reference*, 1994.

[CG85] W. J. Croft and J. Gilmore. RFC 951: Bootstrap Protocol, 1985. `http://www.ietf.org/rfc/rfc951.txt`.

[Cre91] R. Crelier. OP2: A Portable Oberon-2 Compiler. In *Second International Modula-2 Conference*, Loughborough University of Technology, UK, Sept. 1991.

[Cry95] Crystal Semiconductor Corporation, Austin. *CS8900 Ethernet Controller Data Sheet*, 1995.

[Dig96] Digital Equipment Corporation, Maynard. *Digital Semiconductor SA-110 Microprocessor Technical Reference Manual*, 1996.

[Dro97] R. Droms. RFC 2131: Dynamic Host Configuration Protocol, 1997. `http://www.ietf.org/rfc/rfc2131.txt`.

[ESS97] ESS Technology, Inc., Fremont. *ES1887 AudioDrive Data Sheet*, 1997.

[Fir97] FirmWorks, Mountain View. *OpenFirmware Client Interface Developer's Guide*, 1997.

[Fir98] FirmWorks, Mountain View. *OpenFirmware Command Reference*, 1998.

[Hau] J. Hauser. SoftFloat - Software Implementation of the IEC/IEEE Standard for Binary Floating-Point Arithmetic. `http://http.cs.berkeley.edu/~jhauser/arithmetic/softfloat.html`.

[IEE94] IEEE. *IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*, 1994.

[IGS97] IGS Technologies, Santa Clara. *CyberPro 2010 Software Programmer's Guide to Register Definitions*, 1997.

[MW95] H. Mössenböck and N. Wirth. *The Programming Language Oberon-2*. ETH Zürich, 1995.

[Sol92] K. Sollins. RFC 1350: THE TFTP PROTOCOL (REVISION 2), 1992. `http://www.ietf.org/rfc/rfc1350.txt`.

[Wei84] R. P. Weicker. DHRYSTONE: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[WG89] N. Wirth and J. Gutknecht. The Oberon System. *Software – Practice and Experience*, 19(9):857–893, Sept. 1989.

[WG92] N. Wirth and J. Gutknecht. *Projekt Oberon - The Design of an Operating System and Compiler*. ACM Press, New York, 1992.

[Wir88] N. Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7):671–690, July 1988.

[Wir97] N. Wirth. A Computer System for Model Helicopter Flight Control Technical Memo Nr. 2: The Programming Language Oberon SA. Technical Report 285, Institute for Computer Systems, ETH Zurich., 1997.