Portierung eines C-Compilers auf eine capability-basierte Architektur

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von:

Klaus Espenlaub

Gutachter: *Prof. Dr. James Leslie Keedy*
Prof. Dr. Jörg Kaiser
Betreuer: *Dipl.-Ing. Jörg Siedenburg*

1997

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Entwicklung zuverlässiger Software ist heute immer noch ein großes und teilweise ungelöstes Problem. Software wird zunehmend unter Berücksichtigung von Korrektheit, Wartbarkeit und Zuverlässigkeit entworfen. Die Entwicklungsmethoden bleiben jedoch meist auf die Programmiersprachenebene oder gar auf den konzeptuellen Entwurf des Softwaresystems beschränkt.

Die MONADS-Architektur setzt diese Strukturierungskonzepte bis auf die Hardwareebene fort und bietet schon auf Maschinensprachenebene Unterstützung für die Strukturierung von Programmen an. Die Robustheit, Wartbarkeit und Wiederverwendbarkeit von modulorientierten Systemen finden auf diese Weise Unterstützung auf jeder Ebene der Programmentwicklung.

Module erweitern in der MONADS-Architektur das in herkömmlichen Systemen zur Speicherung persistenter Daten benutzte Dateisystem. Module besitzen im Gegensatz zu Dateien rein prozedurale Schnittstellen. Dies schränkt die möglichen Änderungen an der Datenbasis auf die Operationen ein, die von der Schnittstelle angeboten werden. Die Kapselung erleichtert die Implementierung großer Softwaresysteme durch die erzwungene, strikte Aufgabentrennung. Datenschutz und Datensicherheit werden durch das Schutzkonzept der Architektur unterstützt. Der MONADS-PC-Rechner ist eine Implementierung dieser Architektur.

Als Programmiersprache steht auf dem MONADS-PC bisher nur eine stark erweiterte Pascal-Variante zur Verfügung. Andere Programmiersprachen erreichten nie den Punkt, der die Benutzung für normale Programmierer erlaubt hätte.

Es ist daher wünschenswert, weitere Programmiersprachen für den MONADS-PC verfügbar zu machen, um die Zahl der einsetzbaren Programme zu erhöhen. Die Sprache C wird in der Industrie sehr häufig für die Programmentwicklung eingesetzt. Die Erschließung der umfangreichen Codebasis macht C besonders interessant.

Die MONADS-Architektur unterstützt die Erstellung von Compilern für Hochsprachen an vielen Stellen. Es werden die Grundbausteine für abstrakte Datentypen in Form von Segmenten beliebiger Größe angeboten. Das Schutzkonzept ist darauf ausgerichtet, durch strikte Prüfungen möglichst viele Programmfehler zu erkennen. Es gibt meist genau eine Methode, um bestimmte Manipulationen an Daten durchzuführen.

Die Sprache C paßt nicht ohne weiteres in dieses Modell der vorgesehenen Sprachen. C wurde mit dem Ziel der portablen Implementierung hardwarenaher Programme[1] entwickelt. Der architekturelle Schutz ist aber gerade für schwach getypte Sprachen ein Hindernis. Einige Operationen, z.B. die Zeigerarithmetik, haben auf der MONADS-Architektur keine eindeutige Entsprechung.

Die Portierung eines C-Compilers soll jedoch nicht darüber hinwegtäuschen, daß die Sprache C für große Programmsysteme eigentlich wegen ihrer beschränkten Abstraktionsfähigkeit ungeeignet ist.

Die Ziele dieser Diplomarbeit sind:

- Entwurf eines Konzepts für die Portierung eines C-Compilers auf die Architektur und das Betriebssystem des MONADS-PC,
- Implementierung der Maschinenbeschreibung des MONADS-PC für den zu portierenden Compiler (voraussichtlich GNU C) und gegebenenfalls die Korrektur auftretender Compilerfehler,
- Erstellung des C-Laufzeitsystems,
- Entwurf und Implementierung der Entwicklungsprogramme (Assembler, Linker, usw.),
- Durchführung einer prototyphaften Portierung der C-Bibliothek,
- Test des Compilers und Bewertung der Codegüte und
- Analyse der spezifischen Probleme bei der Portierung auf den MONADS-PC, sowohl auf Compilerseite als auch auf Maschinenebene.

Die Erzeugung des C-Compilers für den MONADS-PC erfolgt auf Rechnern des Typs Sun SPARCstation 10 bzw. Sun Ultra 2 mit einem UNIX-Betriebssystem. Es ist vorgesehen, C-Programme auf einem UNIX-System in ausführbare Module für den MONADS-PC zu übersetzen. Die Module können anschließend auf den MONADS-PC übertragen und dort ausgeführt werden. Der Compiler ist also ein Cross-Compiler, der nicht auf dem Zielsystem abläuft. Das Ergebnis der Arbeit soll eine einsatzfähige Entwicklungsumgebung für C-Programme sein.

Es ist deswegen für die Konzeption der Portierung von C unbedingt erforderlich, sich mit der Architektur des Rechners und dem zu portierenden C-Compiler auseinanderzusetzen.

[1] Ein Beispiel für den Einsatzbereich von C ist die Implementierung von Betriebssystemkernen.

Die Portierung ist wegen der ungewöhnlichen Eigenschaften der Prozessorarchitektur nicht ohne genaue Untersuchung aller Alternativen möglich. Der Beweis, daß eine Portierung möglich ist, wurde schon in [11] erbracht. Die dabei implementierte Lösung ist aber aus verschiedenen Gründen nicht befriedigend.

Aufbauend auf diese Portierung des GNU C-Compilers wird untersucht, ob nicht effizientere und für die Programmierer tragbarere Lösungen gefunden werden können. Das wichtigste Ziel dieser Arbeit ist, daß jeder von C-Compilern übersetzbare, portable Quelltext ohne Modifizierung auf dem MONADS-PC läuft. Dies hat natürlich Auswirkungen auf alle oben aufgezählten Ziele der Arbeit. Dieses Hauptziel gibt ein Maß vor, an dem die Lösungsideen gemessen werden können.

Die Lösungsmöglichkeiten müssen neu untersucht werden. Falls die Ziele nicht mit den bekannten Möglichkeiten erreicht werden können, müssen neue gefunden werden. Speziell die Zeigerdarstellung, die Speicherorganisation und die Aufteilung in Module haben starke Wechselwirkungen mit dem Ziel der Compilierbarkeit existierender Quelltexte.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit stellt die Konzeption und Portierung des GNU C-Compilers und den Entwurf und Implementierung der benötigten Hilfsprogramme vor. Sie ist in 8 Kapitel unterteilt.

Die Architektur des MONADS-PC wird in Kapitel 2 vorgestellt, wobei besonderer Wert auf die Unterschiede zu üblichen Rechnerarchitekturen gelegt wird.

Kapitel 3 faßt die wichtigsten Eigenschaften und Möglichkeiten des GNU C-Compilers zusammen. Der Aufbau des Compilersystems und die Auswirkungen auf die gesamte Entwicklungsumgebung werden analysiert.

Der bisher einzige für den MONADS-PC verfügbare Compiler wird in Kapitel 4 vorgestellt. Der MONADS-Pascal-Compiler implementiert einen an die MONADS-Architektur angepaßten Pascal-Dialekt. Es wird ausführlich auf die Repräsentierung der von der Architektur angebotenen Konzepte eingegangen. Anhand eines Beispiels werden die Unterschiede zu Standard-Pascal erläutert.

Ausgehend von den Grundlagen wird im Kapitel 5 untersucht, welche Möglichkeiten es gibt, die Konzepte des C-Compilers auf die MONADS-Architektur abzubilden. Einige existierende Programme nutzen die C-Spezifikation bis an die Grenze aus. Die Lösungsansätze müssen deshalb auf Probleme untersucht werden, die nur in Extremfällen auftreten.

Der Entwurf der Portierung wird im Kapitel 6 so weit verfeinert, daß die Umsetzung in eine Maschinenbeschreibung erfolgen kann. Die Maschinenbeschreibung legt alle für den Compiler relevanten Eigenschaften des Prozessors fest. Die Details der Portierung werden in diesem Kapitel festgelegt.

Auf den Compiler aufbauend wird in Kapitel 7 eine Portierung der C-Bibliothek entwickelt. Die Probleme bei der Implementierung einzelner Funktionen bzw. Funktionsgruppen werden eingehend untersucht. Die plattformabhängigen Teile einiger Funktionen werden implementiert, um den Compiler testen zu können.

In Kapitel 8 schließt sich ein ausführlicher Test des Compilers an. Es werden möglichst viele Testprogramme übersetzt, um möglichst viele Fehler bei der Portierung des Compilers aufzudecken und zu beseitigen. Die Auswertung der Erfahrungen mit der Portierung und der Vergleich mit dem MONADS-Pascal-Compiler stellen die Arbeit mit den vorhergehenden Arbeiten in Zusammenhang und bieten Gelegenheit zur Beurteilung der aufgetretenen Probleme.

Der Ausblick auf zukünftige Arbeiten, die auf diese Diplomarbeit aufbauen können, und die Zusammenfassung schließen in Kapitel 9 die Arbeit ab.

Kapitel 2

Architektur des MONADS-PC

Die Wurzeln der MONADS-Architektur [6] liegen in dem durch die Softwarekrise entstandenen Wunsch, die Qualität von Software zu steigern, die Entwicklungskosten zu senken und Datenschutzansätze zu berücksichtigen. Anders als bei vielen anderen Lösungsversuchen wurde hier keine reine Softwarelösung, sondern eine Kombination von Hardware und Software als Lösungsansatz gewählt.

Wichtige Konzepte der Entwicklung waren:

- eine einheitliche virtuelle und persistente Speicherverwaltung, die u.a. ein konventionelles Dateisystem überflüssig macht. Der Speicher wird auf unterster Ebene seitenorientiert verwaltet. Höhere Ebenen organisieren die Daten in Segmenten.
- eine einheitliche, modulare Struktur des Systems. Programme, Programmmodule, Funktionsbibliotheken, Betriebssystemteile, Dateien usw. sollten durch eine an den Konzepten des Softwareentwurfs angelehnte Implementierungsweise aufgebaut werden. Dies bildet auch die Grundlage des semantischen Schutzes von Daten (Dateisysteme bieten dagegen nur einen vergleichsweise groben Schutz vor unbefugten Zugriffen).
- eine einheitliche Prozeßstruktur mit Unterstützung von Wiedereintrittsfähigkeit und Blockstrukturierung. Prozesse sind langlebiger als in vielen anderen Systemen, bei denen ein Programmstart die Erzeugung eines neuen Prozesses voraussetzt.

Die Architektur sollte Lösungen für die bestehenden Probleme der Softwareentwicklung in konventionellen Systemen aufzeigen. Diese Probleme werden zwar nicht automatisch durch die oben beschriebenen Maßnahmen beseitigt, aber der Zwang zur Benutzung guter Softwareentwurfsmethoden führt in der Regel in die richtige Richtung.

Um die Konzepte in der Realität untersuchen zu können, wurde ein Rechner gebaut: der MONADS-PC. Der Betriebssystemkern des MONADS-PC ist verantwortlich dafür, die Persistenz der Daten zu gewährleisten und die Abfolge der

Prozesse zu steuern. Das Betriebssystem selbst ist modular aufgebaut. Der Kern ist natürlich ebenso ein spezielles Modul.

2.1 Prozesse

Jedem Prozeß ist ein Prozeßstack zugeordnet. Der Stack beinhaltet Informationen über laufende Unterprogrammaufrufe, lokale Variablen und Zwischenwerte von Berechnungen. Der Stack besteht somit aus einer Folge von Stackrahmen, die den Berechnungsstand und die Blockstrukturierung wiederspiegeln. Ein Stack kann potentiell sehr groß werden, in der Implementierung auf dem MONADS-PC 256 MBytes. Der in Systemen mit begrenzten Stacks bestehende Zwang, lokale Variablen in globale umzuwandeln oder auf den Heap auszulagern, entfällt.

Stacks sind wie alle Daten in der MONADS-Architektur persistent, das bedeutet, daß ein Benutzer beim Abmelden vom System nur den Prozeß inaktiviert. Somit ist es möglich, daß man an der Stelle, in exakt dem Zustand, in dem man die Sitzung beendet hat, zu einem späteren Zeitpunkt wieder fortfahren kann.

Prozesse und Module sind voneinander unabhängige Konzepte. Der Code eines Moduls kann gleichzeitig von mehr als einem Prozeß ausgeführt werden und ein Prozeß kann beliebige Module benutzen. Die Lösung der dabei auftretenden Synchronisationsprobleme ist die Aufgabe des Moduls.

2.2 Objektmodell

Objekte[2] auf dem MONADS-PC besitzen alle den gleichen Aufbau: Ein die Daten kapselndes Modul mit Zugriffsfunktionen. Die internen Daten sind außerhalb eines Moduls nicht sichtbar und können nur durch die Benutzung der Schnittstellen weitergegeben werden. Der Entwurf von Software mit datenkapselnden Modulen ist mittlerweile weit verbreitet. Diese Strukturierung des Entwurfs hat Vorteile bei der Verläßlichkeit, Wartbarkeit und Erweiterbarkeit des Softwaresystems. Die MONADS-Architektur bietet deshalb Unterstützung für die Umsetzung des Modulkonzepts schon auf Maschinenebene.

Alle Programmstrukturen werden auf dieses Modulkonzept abgebildet. In klassischen Programmiersprachen implementierte Programme besitzen konzeptuell nur eine Schnittstellenfunktion, nämlich „Programm starten“. Funktionsbibliotheken wiederum besitzen meist keine gekapselten Daten, bieten jedoch eine Vielzahl von Schnittstellenfunktionen an. Bei allen Varianten bilden die gekapselten Daten die globalen Variablen in den Modulen.

Dateien können ebenfalls mit dieser Struktur dargestellt werden. Die dauerhaft gespeicherten Daten einer Datei stellen hier die gekapselten Informationen dar.

[2] Objekte werden in der MONADS-Terminologie oft „Module“ genannt.

Zugriff auf die Daten ist jedoch nur über die Schnittstelle möglich. So können Zugriffe leicht überwacht werden.

Die gleichen Konzepte werden auf das Betriebssystem angewandt. Die Betriebssystemkomponenten werden durch einzelne Module implementiert. Der in vielen Betriebssystemen benutzte monolithische Aufbau kann durch Zerlegung in verschiedene Aufgabengruppen mit definierten Schnittstellen vermieden werden.

Alle Objekte mit der gleichen Schnittstelle bilden eine Klasse. Es kann mehrere Implementierungen jeder Klasse geben. Die Implementierung einer Klasse wird als Typemanager bezeichnet. Typemanager bieten als einzige Schnittstellenfunktion die Erzeugung einer neuen Instanz an. Jede Instanz besitzt ihren eigenen Datenbestand, benutzt jedoch den Code des Typemanagers. Bei Implementierungen, die keine Instanzdaten besitzen, kann diese Erzeugung von Instanzen entfallen und die Schnittstellen sind direkt aufrufbar.

Wegen des durch den Schutzmechanismus relativ hohen Aufrufaufwands sind Module in der MONADS-Architektur für relativ große Strukturen gedacht. Dies resultiert aus der Zielsetzung der Entwicklung: Module sind als Verallgemeinerung des Dateikonzepts entworfen worden. So wie Dateien für die Speicherung von sehr kleinen Datenmengen benutzt werden können, dabei aber meist ineffizient sind, so sind Module auf dem MONADS nicht für einfache Datenstrukturen ausgelegt. Dies würde auch dem über die semantische Schnittstelle möglichen Schutz der Daten widersprechen.

2.3 Lebensdauer von Daten

In einem persistenten System gibt es verschiedene Anforderungen an die Lebensdauer von Daten. Die lokalen Variablen einer Funktion besitzen oft nur eine kurze Lebensdauer, jedenfalls eine kürzere als Instanzdaten eines Moduls. In der MONADS-Architektur gibt es vier Gruppen von Daten, die unterschiedliche Lebensdauer und Sichtbarkeit haben:

Klassenvariablen (type related): Klassenvariablen sind allen Instanzen eines Typemanagers gemeinsam. Die Lebensdauer erstreckt sich bis zum Löschen aller Instanzen und des Typemanagers einer Klasse. Aus Sicherheitsgründen dürfen in der derzeitigen Implementierung des MONADS-PC Klassenvariablen nur gelesen werden, sie sind also auf Konstanten beschränkt.

Instanzvariablen (instance): Alle Benutzer einer Modulinstanz teilen die Instanzvariablen. Ihre Lebensdauer ist an die Existenz der Instanz geknüpft.

Benutzungsbezogene Variablen (retained): Solche Variablen sind fest einem Prozeß zugeordnet, der ein bestimmtes Modul benutzt. Die Variablen können nur vom erzeugenden Prozeß zugegriffen werden, jedoch nur während der Ausführung von Code in dem Modul. Die Lebensdauer erstreckt sich über mehrere Aufrufe an das Modul.

Lokale Variablen (local): Lokale Variablen werden beim Aufruf einer Funktion angelegt und mit dem Rücksprung wieder freigegeben. Dies erlaubt eine effiziente Verwaltung lokaler Variablen im Rahmen des aufrufenden Prozeßstacks.

Diese Strukturierung erlaubt z.B. eine einfache Implementierung von Dateien. In der Instanz werden die eigentlichen Informationen abgelegt. Darüberhinaus ist es oft unumgänglich, die aktuelle Lese- oder Schreibposition über einzelne Aufrufe hinweg zu erhalten. Die Trennung zwischen verschiedenen Benutzern läßt sich gewährleisten, wenn für jeden ein eigener Dateizeiger in den benutzungsbezogenen Daten verwaltet wird.

Daten gleicher Lebensdauer werden in einer größeren Struktur zusammengefaßt, den Adreßräumen[3]. Von den Stackadreßräumen abgesehen, werden alle Adreßräume als Heap verwaltet. Es gibt vier Arten von Adreßräumen:

Codeadreßraum: Enthält den Code der Funktionen, Verwaltungsinformationen über die Schnittstellenfunktionen und die modulinternen Funktionen. Alle Klassenvariablen eines Typemanagers werden ebenso hier abgelegt. Jeder Typemanager hat einen eigenen Adreßraum.

Instanzadreßraum: Enthält die Instanzvariablen eines Moduls. Jede Instanz hat einen eigenen Adreßraum.

Benutzungsbezogener Adreßraum: Enthält alle benutzungsbezogenen Variablen eines bestimmten Moduls, das von einem Prozeß benutzt wird. Wenn ein Modul geöffnet wird (siehe Abschnitt 2.5), wird jedesmal ein neuer Adreßraum zugeordnet.

Stackadreßraum: Enthält den Stack eines Prozesses. Jeder Stack wird in einem eigenen Adreßraum abgelegt.

Der Zugriff auf diese verschiedenen Typen von Variablen erfolgt in der MONADS-Architektur immer auf einheitliche Weise.

2.4 Segmentierung und Speicherverwaltung

In der MONADS-Architektur wird der Speicher über Segmente angesprochen, die eine fast beliebige Größe besitzen können. Der Startpunkt der Adressierung von Segmenten in einer ausgeführten Funktion ist die Basistabelle, die Verweise auf verschiedene Segmentlisten enthält. Die Segmentlisten entsprechen den verschiedenen Gruppen von Daten im vorigen Abschnitt. Die Basistabelle ist in zwei Teile aufgeteilt. Die erste Basistabelle enthält Zeiger auf die jeweils gültigen Stackrahmen, die in der aktuellen Funktion sichtbar sind. Die zweite Basistabelle

[3]MONADS-Adreßräume werden mit dem englischen Begriff „Address Space“ bezeichnet, wenn Verwechslungsgefahr mit anderen Begriffen besteht.

enthält Zeiger auf die Klassenvariablen, Instanzvariablen und benutzungsbezogene Variablen.

Die Architektur kennt zwei verschiedene Arten von Segmenten: Stacksegmente und allgemeine Segmente. Stacksegmente enthalten ausschließlich Daten und werden für die Parameterübergabe beim Aufruf von Funktionen eingesetzt. Diese Segmente werden immer stackartig angelegt und am Funktionsende freigegeben. Allgemeine Segmente werden dynamisch angelegt. Sie können sowohl Daten als auch Zeiger auf weitere Segmente enthalten. Allgemeine Segmente können nur auf Heaps angelegt werden.

Segmente werden entweder über Zeiger in allgemeinen Segmenten oder über Segment Capabilities (siehe Bild 2.1) adressiert. Zeiger sind eine verkürzte Darstellungsform von Segment Capabilities. Sie enthalten nur die Anfangsadresse des Segments. Die Länge und Zugriffsrechte sind in den Verwaltungsinformationen des betreffenden Segments gespeichert.

Abbildung 2.1: Aufbau einer Segment Capability

Das Feld, in dem die Zugriffsrechte gespeichert sind, enthält auch eine Beschreibung der in dem Segment gespeicherten Information. Auf diese Weise können Segmente auch Daten enthalten, die nur auf bestimmte Weise manipuliert werden dürfen. Beispiele hierfür sind die oben erwähnten Segment Capability-Listen, darüberhinaus gibt es noch Module Capabilities, Module Call-Segmente und Semaphorsegmente. Segment Capability-Listen dürfen nicht mit den Zeigern in einem Segment verwechselt werden, denn in einer Segment Capability-Liste können Segment Capabilities ungekürzt gespeichert werden.

Die Benutzung von Zeigern zur Verbindung einzelner Datenelemente erlaubt die Bildung von beliebig komplexen Datentypen. Auf diese Weise können alle oben beschriebenen Gruppen von Daten einheitlich adressiert und organisiert werden.

Eine Grundregel gilt jedoch für die Zeiger auf andere Segmente: der Verweis darf höchstens die gleiche Lebensdauer besitzen wie die Lebensdauer der Daten, auf die verwiesen wird. Dies stellt sicher, daß Zeiger niemals auf nicht mehr existente Daten verweisen. Somit können z.B. keine Verweise von allgemeinen Segmenten auf Stacksegmente existieren, da Stacksegmente eine kürzere Lebensdauer haben.

Ein Modul kann nur auf Daten zugreifen, die ausgehend von der Basistabelle erreichbar sind. Somit kann durch entsprechende Aktualisierung der Basistabelle auf Architekturebene die Kapselung von Daten erzwungen werden. So wird sichergestellt, daß ein Modul nur auf die von den Sichtbarkeitsregeln bestimmten Daten zugreifen kann. Deshalb ist der Funktionsaufrufmechanismus der Grundstein des Schutzes auf dem MONADS-System.

2.5 Funktionsaufruf

Jedes Modul enthält zwei Listen, welche die Schnittstellenfunktionen und die internen Funktionen beschreiben. Diese beiden Listen mit Einsprungpunkten haben sieben Elemente: Die Startadresse und Länge der Funktion, der Einsprungpunkt in diesem Bereich, die Zahl lokaler Variablen und Parameter, die lexikalische Ebene und eine Adresse, die im Fall einer Exception (siehe Abschnitt 2.6) angesprungen wird. Die Zahl der lokalen Variablen und Parameter beschreibt die Größe der Segmentliste, die beim Aufruf der Funktion im Verwaltungsteil des Stacks angelegt wird. Alle Parameter und lokalen Variablen werden über diese Segmentliste auf dem Stack adressiert. Die Angabe der lexikalischen Ebene dient dazu, die entsprechenden Änderungen an der ersten Basistabelle vorzunehmen, so daß nur sichtbare Variablen adressiert werden können.

Der Funktionsaufruf erfolgt in drei Teilen: Vorbereitung des Aufrufs, Übergabe der Parameter und Aufruf. Die Vorbereitung und Durchführung des Aufrufs ist dabei der einzige Teil, der sich bei Aufrufen an Schnittstellenfunktionen bzw. interne Funktionen unterscheidet.

Bei Aufrufen innerhalb des Moduls ist nur die Angabe der gewünschten Funktionsnummer erforderlich. Die Funktionsnummer ist ein Index in die Liste der Einsprungpunkte interner Funktionen. Es wird eine Segmentliste für die Parameterübergabe auf dem Stack angelegt und Verwaltungsinformationen im neuen Stackrahmen eingetragen. Der Aufruf aktualisiert die erste Basistabelle, um die Sichtbarkeit der lexikalischen Ebenen anzupassen.

Externe Aufrufe aktualisieren zusätzlich die zweite Basistabelle, um die Sichtbarkeit der in Modulen gespeicherten Daten anzupassen. Bei externen Aufrufen ist darüberhinaus zu beachten, daß eine Module Capability (siehe Bild 2.2) für die Identifizierung des Moduls und die Prüfung der Zugriffsrechte vorgewiesen werden muß. Nur wenn die Zugriffsrechte zum Aufruf berechtigen, wird dieser durchgeführt. Sonst wird eine Exception ausgelöst.

Abbildung 2.2: Aufbau einer Module Capability

Module Capabilities werden von speziellen Maschinenbefehlen erzeugt und manipuliert. Sie werden durch die Angabe eines speziellen Typs bei der Erzeugung des enthaltenden Segments vor Manipulationen durch normale Datenzugriffe geschützt. Eine Module Capability wird bei der Übersetzung eines Typemanagers oder bei der Erzeugung einer Instanz erzeugt. Die Befehle zur Manipulation umfassen das Erstellen einer Kopie (außer dies ist durch eine Angabe im Statusfeld verboten worden), die Einschränkung der Rechte und einige Statusabfragen. Die Erweiterung von Rechten ist bei normalen Module Capabilities nicht möglich,

dies ist nur dem Inhaber der bei der Erzeugung eines Moduls generierten speziellen Module Capability erlaubt. Diese spezielle Module Capability existiert nur einmal. Somit ist eindeutig festgelegt, wer das Recht hat, beliebige Zugriffsrechte zu setzen.

Weil Module üblicherweise mehrere Aufrufe an ein anderes Modul durchführen und die Kosten der Anpassung der adressierbaren Umgebung recht hoch sind, müssen Module ähnlich wie Dateien geöffnet werden. Hierfür muß beim Aufruf eine weitere Datenstruktur bereitgestellt werden, ein sogenanntes Module Call-Segment. In dieser (ebenfalls in einem speziellen Segmenttyp abgelegten) Struktur wird u.a. die zweite Basistabelle abgelegt.

Um diese Struktur zu initialisieren, muß mit einem speziellen Befehl die `open`-Schnittstellenfunktion aufgerufen werden. Bei diesem Aufruf muß die Module Capability vorgelegt werden, deren Zugriffsrechte überprüft und in das Module Call-Segment kopiert werden. Die Module Capability muß nur einmal beim `open`-Aufruf vorgelegt werden.

Die vom Autor des Moduls geschriebene `open`-Funktion legt die benutzungsbezogenen Variablen an, die bis zum Schließen des Moduls erhalten bleiben. Nach dem Aufruf von `open` ist die zweite Basistabelle vollständig ausgefüllt. Alle weiteren Aufrufe benötigen nur noch das Module Call-Segment, um die sichtbaren Adreßbereiche zu setzen. Weil beim Aufruf einer externen Funktion die zweite Basistabelle jeweils komplett ausgetauscht wird, ist der Schutz der im Modul gekapselten Daten gewährleistet.

2.6 Exceptions

Die Architektur sieht eine Implementierung eines Exception-Systems vor. Jeder Fehler löst eine Exception aus, z.B. ein Zugriff über die Segmentgrenzen hinaus oder eine Division durch Null. Darüberhinaus kann jede Klasse beliebige Exceptions in der Klassenschnittstelle definieren und in der Implementierung auslösen. Diese Möglichkeit wird hauptsächlich bei Vor- und Nachbedingungen der verschiedenen Schnittstellenaufrufe eingesetzt.

Jede Funktion kann Code für die Behandlung von eventuell auftretenden Exceptions enthalten, muß dies jedoch nicht. Die Adresse des Behandlungscodes wird in der Beschreibung des Einsprungpunkts oder mit einem speziellen Maschinenbefehl angegeben. Wenn die aktuelle Funktion keinen Behandlungscode enthält, dann wird der Stack so weit abgebaut, bis eine Funktion im Aufrufpfad gefunden wird, die aufgetretene Exceptions behandelt. Dies erlaubt eine elegante Behandlung von Fehlersituationen. Auch der Abbruch von Programmen kann implementiert werden, indem vom Betriebssystem eine Exception ausgelöst wird.

Der konzeptionellen Eleganz stehen jedoch in der aktuellen Implementierung viele Probleme gegenüber. Wenn eine Exception auftritt, aber die aktuelle Funktion keinen Behandlungscode besitzt, dann wird die Ausführung an einer Stelle

unterbrochen, an der kein Wiederaufsetzen möglich ist. Der Stack wird zur Behandlung der Exception durch eine weiter außen liegende Funktion abgebaut und die lokalen Variablen gehen verloren.

Dies kann bei asynchronen Exceptions jederzeit auftreten, z.B. bei einem vom Benutzer ausgelösten Programmabbruch. Dies birgt die Gefahr, daß Instanzvariablen durch die Exceptionbehandlung nicht mehr konsistent sind. Nicht vorgenommene Semaphoroperationen führen so zu Synchronisationsfehlern.

Besonders störend sind neu auftretende, asynchrone Exceptions in einer gerade laufenden Behandlungsroutine. Der für synchrone Exceptions sinnvolle Mechanismus, die nächste, weiter außen liegende Behandlungsroutine anzuspringen, kann hier negative Auswirkungen haben. Beispielsweise behandelt der Kommandozeileninterpreter alle Exceptions und wäre somit nach dem Abbruch von Programmen wieder benutzbar. Wenn aber hier während der Behandlung einer Exception eine weitere auftritt, z.B. durch vom Benutzer schnell hintereinander angeforderte Programmabbrüche, dann wird auch der Kommandozeileninterpreter beendet. Der Benutzerstack ist somit völlig leer, was einem Löschen des Benutzers entspricht. Dies ist natürlich unerwünscht.

Die Exceptions in der aktuellen Form sind daher nicht für die direkte und systematische Benutzung in Programmen geeignet.

Kapitel 3

Der GNU C-Compiler

Die Entscheidung, einen bestimmten Compiler als Basis für die Portierung auf eine neue Architektur zu nehmen, ist eine der wichtigsten Entscheidungen der ganzen Arbeit. Die Auswahl ist jedoch sehr begrenzt, weil kommerzielle Compiler für eine Portierung wegen des fehlenden Compilerquelltextes nicht geeignet sind. Es wäre zwar teilweise möglich, den Quelltext für einen solchen Compiler zu erhalten, die Kosten dafür wären aber vergleichsweise hoch.

Bei den Entwicklungsplattformen gibt es noch weniger Auswahl. Die meisten für die Erzeugung von Compilern notwendigen Programme wurden unter UNIX entwickelt. Dies allein begründet die Verwendung eines UNIX-basierten Rechners für die Entwicklung aber noch nicht. Alle diese Programme sind auch auf anderen gängigen Betriebssystemen verfügbar. Die Erzeugung eines Compilers ist auf einem UNIX-System immer noch am einfachsten, weil die Compilerentwickler selbst hauptsächlich auf UNIX-Systemen arbeiten. Der Erzeugungsprozeß eines Compilers ist hochautomatisiert, verlangt aber eine weitgehende Kompatibilität mit der UNIX-Entwicklungsumgebung, die andere Betriebssysteme meist nicht erreichen. Der erzeugte Compiler muß nicht unbedingt auf dem selben Betriebssystem laufen, wenn die Hilfsprogramme für die Erzeugung von Programmen eines anderen Betriebssystems vorhanden sind.

In Frage kamen aus Zeitgründen nur Compiler, die aufgrund ihres Entwurfs eine leichte Portierung erlauben. Der Zeitaufwand einer teilweisen Neuimplementierung eines Compilers ist einfach zu groß für eine Diplomarbeit. Der Codegenerator und wahrscheinlich auch große Teile des Optimierers müßten neu implementiert werden.

Derzeit gibt es zwei im Quelltext verfügbare portable C-Compiler. Der erste, noch relativ neue ist LCC [3], ein portierbarer ISO C-Compiler. Bisher liegen nur wenige Erfahrungen zu seiner Portierung vor, weil er nur für vier Prozessorfamilien (Alpha, MIPS, SPARC und x86) portiert wurde. Alle Portierungen sind von den Autoren selbst implementiert worden. Der Compiler läuft derzeit unter diversen UNIX-Varianten und MS-DOS. Die Portierung auf eine andere Plattform ist im oben angegebenen Buch beschrieben, das u.a. den dokumentierten Quelltext

für den Compiler enthält. Die Portierung ist relativ leicht möglich. Es sind viele Anpassungsmöglichkeiten an die Gegebenheiten des Zielsystems vorhanden. Allerdings ist es nicht offensichtlich, ob die Besonderheiten des MONADS-PC in der Maschinenspezifikation auszudrücken sind. Derzeit werden nur die gängigsten Betriebssysteme unterstützt. Das schränkt den Einsatzbereich des Compilers etwas ein.

Der Compiler enthält in der Standardkonfiguration immer sämtliche Codegeneratoren. Die Übersetzung von Quelltexten für andere Prozessoren bzw. Betriebssysteme ist so leicht möglich. Es wird so aber auch viel Code eingebunden, der nur selten benötigt wird. Als sehr problematisch ist die mittlerweile veraltete Dokumentation durch das Buch anzusehen, da der Compiler in der Zwischenzeit weiterentwickelt wurde. Durch den zentralistischen Entwicklungsansatz der Autoren ist kein Zugriff auf den gerade aktuellen Compilerquelltext möglich. Das erschwert die effektive Abhilfe bei eventuell auftretenden Compilerfehlern. Das Auftreten von Fehlern im Compiler ist gerade bei der Portierung auf den MONADS-PC zu erwarten, weil der Prozessor einige Eigenschaften hat, die ihn stark von allen bisherigen Portierungen unterscheiden.

Der zweite portable Compiler ist der GNU C-Compiler der Free Software Foundation. Er ist schon seit Jahren weit verbreitet und wird an vielen Stellen auch für kommerzielle Projekte verwendet. Der Compiler ist für fast jeden Prozessor und fast jedes Betriebssystem verfügbar. Auf manchen Betriebssystemen ist der GNU C-Compiler (GCC) der einzige benutzbare Compiler, weil die mitgelieferten oder vom Hersteller angebotenen Compiler teilweise starken Einschränkungen unterliegen oder nur der Kernighan & Ritchie-Sprachstandard [4] angeboten wird. GCC unterstützt den ISO C-Standard mit einigen Erweiterungen und hat nur wenige hart kodierte Beschränkungen. Aufgrund seiner langjährigen Verfügbarkeit und der vielen Portierungen ist er gut ausgetestet und bietet eine stabile Basis für Weiterentwicklungen.

Die Wahl fiel auf den GNU C-Compiler, weil er als einziger frei verfügbarer C-Compiler bereits auf eine sehr große Anzahl Prozessoren portiert wurde. Es gibt Portierungen für etwa 30 verschiedene Prozessorarchitekturen und mehrere Dutzend Betriebssysteme in hunderten Varianten. Die Portierung ist im Handbuch [10] ausführlich beschrieben. Da sehr viele Prozessoren unterstützt werden, kann oft auf Teile anderer Portierungen zurückgegriffen werden.

Da die meisten Portierungen nicht von den ursprünglichen Entwicklern implementiert wurden, ist das Wissen über die interne Funktion des Compilers vielen bekannt. Der Compiler wird laufend weiterentwickelt. Die Portierung des Compilers kann immer von der gerade aktuellen Entwicklerversion ausgehend durchgeführt werden. Diese Version des Compilers wird etwa wöchentlich aktualisiert. Sie wird nicht in größerem Umfang veröffentlicht, weil sie wie jede Software im Teststadium Fehler enthält. Derzeit ist die Entwicklerversion auf dem FTP-Server `vger.rutgers.edu` im Unterverzeichnis `/pub/gcc` für jeden Interessenten verfügbar. Die aktuelle Entwicklerversion korrigiert normalerweise alle in den vorigen Versionen gefundene Fehler und ist trotz der neuen, sich gerade im Test

befindenden Codeteile meist sehr zuverlässig. Wenn Probleme auftreten, sind die Ursachen oft mit dem Protokoll der durchgeführten Änderungen einzukreisen.

Eine weitere interessante Eigenschaft des GNU C-Compilers ist die Verfügbarkeit von Frontends für andere Programmiersprachen. Diese verwenden den gleichen Codegenerator und profitieren so von der weiten Einsetzbarkeit des Compilers. Die Benutzung eines Codegenerators für viele Sprachen hat den Vorteil, daß dieser gut getestet ist und daß die Optimierungen automatisch in allen Sprachen wirksam sind. Es gibt u.a. Frontends für Ada, C++, Chill, FORTRAN-77, Java, Objective C, MODULA-2 und Pascal.

Dies eröffnet die Perspektive, in Zukunft mit wenig Aufwand die gebräuchlichsten Programmiersprachen auch auf dem MONADS-PC zur Verfügung zu haben. Speziell objektorientierte Programmiersprachen wären hier interessant, um die Lücke zwischen den von der Architektur angebotenen Modulen und der internen Datendarstellung zu verringern. Die Kombination von großen und kleinen Objekten würde eine viel komfortablere und weniger fehleranfällige Programmierung des MONADS-PC erlauben.

3.1 Eigenschaften des GNU C-Compilers

Die schon oben erwähnte hervorstechendste Eigenschaft des GNU C-Compilers ist die hohe Portabilität des Codegenerators. Der Compilercode selbst ist fast ebenso portabel, denn die Implementierung enthält nur wenige Annahmen über das zugrundeliegende Betriebssystem.

Die Lauffähigkeit des portierten Compilers auf dem MONADS-PC selbst ist kein vordringliches Ziel. Es ist wegen der großen Komplexität des Compilers damit zu rechnen, daß das ausführbare Programm des Compilers für den MONADS-PC sehr groß wird. Darüberhinaus ist die derzeitige Implementierung des MONADS-PC nicht gerade als schnell zu bezeichnen. Es ist oft einfacher, die Programme auf einem anderen Rechner zu übersetzen. Für einen auf dem MONADS-PC selbst ablaufenden Compiler ist es erforderlich, einen großen Teil der C-Bibliothek funktionsfähig zu haben.

Zuvor muß die neu erstellte Portierung zuverlässig funktionieren. Danach kann begonnen werden, die C-Bibliothek zu portieren bzw. Teile davon neu zu erstellen. Das Ziel dieser Diplomarbeit ist daher primär die Portierung des Compilers. Die Portierung der C-Bibliothek soll zwar begonnen werden, jedoch soll dies nur Prototypcharakter haben. Die anzuwendenden Konzepte sollen deutlich gemacht werden. Die Portierung der C-Bibliothek erfordert sehr viel Zeit, ist aber großteils nicht schwierig.

Gemessen am Umfang des Compilers ist es einfach, einen neuen Codegenerator zu implementieren: es genügt, eine üblicherweise etwa 3000 Zeilen umfassende Beschreibung der Zielmaschine zu erstellen. Verglichen mit dem Umfang des allen Portierungen gemeinsamen Codes von etwa 300 000 Zeilen ist dies ein sehr

kleiner Teil. Die Maschinenbeschreibung besteht aus einer Beschreibung aller für den Compiler relevanten Befehle und den Eigenschaften des Prozessors. Die Beschreibung erfolgt teilweise in einer an die intern verwendete „Register Transfer Language“ (kurz RTL) angelehnten Spezifikationssprache und teilweise in C-Code. Die Beschreibung der vom Befehlssatz unabhängigen Eigenschaften des Prozessors Befehlssatz erfolgt durch eine Sammlung von Makrodefinitionen. Eine ausführliche Beschreibung der RTL ist in [10] zu finden. Die verwendete RTL wurde von der Darstellung in [2] inspiriert und unterscheidet sich von anderen, oft ebenfalls RTL genannter Zwischencodes anderer Compiler.

Trotz der primären Ausrichtung auf die Portabilität wird durch Einsatz von Codeoptimierungen recht effizienter Code erzeugt. Die Codegüte von übersetzten Programmen ist mit kommerziell erhältlichen Compilern vergleichbar, oftmals wird schnellerer Code erzeugt. Der Compiler selbst ist dabei bei der Übersetzung recht schnell. Dies hat sicherlich zur Beliebtheit des Compilers beigetragen. Ein anderer Faktor ist die Möglichkeit, auf vielen Betriebssystemen und Architekturen denselben Compiler benutzen zu können. Dies beschleunigt die Entwicklung von portablen Programmen durch die Einheitlichkeit der Programmierumgebung.

3.2 Aufbau des GNU C-Compilers

Das Compilersystem hat einen sehr ähnlichen Aufbau wie andere, in der UNIX-Welt bekannte Compiler. Das einzige Programm, das bei normaler Verwendung des Compilers benutzt wird, ist der sogenannte Compiler Driver `gcc`. Die Hauptaufgabe dieses Programms ist es, die für die Übersetzung benötigten Programme in der richtigen Reihenfolge und mit den benötigten Parametern zu starten.

Im Falle der Übersetzung von C-Programmen (siehe Bild 3.1) ist das zunächst einmal der Präprozessor `cpp`, der textuelle Transformationen am Eingabequelltext vornimmt, z.B. die mit `#define` definierten Makros einsetzt. Die Ausgabe des Präprozessors ist wiederum die Eingabe des eigentlichen Compilers `cc1`, der das C-Programm in Assemblercode übersetzt. Je nach den bei der Compilierung angegebenen Optionen wird der Assemblerquelltext noch assembliert und eine Objektdatei erzeugt. Diese Objektdatei wird anschließend mit anderen Objektdateien und Bibliotheken zu einem ausführbaren Programm gebunden. Die gestrichelten Linien deuten den Informationsfluß bei Angabe mehrerer Quelldateien und optionaler Assemblerquelltexte bzw. Objektdateien an.

Es ist also normalerweise nicht nötig, den Assembler bzw. den Linker direkt aufzurufen. Diese Programme erwarten oft auf jedem Betriebssystem andere Parameter und sind so nicht einmal in ihrem Aufruf portabel. Durch die Benutzung des Compiler Drivers kann der Aufruf systemunabhängig gestaltet werden.

Als zusätzliche Funktion bietet der Compiler Driver von GNU C die Möglichkeit an, zwischen verschiedenen installierten Compilerversionen bzw. Zielsystemen

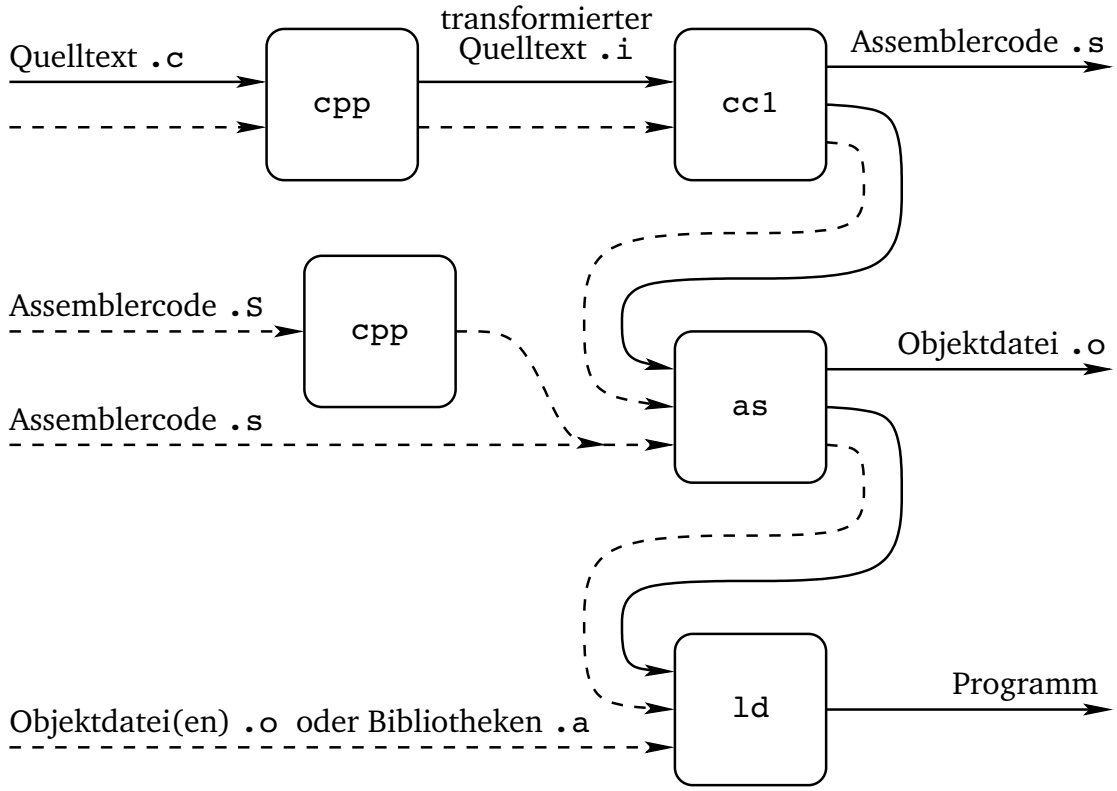

Abbildung 3.1: Einzelkomponenten des GNU C-Compilersystems

umzuschalten. Durch Angabe jeweils einer Option kann eine andere Compilerversion oder eine andere Zielplattform gewählt werden. Die verschiedenen Compilerversionen und Versionen für unterschiedliche Zielsysteme können nebeneinander benutzt werden, ohne sich zu beeinflussen.

Der eigentliche Compiler `cc1` besteht aus drei größeren Teilen (siehe Bild 3.2), die über definierte Schnittstellen miteinander interagieren. Eines davon ist das Frontend, das aus einem syntaktisch korrekten Programmquelltext einen abstrakten Syntaxbaum konstruiert. Im GCC ist die Baumdarstellung Teil der Schnittstellenbeschreibung des Zwischencodegenerators. Die Baumdarstellung ist von der Quellsprache unabhängig und erlaubt so, die verschiedenen unterstützten Programmiersprachen in eine einheitliche Darstellung zu bringen.

Der zweite Teil des Compilers generiert aus dem abstrakten Syntaxbaum eine Darstellung in einer Zwischensprache, genannt RTL. Daran anschließend wird die RTL in mehreren Schritten optimiert, die Register zugeteilt und schließlich Assemblercode ausgegeben. Die sprach- und maschinenabhängige Optimierung im zweiten Übersetzungsschritt formt hauptsächlich arithmetische Ausdrücke um und sorgt für eine möglichst einfache RTL-Darstellung. Die Optimierungen in diesem Schritt beschränken sich auf einfache Verfahren, die meist nur lokal die RTL-Darstellung verbessern, z.B. die optimale Anordnung von Schleifen. Die Eliminierung gemeinsamer Ausdrücke und andere leistungsfähige Optimierun-

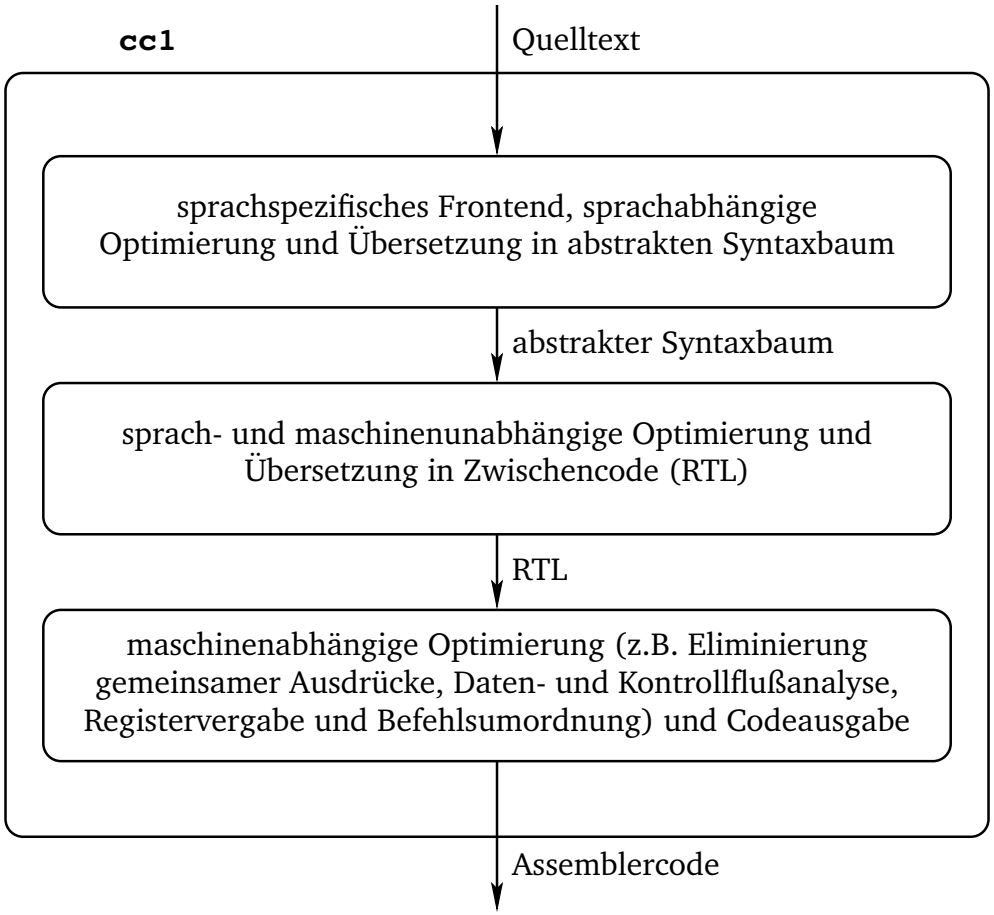

Abbildung 3.2: Grobaufbau des GNU C-Compilers

gen erfolgen erst nach der Erzeugung der RTL-Darstellung, denn es ist nicht ohne weiteres möglich, diese in der Baumdarstellung oder während der RTL-Erzeugung vorzunehmen.

Diese strikt durchgehaltene Trennung der verschiedenen Funktionsbereiche mit klar definierten Aufgaben ermöglicht es, jeden Teil ohne Veränderungen an anderen zu modifizieren. Das erlaubt, neue Programmiersprachen oder neue Optimierungsverfahren für die Codegenerierung zu implementieren. Alle neuen Optimierungsverfahren sind so sofort für alle Programmiersprachen verfügbar.

3.3 Arbeitsweise von GNU C

Der GNU C-Compiler ist, wie viele andere Compiler, ein parsergesteuerter Übersetzer (siehe auch [1, 12]). Das bedeutet, daß der Parser die zentrale Koordinierung des gesamten Übersetzungsprozesses übernimmt. Der Parser ruft z.B. den Scanner auf, um neuen Quelltext zu lesen und ruft die verschiedenen Funktionen zur Erzeugung des abstrakten Syntaxbaums auf. Sobald vom Parser erkannt

wird, daß eine Funktion vollständig ist, wird die RTL-Darstellung generiert und die Optimierung der RTL-Darstellung und die Assemblerquelltexterzeugung angestoßen.

Der abstrakte Syntaxbaum wird so während der Abarbeitung des Quelltexts Stück für Stück aufgebaut. Sofort nach der RTL-Erzeugung werden praktisch alle Anweisungen wieder aus der Baumdarstellung gelöscht, um Platz zu sparen. Nur Informationen, die für spätere Übersetzungsschritte noch relevant sind, verbleiben in der Baumdarstellung. Dies sind globale Variablendeklarationen, die Beschreibung der lexikalischen Ebenen des Programms und andere, während der gesamten Übersetzung benötigte Daten. Lexikalische Ebenen sind für die Betrachtung von C nicht relevant, sie sind für die Unterstützung anderer Sprachen dennoch in der Baumdarstellung vorhanden.

Die Erzeugung der RTL-Darstellung berücksichtigt schon die Existenz bestimmter Befehlsmuster. Die RTL ist somit immer prozessorabhängig. Die vom Prozessor unterstützten Adressierungsarten werden ebenfalls bei der RTL-Erzeugung berücksichtigt. Nicht vorhandene Befehle werden ebenfalls sofort bei der Erzeugung der RTL in mehrere andere Befehle umgesetzt.

Die RTL wird für die Erzeugung des Codes einer Funktion benutzt. Sie beinhaltet nicht alle Informationen, die für die Codegenerierung notwendig sind. Auf vielen Prozessoren wird z.B. der Code für den Funktionsanfang und das Funktionsende direkt vom bei der Portierung angegebenen Code in die Assemblerquelltextdatei geschrieben. Es ist aber optional möglich, Teile dieser Abschnitte in die RTL-Darstellung einzubeziehen, damit der Optimierer und der Scheduler darauf Zugriff haben. Dies lohnt sich nur bei Prozessoren, die bereits an diesen Stellen von Optimierungen und Befehlsumstellungen profitieren.

Der Parser stößt nach Fertigstellung der RTL-Zwischencodedarstellung den Optimierer an. Dieser arbeitet hauptsächlich mit der RTL-Darstellung, manchmal wird auch auf Daten der Baumdarstellung zurückgegriffen. Folgende Optimierungsschritte werden bei voll aktivierter Optimierung durchgeführt:

Sprungoptimierung: Alle Sprünge auf den unmittelbar folgenden Befehl werden entfernt, Sprünge über andere Sprungbefehle hinweg und Sprünge auf Sprünge werden vereinfacht. Unbenutzte Codemarken und unerreichbare Codestellen werden gestrichen. Diese Optimierung wird mehrmals an verschiedenen Stellen des Optimierungsvorgangs wiederholt, damit weitere Vereinfachungen des von anderen Optimierungen erzeugten Codes vorgenommen werden können.

Optimierung bedingter Sprünge: Hier werden bedingte Sprünge vereinfacht, deren Ziel die identische oder die negierte Bedingung auswertet (jump threading).

Beseitigung gemeinsamer Teilausdrücke: Hierzu wird zuerst der Bereich ermittelt, in dem ein Register (bisher sind dies noch virtuelle Register) benutzt wird. Der Code wird in erweiterte Basisblöcke aufgeteilt, die nur an

einem einzigen Punkt betreten, jedoch möglicherweise an mehreren Stellen verlassen werden können. In solchen Blöcken werden mehrfach berechnete Ausdrücke durch eine Referenz auf ihr erstes Auftreten ersetzt (common subexpression elimination). Konstante Ausdrücke werden durch ihren Wert ersetzt (constant propagation).

Schleifenoptimierung: Invarianter Code wird vor die Schleife verschoben, Induktionsvariablen werden eliminiert, Schleifen teilweise oder ganz abgerollt und die Kosten für Berechnungen von Variablenwerten in Schleifen durch Umstellung des Ausdrucks verringert (invariant code motion, induction variable elimination, loop unrolling und strength reduction).

Daten- und Kontrollflußanalyse: Die einzige unmittelbare Optimierung in diesem Durchlauf ist die Entfernung von unerreichbarem Code, der Schleifen enthält. Diese werden von der Sprungoptimierung nicht erkannt. Dieser Durchlauf schafft die Voraussetzung für die folgenden Optimierungen. Der Code wird in Basisblöcke aufgeteilt, die an jeweils genau einer Stelle betreten und wieder verlassen werden.

Kombination mehrerer Befehle: Zwei oder drei Befehle können u.U. je nach Datenfluß durch Eliminierung unnötiger Kopieroperationen zu einem Befehl zusammengefaßt werden. Zusätzlich werden dadurch mögliche algebraische Vereinfachungen durchgeführt. Auch diese Optimierung wird bei Bedarf mehrfach wiederholt, um die Änderungen der RTL durch andere Optimierungen zu berücksichtigen. Der Zweck des frühen ersten Aufrufs dieser Optimierung ist die daraus resultierende deutliche Verbesserung des generierten Codes.

Befehlsanordnung: Viele RISC-Prozessoren besitzen mehrere unabhängige Recheneinheiten oder Befehle, deren Ergebnisse erst nach mehreren Takten zur Verfügung stehen. Bei diesen Architekturen werden die Befehle so umgestellt, daß möglichst wenige Blockierungen durch zu diesem Zeitpunkt nicht verfügbare Funktionseinheiten vorkommen (scheduling). Diese Optimierung wird vor der endgültigen Codeausgabe wiederholt.

Registerverteilung: Hier werden in drei Schritten die virtuellen[4] Register in reale Register umgewandelt. Zuerst wird die geeignetste Registerklasse für jedes virtuelle Register bestimmt. Mit einem schnellen Algorithmus wird die Registervergabe innerhalb der Basisblöcke erledigt. Den verbleibenden virtuellen Registern werden mit einer Variante des Färbeverfahrens reale Register zugeordnet.

Register- und Speichervergabe: Die in der vorigen Phase ermittelte Registerbelegung wird in die Befehle eingesetzt und auf Konflikte überprüft. Es ist möglich, daß das zugeteilte Register an der Stelle nicht einsetzbar ist oder daß überhaupt kein freies Register gefunden werden konnte. In diesen Fällen werden Register entsprechend im Stack zwischengespeichert und der benötigte Wert in ein geeignetes Register geladen.

[4] Es wird bis zu diesem Punkt angenommen, daß es beliebig viele Register gibt.

weitere Sprungoptimierungen: Viele RISC-Prozessoren führen aus Effizienzgründen noch den Befehl nach Sprungbefehlen aus, um Blockierungen der Pipeline zu vermeiden. Hier kann evtl. ein Befehl eingesetzt werden, der ursprünglich am Ziel des Sprungs stand. Ebenso bieten einige Prozessoren die Möglichkeit an, die Wahrscheinlichkeit[5] der Sprungdurchführung anzugeben. GCC kann hier auf die Ergebnisse eines Probedurchlaufs des Programms zurückgreifen, um realistische Werte zu bekommen.

Nach all diesen Optimierungen wird schließlich der Assemblerquelltext ausgegeben. Parallel dazu werden noch verbliebene überflüssige Befehle entfernt. Wenn in der Maschinenbeschreibung Peephole-Optimierungen angegeben sind, wird versucht, mit diesen den Assemblerquelltext weiter zu vereinfachen.

Weil der Compiler eine Funktion nach der anderen abarbeitet, kann der Code für den Funktionsanfang und das Funktionsende problemlos eingefügt werden. Die genaue Syntax der ausgegebenen Maschinenbefehle wird von Teilen der Maschinenbeschreibung festgelegt. Optional werden noch zusätzliche Informationen für symbolische Debugger in den Assemblerquelltext geschrieben. Bisher werden nur verschiedene UNIX-Debugger unterstützt.

3.4 Externe Hilfsprogramme

GCC benötigt einige externe Hilfsprogramme zur Erzeugung ausführbarer Programme, wie es für UNIX-basierte Compiler üblich ist. Ein Assembler ist unbedingt nötig, um aus dem Assemblerquelltext Maschinencode zu erzeugen. Zur Unterstützung getrennter Compilierung ist meist ein Linker erforderlich, der die Objektdateien nachträglich zu einem ausführbaren Programm bindet. Es gibt aber auch Möglichkeiten, ohne einen expliziten Bindevorgang ausführbare Programme zu erhalten. Ein solches Verfahren zur dynamischen Bindung wurde in [11] eingesetzt.

Zusätzlich werden für sinnvolles Arbeiten noch Programme für die Verwaltung von Bibliotheken benötigt. Der Linker muß ebenfalls das Binden mit Bibliotheken unterstützen. Beim Binden mit Bibliotheken werden nur die Teile der Bibliothek zum Programm dazugebunden, die direkt oder indirekt referenziert werden. Dies unterscheidet Bindevorgänge mit Bibliotheken von Objektdateien, die immer dazugebunden werden, ohne die Referenzierung zu prüfen.

Das Paradigma der getrennten Übersetzung erfordert einen Assembler, der mit noch nicht bekannten Symbolen umgehen kann und dem Linker die Information zum korrekten Zusammenfügen der Teile zur Verfügung stellt. Der Assembler muß dazu Teile der Symboltabelle in der erzeugten Objektdatei ablegen. Der Linker kennt so die in anderen Objektdateien zu definierenden Symbole und kann die entsprechenden Referenzen auflösen. Objektdateien sind wegen der

[5] Dies ist natürlich nur bei bedingten Sprüngen relevant.

noch vorhandenen symbolischen Information frei im Speicher verschiebbar, erst der Linker definiert für die Symbole endgültige, feste Adressen.

Der bisher vorhandene Assembler für den MONADS-PC ist für solche Arbeitsweise nicht ausgelegt. Es wäre auch sehr schwierig, die Funktionalität für getrennte Assemblierung einzubauen. Bei der Entwicklung des vorhandenen Assemblers wurde fest vorgegeben, daß die Ausgabe immer ein komplettes Modul ist. Dies stimmt nicht mit den Anforderungen des C-Compilers überein. Es muß also eine Lösung gefunden werden, die eine Aufteilung in Objektdateien erlaubt. Diese Objektdateien werden anschließend zu einem lauffähigen Programm gebunden.

Kapitel 4

MONADS-Pascal

Die einzige bisher auf dem MONADS-PC verfügbare höhere Programmiersprache ist MONADS-Pascal [9], eine modifizierte Version von Standard-Pascal. Die speziellen Eigenschaften der MONADS-PC-Architektur haben die Entwicklung der Sprache stark beeinflußt.

Die Sprache ist aber nicht ausschließlich durch die Architektur definiert. Das Ziel war die Untersuchung der architekturellen Konzepte des MONADS-PC im Rahmen einer Experimentalsprache. Ein weiteres Ziel ist es, eine für systemnahe Programmierung einsetzbare Programmiersprache auf dem MONADS-PC zu haben, die es erlaubt, große Teile des Betriebssystems in einer Hochsprache zu implementieren.

4.1 Unterschiede zu Standard-Pascal

Die Modulorientierung erfordert mehrere Anpassungen von Standard-Pascal. Es muß einerseits möglich sein, die Schnittstelle eines Moduls nach außen festzulegen, andererseits müssen auch andere Module aufgerufen werden können.

MONADS-Pascal unterstützt die vollständige Trennung der Klassendefinition von der Implementierung und erlaubt so, mehrere Implementierungen einer Klasse gleichzeitig verfügbar zu haben. Damit dies realisiert werden kann, muß jede Implementierung eindeutig identifizierbar sein. Die Implementierungen werden entsprechend dem Konzept des MONADS-PC zur Laufzeit referenziert, denn auch die Bindung zwischen Implementierung und Benutzung wird explizit zur Laufzeit durchgeführt. Die Bindung von Modulen wird durch Vorlage der Module Capability der gewünschten Implementierung vorgenommen.

Die Architektur realisiert den Schutz vor unberechtigter Benutzung auf Modulebene über Module Capabilities. Module Capabilities sind als spezieller Datentyp in MONADS-Pascal verfügbar. Auf Module Capabilities sind nur wenige Operationen erlaubt, um den Schutz zu gewährleisten. Da Module Capabilities ein

Modul nur identifizieren und schützen, muß für die Benutzung von Modulen ein weiterer Datentyp eingeführt werden, die Klassenvariable. Sie entspricht auf Architekturebene einem Module Call-Segment. Sie hat als Typ die Klasse und identifiziert für alle Aufrufe die Instanz, in der die Daten abgelegt werden. Die Typisierung der Module Capabilities wird sowohl zur Übersetzungs- als auch zur Laufzeit überprüft, um sicherzugehen, daß die richtigen Module angesprochen werden. Dazu werden in einem Verzeichnis die jeweils zu den Module Capabilities gehörenden Klassennamen abgelegt.

Entsprechend den von der Architektur angebotenen Daten verschiedener Lebensdauer können zwei unterschiedliche Arten von Modulen implementiert werden, Typemanager oder Code-Module. Die Lebensdauer aller globalen Variablen in Code-Modulen ist auf die Zeit zwischen einem `open`-Aufruf und dem zugehörigen `close`-Aufruf an das Modul beschränkt. Die Daten sind daher ausschließlich für den Aufrufer (natürlich nur über die Schnittstelle) zugreifbar. Der Speicherplatz ist direkt der entsprechenden Klassenvariable zugeordnet.

Typemanager bieten eine weitere spezielle Schnittstellenfunktion an, die erlaubt, Instanzen mit längerer Lebensdauer als die von Code-Modulen anzulegen. Ein `create`-Aufruf erzeugt eine Instanz, die alle mit dem speziellen Schlüsselwort `instance` deklarierten Variablen so lange speichert, bis die Instanz explizit gelöscht wird. Die Instanzvariablen sind für alle Aufrufer zugreifbar und können daher zum Austausch von Informationen zwischen verschiedenen Aufrufern benutzt werden. Die von Code-Modulen bekannten `open`- und `close`-Aufrufe sind auf den Instanzen ebenfalls verfügbar. Daten können auch weiterhin für einzelne Aufrufer gespeichert werden. Ein Typemanager kann also z.B. eine Textdatei implementieren, wobei für jeden Aufrufer die aktuelle Position im Text separat gespeichert wird. Die eigentliche Datei ist in Instanzvariablen abgelegt.

Die Synchronisation gemeinsamer Zugriffe ist in der Sprache nicht festgelegt. Dies ist durch von der Umgebung vorgegebene Mittel zu bewerkstelligen, z.B. mit den auf dem MONADS-PC vorhanden Semaphoroperationen.

4.2 Unterstützung der MONADS-PC-Architektur

Die Unterstützung der MONADS-PC-Architektur ist wegen der engen Anlehnung an die Konzepte mit relativ geringem Aufwand zu bewerkstelligen. Es ist aber an einigen Stellen möglich, aus mehreren Implementierungsalternativen zu wählen. Ein Beispiel ist die Granularität der Segmente. Die Robustheit des Codes wäre am größten, wenn jede einzelne Variable in ein eigenes Segment gelegt würde. Die Segmente sind aber für so kleine Objekte schlecht geeignet, da im Verhältnis zu den gespeicherten Daten zu viel Platz für die Verwaltung der Segmente verschwendet wird. Der MONADS-Pascal-Compiler benutzt deshalb ein Segment für alle in einer lexikalischen Ebene definierten unstrukturierten Variablen (z.B. `integer`, `real`, ...). Strukturierte und von der Architektur geschützte Typen

wie Arrays, Klassenvariablen, Module Capabilities usw. werden in eigenen Segmenten abgelegt.

Die Architektur macht es manchmal auch sehr schwer, auf den ersten Blick einfache Konstrukte auf der MONADS-PC-Architektur zu implementieren. Ein Beispiel sind Aggregattypen (Records) von Klassenvariablen, Module Capabilities und normalen Daten. Die von der Architektur vorgegebene und vom Prozessor überwachte Typisierung von Segmenten erlaubt es nicht, mehr als einen Typ in einem Segment zu verwenden. Das erfordert teilweise die Verwendung mehrerer Segmente für eine Variable. Potentiell kann dieser Fall geschachtelt auftreten, wenn ein Aggregattyp für ein Feld eines anderen Aggregattyps verwendet wird. In solchen Situationen ist es schwer, die Daten effizient zu speichern und trotzdem die freie Referenzierbarkeit zu gewährleisten. Beispielsweise können Felder, die selbst Aggregattypen sind, nicht in den äußeren Aggregattyp eingebettet werden. Diese komplizierte Speicherungsstruktur ist aufwendig anzulegen und verschwendet durch die oft sehr kleinen Segmente viel Platz.

Die zwischen Modulen eingesetzte Aufrufkonvention wird leicht verändert auch für Aufrufe von modulinternen Prozeduren verwendet. Die Architektur verhindert hier die Verletzung der Sichtbarkeitsregeln lokaler Variablen. Der Compiler setzt die im alten Befehlssatz vorhandene `tos`-Adressierungsart ein, die eine Mischform einer Akkumulatormaschine und einer klassischen Stackmaschine implementiert. Der erste Operand und das Zielregister ist bei arithmetischen Operationen immer der Akkumulator, der zweite Operand kann vom Stack genommen werden. Der Nachteil dieser einfachen Organisation des Berechnungsstacks ist die fehlende Adressierbarkeit von schon berechneten Ergebnissen anderer Teilausdrücke. Dies verhindert die Eliminierung gemeinsamer Teilausdrücke, außer diese werden explizit in lokalen Variablen abgelegt. Da diese Organisation restriktiver ist als es eigentlich nötig wäre, wird sie im überarbeiteten Befehlssatz durch universellere Einsetzbarkeit von Registern und freie Adressierbarkeit des Berechnungsstacks ersetzt. Die alten Adressierungsarten werden derzeit als Übergangslösung weiter unterstützt.

4.3 Programmieren mit Modulen

In diesem Abschnitt werden einige kleine Beispiele vorgestellt, die Unterschiede von MONADS-Pascal und Standard-Pascal illustrieren.

In Bild 4.1 ist eine Klassendefinition dargestellt, die einen Typemanager beschreibt. Die Klassendefinition eines Typemanagers enthält immer eine Deklaration der Schnittstellenfunktion `create`. Auch wenn `open` bzw. `close`, bei Typemanagern auch `create` und `delete`, in der Klassendefinition nicht angegeben sind, so sind diese Funktionen implizit doch immer vorhanden. Die fehlende Deklaration wird so behandelt, als ob keine Parameter oder Rückgabewerte angegeben wären. Wenn eine der sogenannten Basis-Funktionen nicht explizit

deklariert ist, dann kann sie nicht implementiert werden. Der Compiler erzeugt in diesem Fall automatisch eine leere Funktion.

```
type
  directory =
    class
      procedure create;
      function insert(name: string;
                      entry: modcap): boolean;
      procedure list(outmc: modcap of basic_text);
      function rename(old_name: string;
                      new_name: string): boolean;
      function remove(name: string): boolean;
      function lookup(name: string;
                      var entry: modcap): boolean;
    end;
```

Abbildung 4.1: Klassendefinition in MONADS-Pascal

Klassendefinitionen können von einer anderen Klassendefinition erben und diese so erweitern. Mehrfachvererbung ist nicht erlaubt. Die zu erweiternde Klasse wird nach dem Schlüsselwort `class` angegeben. Es ist aber nicht möglich, Funktionen der zu erweiternden Klasse umzudefinieren, denn das würde eine inkompatible Klasse erzeugen.

Alle in einer Klassendefinition angegebenen Schnittstellenfunktionen müssen implementiert werden. Dies bedeutet, daß auf den Code der zu erweiternden Klasse nicht zurückgegriffen werden kann, wie es in den meisten objektorientierten Programmiersprachen erlaubt ist. MONADS-Pascal unterstützt nur Typvererbung, aber keine Codevererbung. Klassen dienen nur als Strukturierungsmittel für Software, nicht als Mittel zur Wiederverwendung von Code.

Bild 4.2 ist ein Teil einer Implementierung der im letzten Beispiel benutzten Klasse `directory`. In Modulen können globale Variablen definiert werden. Die in Pascal übliche Definition von globalen Variablen wurde zweigeteilt, um die verschiedene Lebensdauer von Variablen in Typemanagern ausdrücken zu können.

Implementierungen einer Klasse müssen eine Definition ihrer eigenen Klasse enthalten. Zusätzlich benutzen Implementierungen typischerweise mehrere andere Klassen, die ebenfalls deklariert werden müssen. Es ist aber fehleranfällig, die Typdefinitionen an mehreren Stellen abzulegen. Es wird daher ein Mechanismus zur Einbindung der Klassendefinitionsdateien angeboten. Derzeit basiert die Einbindung auf dem Aufruf des C-Präprozessors `cpp`, der mit der Direktive `#include` die benötigte Funktionalität anbietet.

Modulimplementierungen können auch interne Funktionen definieren, die nicht Teil der Schnittstelle sind. Sie können nur innerhalb des Moduls benutzt werden. Im Beispiel ist dies `exists`. Weil selbst innerhalb eines Moduls der Aufruf von Schnittstellenfunktionen ohne den Besitz der entsprechenden Module Capability

```
directory manager mydir;

#include <directory.h>
#include <basic_text.h>

type
  entryptr = ^entry;
  entry = record ... end;

instance entrylist: entryptr;

retained node: entryptr;

function exists(name: string);
begin
  ...
end;

interface

procedure create;
begin
  entrylist := NIL;
end;

function insert(name: string; entry: modcap): boolean;
begin
  ...
end;

procedure list(outmc: modcap of basic_text);
var tty: basic_text;
begin
  tty.open(outmc, write_only, 0, false);
  node := entrylist;
  while (node <> NIL) do begin
    tty.writeline(node^.name);
    node := node^.next;
  end;
  tty.close;
end;

function rename(...): boolean;...
function remove(...): boolean;...
function lookup(...): boolean;...

.
```

Abbildung 4.2: Beispielimplementierung eines Moduls in MONADS-Pascal

nicht möglich ist, bieten interne Funktionen oft Aufgaben an, die von mehreren Schnittstellenfunktionen benötigt werden. Es ist analog zu Standard-Pascal möglich, Funktionen zu schachteln und so lexikalische Ebenen zu verwenden. Geschachtelte Funktionen sind immer lokal.

In der Funktion `list` wird demonstriert, wie ein anderes Modul aufgerufen werden kann. Hier wird über die Variable `tty` eine Instanz der Klasse `basic_text` angesprochen. Die Klasse `basic_text` definiert eine generische Schnittstelle für Textdateien. Auf dem MONADS-PC sind u.a. Implementierungen für die Terminalansteuerung und für normale Textdateien verfügbar. Wie von der Architektur vorgegeben, muß eine Module Capability vorgelegt werden, die beim `open`-Aufruf vom Benutzer übergeben wird.

Da ein Modul potentiell Zugriff auf mehrere Heaps mit unterschiedlicher Lebensdauer hat, muß schon beim Anlegen eines dynamischen Datenbereichs der Ort der Ablage bestimmt werden. In den meisten Fällen kann der korrekte Heap aus dem Ort der Zeigervariable, die angelegt werden soll, ermittelt werden. In bestimmten Fällen kann es aber auch Sinn machen, die Adresse von `instance`-Daten in `retained`-Variablen oder in lokalen Variablen zu halten. Daher wird in MONADS-Pascal der Standard-Pascal-Aufruf `new()` mit der (optionalen) Angabe des Heaps erweitert: `new instance()` bzw. `new retained()`. Falls die Angabe des gewünschten Heaps nicht erfolgt, wird der gleiche Heap wie der des angegebenen Zeigers verwendet. Je nach angegebenem Ausdruck kann teilweise erst zur Laufzeit ermittelt werden, welcher das ist. Der Zusatz `retained` ist eigentlich überflüssig, denn Verweise von `instance`-Zeigern auf `retained`-Daten sind von der MONADS-Architektur nicht erlaubt. Da dies teilweise nur zur Laufzeit ermittelt werden kann, ist die Möglichkeit zur expliziten Angabe vorhanden, um Programmfehler erkennen zu können.

MONADS-Pascal bietet einen sehr komfortablen Weg, mit den Zugriffsrechten auf Module umzugehen. Es bietet Mengenoperationen auf der Menge aller Schnittstellenfunktionen jeweils einer Klasse an, die automatisch mit den symbolischen Namen angesprochen werden können. Damit lassen sich Abfragen, ob eine bestimmte Schnittstellenfunktion mit den Zugriffsrechten einer Module Capability aufrufbar ist, ohne Kenntnis der Schnittstellennummern ausführen. Auf gleiche Weise können die Rechte einer Module Capability auf komfortable Weise eingeschränkt werden.

Dies wird über den speziellen Mengentyp `rightset` implementiert, der wie alle Mengendeklarationen noch die Angabe eines Grundtyps erfordert, in diesem Fall einen Klassennamen. Alle bekannten Mengenoperationen können auch auf diese speziellen Mengen angewandt werden. Es ist einfach möglich, neue Zugriffsrechte für eine Module Capability zu berechnen oder zu prüfen, ob eine dem Modul übergebene Module Capability genügend Rechte enthält, um die vom Modul benötigten Schnittstellenfunktionen aufzurufen.

Stringoperationen sind in MONADS-Pascal besonders leistungsfähig, denn der Typ `string` besitzt keine Längenbeschränkung. Dies erleichtert den Umgang

mit Strings deutlich, denn im Gegensatz zu vielen Pascal-Implementierungen, die oft kleine maximale Stringlängen vorschreiben, kann hier ohne Gefahr des Überlaufs von Strings gearbeitet werden. Dies vermeidet den komplexen Code, der bei Compilern mit beschränkter Maximallänge manchmal erforderlich ist, um auch mit längeren Strings arbeiten zu können. Viele der in einigen Pascal-Dialekten definierten Operationen auf Strings werden unterstützt. Beispielsweise erleichtert die Verwendung von „+“ als Konkatenationsoperator die Arbeit.

An einem wichtigen Teil von Standard-Pascal mußten große Anpassungen vorgenommen werden: den Dateioperationen. Durch die generelle Persistenz der Daten sind Dateien im klassischen Sinne überflüssig. Nur der vordefinierte Typ `text` blieb in leicht abgewandelter Form erhalten, um komfortabel Informationen in Textform ein- und ausgeben zu können. Der Typ-Konstruktor `file of` wurde völlig weggelassen.

Der Compiler kennt die Schnittstelle der Klasse `basic_text`, um die Implementierung von Zugriffen auf Textdateien möglichst einfach zu machen. Alle Operationen auf einer Variable vom Typ `text` werden vom Compiler in Aufrufe an eine Instanz der Klasse `basic_text` umgesetzt. Um die hierfür benötigte Module Capability dem Laufzeitsystem zu übergeben, gibt es die gegenüber Standard-Pascal veränderten Aufrufe `mreset`, `mrewrite` und `mappend`, die hier als Parameter die Dateivariable und eine Module Capability vom Typ `basic_text` erwarten. Diese Routinen schaffen die Verknüpfung der `text`-Variable und der `basic_text`-Instanz.

Ebenso wie in Standard-Pascal nehmen die Variablennamen `input` und `output` eine Sonderrolle ein. Sie werden standardmäßig für Ein- und Ausgabeoperationen verwendet. Anders als in Standard-Pascal müssen diese Variablen explizit deklariert (sinnvollerweise als `retained`-Variable) und durch die oben angegebenen Bindeaufrufe initialisiert werden. Meistens geschieht dies in der `open`-Schnittstellenfunktion. Anschließend können Standardfunktionen wie `readln` und `writeln` benutzt werden.

4.4 Eigenschaften des Compilers

Der MONADS-Pascal-Compiler ist als Cross-Compiler auf Sun-Workstations und als Native-Compiler auf dem MONADS-PC selbst verfügbar. Üblicherweise wird aus Zeitgründen die Entwicklung auf Sun-Workstations und das anschließende Herunterladen auf den MONADS-PC bevorzugt. Programme von der Größe des Compilers benötigen lange Übertragungszeiten und verbrauchen einen großen Teil des verfügbaren Speichers. Die auf dem MONADS-PC laufende Compilerversion ist in der Lage, sich selbst zu übersetzen. Das dauert sehr lange, weil der Hauptspeicher knapp ist.

Der Compiler übersetzt Module in einem einzigen Durchlauf. Das wirkt sich auf die möglichen Optimierungen und die Art der Codegenerierung aus. Der Com-

piler erzeugt für die meisten Zwecke ausreichend schnellen Code. Große Teile des MONADS-PC-Betriebssystems sind in MONADS-Pascal geschrieben. Nur die Teile, bei denen es wichtig ist, daß sie mit der maximal möglichen Geschwindigkeit ablaufen, sind in Assembler implementiert. Dazu gehört auch der Betriebssystemkern, dessen Codegröße jedoch durch die Modularisierung überschaubar bleibt.

Trotz der Einschränkungen, die bei einem in einem Durchlauf arbeitenden Compiler gelten, werden vergleichsweise komplexe Optimierungen eingesetzt. Teilweise werden Optimierungen vorgenommen, die von einigen kommerziell erhältlichen Compilern nicht beherrscht werden. Für jede `case`-Anweisung wird automatisch die optimale Übersetzungsvariante gewählt. Die unterstützten Alternativen sind Realisierungen mit einer Sprungtabellen oder verschachtelten `if`-Abfragen. Die Schwelle zwischen den beiden Realisierungen kann vom Benutzer angepaßt werden, um zum Preis erhöhter Codegröße die schnellere Variante zuzulassen.

Aufgrund des langjährigen Einsatzes als Hauptentwicklungswerkzeug von Software für den MONADS-PC ist der Compiler gut getestet und es treten nur noch sehr selten Fehler auf. Ein Problem mit extrem komplexen String-Ausdrücken ist derzeit noch im Compiler vorhanden. Unter bestimmten Umständen wird falscher Code erzeugt.

Kapitel 5

Konzepte der Portierung

Die Portierung des GNU C-Compilers auf eine Architektur wie die des MONADS-PC ist keine Aufgabe, die in das Schema für die vielen bisher durchgeführten Portierungen paßt. Die Architektur hat andere Anforderungen, bietet aber auch andere Möglichkeiten als die hier als herkömmlich bezeichneten Architekturen. Es muß zuerst ein Konzept erarbeitet werden, in das die Anforderungen und Möglichkeiten eingearbeitet sind. Darauf aufbauend kann dann die Portierung von GNU C begonnen werden.

5.1 Anforderungen

Die wichtigste Eigenschaft der Portierung soll die problemlose Übersetzbarkeit von vorhandenen C-Quelltexten sein. Dies bedeutet, daß auch Quelltexte, die in älteren Sprachstandards geschrieben sind und nicht dem strikteren ANSI- bzw. ISO-Standard entsprechen, ohne Anpassungen in korrekten Maschinencode übersetzbar sein sollen. Das häufigste Problem ist eine weggelassene Deklaration des Typs von Integer-Funktionsrückgabewerten. Nach der originalen Kernighan & Ritchie-Sprachdefinition ist das zulässig. Dies ist besonders bei kritisch, wenn der Rückgabewert ein Zeiger ist. Zeiger können nicht immer wie Integer-Werte behandelt werden.

Diese Anforderung an die Kompatibilität wurde in der vorigen Portierung [11] zwar auch angeführt, aber in der Implementierung nicht erfüllt. Nur nach dem ANSI C-Standard gültige Programme werden korrekt übersetzt. Diese Einschränkung ist für viele existierende Programme zu streng. Es gibt viele Programme, die nach der Kernighan & Ritchie-Sprachdefinition implementiert wurden, meist um mit allen C-Compilern kompatibel zu sein. Kaum ein Programm lief ohne Änderungen, obwohl keine maschinenabhängigen Funktionsaufrufe oder compilerabhängigen Konstrukte verwendet wurden.

Die Portierung muß im Rahmen der von der Architektur vorgegebenen Sicherheit geschehen. Es ist nicht akzeptabel, die Schutzkonzepte der MONADS-Architektur

aufzugeben, nur weil diese bei der Entwicklung der Sprache C nicht zugrundegelegt wurden. Es hat sich jedoch bei der Untersuchung des Problems gezeigt, daß der Schutz nicht kompromittiert werden muß, wenn einige die Robustheit betreffende Anpassungen vorgenommen werden. Dies wirkt sich aber nur auf die Benutzung bestimmter vorgegebener Funktionen aus, wie z.B. der Speicherverwaltung, die anders als bisher auf dem MONADS-PC üblich gestaltet wurden. Die Sprache C setzt voraus, daß jede Speicherzelle adressierbar ist, evtl. auch über Adreßrechnung. Dies erfordert Änderungen am Stackaufbau, denn sonst können Zeiger auf Variablen im Stack nicht in globalen Variablen gespeichert werden. Für viele Programme wäre das eine nicht tragbare Einschränkung und würde hohen Portierungsaufwand für einzelne Programme erfordern.

Die Modulorientierung des MONADS-PC ist nicht zu umgehen, denn dies würde das Schutzkonzept der Architektur unbrauchbar machen. Diese Maßnahme ist auch nicht nötig. Es gibt mehrere Möglichkeiten, wie C-Programme auf Module abgebildet werden können. Es muß eine Schnittstelle für den Start des Programms vorhanden sein. Die Realisierung für den MONADS-PC fügt dem noch die Schnittstellenfunktion `open` hinzu. Es ist aber nicht von vornherein festgelegt, wie der Code des Programms auf Module verteilt wird. Die verschiedenen Möglichkeiten der Aufteilung des Codes innerhalb eines Programms werden in Abschnitt 5.3 diskutiert.

Die grundlegende Anforderung der Übersetzung existierender Programme impliziert einige weitere Anforderungen, wie die Verwendung einer vollständigen Implementierung der C-Bibliothek. Existierender Code benutzt oftmals auch Funktionen, die erst später oder von anderen Standardisierungsgremien (z.B. X/Open oder POSIX) in den Umfang der C-Bibliothek aufgenommen wurden. Eine Implementierung der wichtigsten Funktionen der C-Bibliothek, wie sie in [11] vorgenommen wurde, ist deswegen nicht mehr ausreichend. Eine vollständige Implementierung einer C-Bibliothek ist sehr zeitaufwendig (mehrere Mannjahre), so daß hierfür auf vorhandenen Code zurückgegriffen werden muß. Ausführliche Informationen hierzu sind im Kapitel 7 zu finden.

In C geschriebene Programme müssen irgendeine Möglichkeit haben, auf andere Module zuzugreifen. Selbst so einfache Dinge wie Bildschirmausgaben können nur mit Hilfe anderer Module implementiert werden. C-Programme werden aber kaum direkt Module benutzen, denn die C-Bibliothek sollte diese Abstraktion anbieten. Die C-Bibliothek muß z.B. Dateizugriffe in Aufrufe an Verzeichnis- und Textdateimodule umsetzen.

Der Mechanismus zum Aufruf anderer Module unterscheidet sich auf Maschinenebene deutlich vom Aufrufmechanismus innerhalb eines Moduls. Der GNU C-Compiler unterstützt nur vergleichsweise kleine Variationen der Aufrufkonventionen und kann deswegen nicht Aufrufcode für beide Konventionen erzeugen. Es ist aber nicht erwünscht, jeden Aufruf an andere Module „von Hand“ neu in Assembler implementieren zu müssen. Dies wäre viel zu fehleranfällig. Der Aufrufcode für Schnittstellenfunktionen soll von einem Generatorprogramm aus der Klassendefinition erzeugt werden.

5.2 Maschineneigenschaften

Der MONADS-PC ist auf den ersten Blick eine relativ gewöhnliche 32 Bit-CISC-Maschine. Das gilt besonders für die angebotenen arithmetischen, logischen und sonstigen für Compiler relevanten Befehle. Für CISC-Maschinen relativ unüblich ist die fehlende Unterstützung für Halbwortzugriffe (16 Bit). Da es in C über den Umweg der Bitfelder möglich ist, Variablen von jeder Wortbreite zu erzeugen, müssen solche Zugriffe aus mehreren existierenden Befehlen zusammengesetzt werden.

Der Befehlssatz des MONADS-PC bietet keine Operationen auf Daten an, die kleiner als ein Wort sind. Deswegen müssen alle Operanden zuerst auf Wortbreite expandiert werden, bevor mit ihnen gearbeitet werden kann. Auch dies ist eine Eigenschaft, die normalerweise eher bei RISC-Architekturen anzutreffen ist. Darüberhinaus ist es nicht erlaubt, bei Wortzugriffen eine Adresse anzugeben, die nicht auf eine Wortgrenze fällt. Die derzeitige Implementierung durch den MONADS-PC unterstützt solche Zugriffe nicht. Für Programme ist aber nicht erkennbar, ob sie einen nicht ausgerichteten Speicherzugriff durchgeführt haben. Es werden einfach falsche Daten transferiert, ohne eine Exception auszulösen. Es muß also vom Entwurf der Compilerportierung gewährleistet sein, daß alle Daten an Wortgrenzen beginnen. In allen anderen Fällen muß der Zugriff entweder byteweise oder mit mehreren ausgerichteten Wortzugriffen realisiert werden. Beide Lösungen sind vergleichsweise langsam.

Eine Eigenschaft, die es auch auf einigen anderen Architekturen gibt, ist die Unterstützung nur eines Satzes von Schiebebefehlen. Die verschiedenen Schieberichtungen werden mit dem Vorzeichen der Schiebedistanz angegeben. Der Compiler unterstützt von sich aus Befehle mit explizit angegebener Schieberichtung, aber die Abbildung auf die real existierenden Befehle ist einfach. Rotieren bzw. Schieben nach rechts kann einfach als Schieben mit negierter Distanz umgesetzt werden. Der Optimierer ist für solche Architekturen ausgelegt und funktioniert daher auch für diese nicht ganz dem Modell des Codegenerators entsprechenden Maschinen. Es muß unbedingt beachtet werden, daß der Optimierer bei konstanten Schiebedistanzen nicht mit negativen Werten zurechtkommt. Für diesen Fall müssen die Schiebedistanzen so beibehalten werden und bei der Assemblercodeausgabe das Vorzeichen entsprechend angepaßt werden. Dies ist leicht möglich, denn die Schieberichtung ist noch in dem Zwischencodebefehl vermerkt.

Damit verbleibt als einzige, nicht mit einzelnen Maschinenbefehlen implementierbare Zwischencode-Funktion die Vorzeichenerweiterung von 16 Bit-Werten. Eine Vorzeichenerweiterung ist auf mehreren Wegen zu erreichen. Eine Möglichkeit der Realisierung ist die Kombination zweier Schiebebefehle. Zuerst wird 16 Bit nach links, dann wieder 16 Bit nach rechts verschoben, das zweite Mal mit Berücksichtigung des Vorzeichens. Eine andere Möglichkeit ist, das Vorzeichenbit der 16 Bit-Zahl abzufragen und mit der daraus gewonnenen Information die oberen 16 Bit des erweiterten Werts zu setzen.

Die derzeitige Implementierung der MONADS-Architektur bietet zum Debugging von Modulen die Möglichkeit, mit speziellen Befehlen ein Register im Prozessor auf die aktuelle Zeilennummer im Quelltext zu setzen. Diese oft sehr wertvolle Funktion soll auch bei C-Programmen auf Anfrage des Benutzers zur Verfügung stehen. Die Standardeinstellung sollte diesen Code nicht erzeugen, denn dies würde zu deutlich vergrößertem Programmcode und verlangsamter Ausführung führen.

Zur Zeit ist es nicht möglich, auf den Code des gerade ausgeführten Moduls zuzugreifen. Deswegen müssen alle Konstanten und sonstigen Informationen, die für die Verarbeitung des Programmtextes benötigt werden, in einem Datensegment abgelegt werden. Es ist ebenfalls unmöglich, Code außerhalb des Codesegments im Modul auszuführen. Dies hat Auswirkungen auf die Realisierungsmöglichkeiten von bei verschiedenen Programmiersprachen möglichen Aufrufen zwischen lexikalischen Ebenen. GNU C unterstützt lexikalische Ebenen im Zusammenhang mit Prozedurvariablen nur, wenn es möglich ist, vom gerade laufenden Programm dynamisch erzeugten Code ausführen zu können. Für ISO C ist dies keine nennenswerte Einschränkung, da von der Sprache keine lexikalischen Ebenen angeboten werden. GNU C bietet als Spracherweiterung verschachtelte Funktionen, die nur von sehr wenigen Programmen eingesetzt werden. Die meisten dieser Programme werden dennoch übersetzbar sein. Nur ein kleiner Teil, der Prozedurvariablen verwendet, wird nicht übersetzbar sein. Das Ziel der Übersetzbarkeit standardkonformer Programme wird dadurch nicht verletzt.

Die übrigen Maschineneigenschaften sind vergleichbar mit vielen anderen gebräuchlichen Prozessoren, es wird daher nicht weiter darauf eingegangen.

5.3 Module

Module sind ein zentrales Element der MONADS-PC-Architektur. Die Architektur unterstützt einen Modulbegriff, der auf einer hohen Abstraktionsebene liegt. Der Zwang, eine prozedurale Schnittstelle zur Verfügung zu stellen, fördert eine klare Strukturierung der Aufgabenverteilung zwischen Modulen. Module sind darüberhinaus das tragende Element des Schutzkonzepts, da sie nur über Capabilities angesprochen werden können. Die von der Architektur angebotenen Module stellen größere Einheiten dar. Die Implementierung von kleinen Objekten über Module ist ineffizient wegen des durch die Sicherheitsprüfungen relativ teuren Aufrufs von Schnittstellenfunktionen.

Für die Umsetzung von Programmen auf ein modulorientiertes System gibt es mehrere Möglichkeiten, die von der Struktur der Module und der Programme vorgegeben werden. Bei C-Programme ist die Aufteilung in Module nicht trivial, denn C-Programme bestehen im allgemeinen Fall aus mehreren Übersetzungseinheiten, die keine rein prozedurale Schnittstellen besitzen.

Die Aufteilung soll automatisch geschehen, damit der Entwickler nicht vor das Problem der Aufteilung in Module gestellt wird. Sonst wäre die leichte Portierbarkeit von existierendem Code nicht realisierbar.

5.3.1 Übersetzungseinheiten als Module

Die erste, in der vorangegangenen Diplomarbeit [11] verfolgte Variante der Aufteilung ist die Gleichsetzung eines Moduls mit der in C implementierten Übersetzungseinheit. Bei dieser Organisation ergibt jeder Compilerlauf ein Modul, das zur Laufzeit mit den anderen Komponenten des C-Programms interagiert. Dieses Konzept erlaubt die Benutzung des vorhandenen Assemblers ohne Modifikationen, denn jede Übersetzungseinheit wird als komplettes Modul vom Compiler ausgegeben. Alle global sichtbaren Funktionsdefinitionen müssen zu Schnittstellenfunktionen der Übersetzungseinheit gemacht werden. Dies erfordert bei den meisten existierenden Programmen sehr viele Schnittstellenfunktionen. Die beim MONADS-PC vorhandene Grenze von 32 schützbaren Schnittstellenfunktionen wird oft überschritten. Die Schnittstellenfunktionen mit Nummern ab 32 können von jedem aufgerufen werden, der eine Capability für das Modul hat.

Ein explizites Binden der Übersetzungseinheiten zum ausführbaren Programm erfolgt nicht. Weil keine Bindeinformationen in den Modulen vorhanden ist, kann dies auch nicht erfolgen. Dies ist eine Abweichung von der üblichen Benutzungsweise des Compilers. Bei allen vorhandenen Quelltexten wird ein manuelles Eingreifen in den vom Entwickler ausgearbeiteten Übersetzungsprozeß verlangt. Die in den allermeisten Fällen mitgelieferten `Makefile`-Dateien können nicht benutzt werden.

Die Bindung der Module zu einem lauffähigen Programm erfolgt in [11] zur Laufzeit. Der Startcode lokalisiert alle Teile des Programms. Zum Programm gehört auch die C-Bibliothek. Der Compiler hat für diesen Zweck Code zur Auflösung von Codeabhängigkeiten erzeugt. Die Module werden in den Verzeichnissen gesucht, die beim Start der C-Laufzeitumgebung spezifiziert wurden. Die gefundenen Module Capabilities und Schnittstellennummern der globalen Funktionen werden in einer Datenstruktur abgelegt. Der vom Compiler erzeugte Code zum Funktionsaufruf greift auf diese Bindeinformation zurück.

Die Realisierung dieser Konzepte ist relativ komplex, denn die oben angegebene generelle Idee funktioniert nur für Funktionsaufrufe. Globale Variablen müssen über einen anderen Mechanismus implementiert werden. Die prozedurale Schnittstelle erlaubt keine unmittelbaren Zugriffe eines Moduls auf die Variablen eines anderen Moduls. Es muß also noch eine Lösung für die Realisierung von globalen Variablen gefunden werden. Die offensichtlichste Möglichkeit, dieses Problem zu lösen, ist die Definition von Zugriffsfunktionen. Das führt in diesem speziellen Problem aber nicht zum Ziel. In C ist es erlaubt, in mehreren Übersetzungseinheiten ein und dieselbe globale Variable zu deklarieren. Es ist nicht klar, welches Modul die Variable enthalten muß. Üblicherweise ist es die Aufgabe des

Bindeprogramms, mehrfach definierte Symbole zu einem Symbol zusammenzufassen. Es müßten eigentlich mehrere Symbole vor der Programmausführung zusammengefaßt werden. Der Compiler kann dies wegen der getrennten Übersetzung nicht erledigen.

Die Abbildungen 5.1 und 5.2 illustrieren dieses Problem: die erste Übersetzungseinheit deklariert die zwei Variablen `items` und `table`. Die Variable `table` wird mit Anfangswerten vorbesetzt. Die zweite Übersetzungseinheit deklariert dieselben beiden Variablen, ohne sie zu initialisieren. Der Linker muß beide Vorkommen von `items` zu einem zusammenfassen und erkennen, daß für `table` die vordefinierten Werte benutzt werden müssen. Der C-Standard legt zusätzlich fest, daß `adjust_length` in der zweiten Übersetzungseinheit auch ohne explizite Deklaration eine globale Funktion ist.

```
int items;
char *table[7] =
   {"Anton", "Berta", "Fritz", "Otto",
    "Peter", "Udo", "Zenzi"};

void adjust_length(void)
{
   items = 7;
}
```

Abbildung 5.1: Erste Übersetzungseinheit mit globalen Symbolen

```
int items;
extern char *table[];

int main(void)
{
   int i;

   adjust_length();
   for (i = 0; i < items; i++) {
      printf("%s\n", table[i]);
   }
}
```

Abbildung 5.2: Zweite Übersetzungseinheit mit globalen Symbolen

In [11] wurde daher ein ganz anderer Ansatz gewählt. Der Start von Programmen muß wegen der dynamischen Bindung von Funktionen ohnehin über ein Laufzeitsystem erfolgen. Die globalen Variablen werden daher im Heap des Laufzeitsystemmoduls angelegt. Allen Teilmodulen werden Zeiger auf diesen Speicherbereich übergeben. Die Zahl und der Speicherplatzbedarf der einzelnen Variablen ist erst zur Laufzeit bekannt. Die Adressen der Variablen können aus

diesem Grund ebenfalls erst während der Initialisierung des Programms ermittelt werden. Für jede globale Variable, die eine Übersetzungseinheit benutzt, ist ein Verweis in das Segment der globalen Variablen erforderlich. Der Verweis wird vom Laufzeitsystem bei der Reservierung des Speichers für globale Variablen bestimmt und in ein Verzeichnis eingetragen. Jeder Zugriff auf globale Variablen benötigt einen zusätzlichen Indirektionsschritt.

Der komplexe Aufbau des Laufzeitsystems führte zu einigen Implementierungsproblemen. Es ist nicht einfach, eine Implementierung zu entwickeln, die allen möglichen Fällen gerecht wird. Wegen den vielen zu erstellenden Datenstrukturen mit Informationen über jedes Modul benötigt das Laufzeitsystem sehr viel Rechenzeit und Speicherplatz. Das verzögert den Programmstart. Die Ablaufgeschwindigkeit von Programmen ist stark vermindert, da alle Funktionsaufrufe über Modulgrenzen hinweg erfolgen und der Zugriff auf globale Variablen ineffizient ist.

Die Verwendung von Schnittstellenfunktionen für statische Funktionen verletzt die einzige in C-Programmen geltende Sichtbarkeitsregel für Bezeichner. Alle als `static` deklarierte Bezeichner sind nur in der definierenden Quelltextdatei sichtbar. Die Einhaltung dieser Regel muß von der Laufzeitumgebung wieder erzwungen werden. Es gibt keine Alternative zu dieser Umgehung des Schutzkonzeptes, denn es kann innerhalb einer Übersetzungseinheit jederzeit die Adresse einer statischen Funktion ermittelt und an andere Übersetzungseinheiten übergeben werden. Dies ist zwar unüblich, verhindert aber die Realisierung von statischen Funktionen als modulinterne Funktionen. Auch für statische Funktionen ist der Weg über eine echte statische Bindung und damit schnellere Ausführung blockiert. Schnittstellenfunktionen innerhalb eines Moduls können ebenfalls nur mittels der korrekten Module Capability aufgerufen werden.

Existierender Code berücksichtigt diese Konsequenzen für die Ablaufgeschwindigkeit nicht. Das äußert sich in der teilweise geringen Arbeitsgeschwindigkeit von Programmen. Diese Portierung, auch wenn sie den Konzepten des MONADS-PC am genauesten entspricht, ist in der Realität nur bedingt geeignet. Das zeigt sich deutlich bei der Übersetzung des Spiels Go, das in der Originalversion aus einem guten Dutzend sehr kleiner Quelltexte besteht. Diese Aufteilung macht die Übertragung auf den MONADS-PC und den Programmstart sehr aufwendig. Der Nachteil ist nur zu umgehen, wenn die einzelnen Programmteile manuell zu einem Quelltext kombiniert und übersetzt werden. Solche manuellen Eingriffe sind natürlich unerwünscht.

Positiv ist der Platzbedarf des Codes zu bewerten, da z.B. die C-Bibliothek nur einmal existiert und alle Programme automatisch darauf zurückgreifen. Auch auf andere Programmteile wäre das Prinzip anwendbar, dies muß aber schon bei der Entwicklung von Programmen berücksichtigt werden. Der Vorteil wird durch den für Zugriffe auf Funktionen und die globalen Variablen erhöhten Codeumfang etwas geschmälert.

Die Effizienz des Bindevorgangs und des Zugriffs auf globale Variablen könnte deutlich erhöht werden, wenn ein separater Bindevorgang die Referenzen schon

zur Übersetzungszeit auflösen würde, soweit sie ermittelt werden können. Das hat Auswirkungen auf den zu erzeugenden Code, z.B. müßte der Compiler die schon vorliegenden Module kennen. Bei gegenseitigen Abhängigkeiten zweier Übersetzungseinheiten könnte die Auflösung nur teilweise durchgeführt werden. Alternativ könnte die Codeerzeugung wie bisher erfolgen und ein nachgeschalteter Linker würde den Code anpassen. Diese Möglichkeit ist nur realisierbar, wenn symbolische Informationen in den erzeugten Modulen abgelegt werden könnten. Der Assembler müßte dafür stark erweitert werden. Beide Lösungen sind sehr aufwendig und erfordern umfangreiche Modifikation entweder des Compilers oder Assemblers. Der Gewinn wäre durch die beschränkten Optimierungsmöglichkeiten nicht sehr hoch.

5.3.2 Programme als Module

Alternativ dazu kann der Ansatz gewählt werden, ein ganzes Programm als Modul aufzufassen. Dazu ist es erforderlich, die verschiedenen Teile des Programms zu einem Ganzen zu binden. Dieser Weg wird seit langem von vielen Entwicklungssystemen (u.a. allen C-Compilern unter UNIX) benutzt, um aus den vielen Einzelteilen eines großen Projekts das ausführbare Programm zu erzeugen.

Der Compiler erzeugt dabei für jede Übersetzungseinheit eine Objektdatei, die den assemblierten Quelltext enthält. Die Objektdatei enthält zusätzlich eine Liste der definierten und undefinierten globalen Symbole dieser Übersetzungseinheit. Der Linker kombiniert die verschiedenen Objektdateien dann unter Auflösung der noch bestehenden gegenseitigen Referenzen. Der Linker „ändert“ also die Teile des Programmes so ab, daß das Ergebnis dem entspricht, was bei der Übersetzung des Programms in einem Stück herausgekommen wäre. Die Befehle selbst werden nicht geändert, nur die Operanden und anderen Referenzen werden durch die tatsächlichen Werte ersetzt.

Dieser Ansatz ist im Hinblick auf die Ausführungszeit effizienter, denn alle Bindevorgänge laufen zur Übersetzungszeit ab. Der Nachteil dieser statischen Bindung ist der größere Programmcode, denn die benötigten Teile der C-Bibliothek werden dem Programm einfach hinzugefügt.

Ein wichtiger Vorteil der statischen Bindung ist die bei allen C-Programmen identische Aufrufschnittstelle. Fertig gebundene C-Programme benötigen nur einen Einsprungpunkt, die Funktion `main`. Diese Funktion hat immer die gleichen Parameter- und Rückgabetypen. C-Programme sind also Implementierungen einer einzigen Klasse. Das vermeidet die Vielzahl der Klassen, die bei der Realisierung von Übersetzungseinheiten durch Module anfallen würden.

Es ist möglich, die beiden Bindeverfahren zu kombinieren. Dabei werden Teile der C-Bibliothek in ein externes Modul verlagert, das dann nur noch einmal auf dem System gehalten werden muß. Die Verlagerung der gesamten C-Bibliothek lohnt sich nicht. Alle von der jeweiligen Bibliotheksfunktion benutzten globalen Variablen des Programms müssen an das Bibliotheksmodul übergeben werden

und ein externer Aufruf ausgeführt werden. Anschließend müssen die globalen Variablen des Programms aktualisiert werden. Viele Bibliotheksfunktionen sind sehr klein und bei diesen wäre der Zeitaufwand für den Aufruf größer als die Ausführungszeit des eigentlichen Codes. Allein die Konvertierung auf eine andere Parameterübergabekonvention benötigt teilweise mehr Zeit als die Ausführung der eigentlichen Funktion.

Die effektive Benutzung der Möglichkeit, den Bibliothekscode aus den einzelnen Programmen in ein Bibliotheksmodul zu verlagern, verlangt jedoch die Änderung der Sprachdefinition. In C können nur Programme implementiert werden, die genau eine Schnittstellenfunktion besitzen. Es ist nicht möglich, auf diesem Weg allgemeine MONADS-Module zu erstellen, da die Parameter von `main` vorgegeben sind. Die Implementierung von Code-Modulen oder Typemanagern benötigt Unterstützung für den Aufruf beliebiger Funktionen. Der Startcode für C-Programme ist für diese Art der Programmierung nicht geeignet. Eine angepaßte Variante müßte ganz anders strukturiert sein, um die Übergabe beliebiger Parameter zuzulassen. Die besondere Behandlung von `main` könnte entfallen. Vorzugsweise wäre die Implementierungsmöglichkeit allgemeiner Module gleich mit der Möglichkeit verbunden, Schnittstellenfunktionen zu definieren. So würden keine unerwünschten Funktionen exportiert. Einige Funktionen in der C-Bibliothek sind global sichtbar, weil sie an mehreren Stellen in der Bibliothek benötigt werden. Sie stellen keine Schnittstellenfunktionen der C-Bibliothek dar. Der Aufruf solcher Funktionen ist unerwünscht und sollte unterbunden werden.

Damit C-Code beliebige Schnittstellen unterstützen kann, muß die Implementierung von Schnittstellenfunktionen möglich sein. Der MONADS-PC schreibt die Aufrufkonvention von Schnittstellenfunktionen großteils vor. Der Aufruf von in C implementierten Schnittstellenfunktionen erfordert daher Konvertierungscode, der die MONADS-Aufrufkonvention in die Aufrufkonvention von C-Programmen umsetzt. Anders als beim Aufruf anderer Module von C-Code aus ist dies vom Compiler effizient realisierbar. Der Ansatz dazu wäre sehr ähnlich zu der Erzeugung des Konvertierungscodes zum Aufruf von Modulen. Es gibt eine 1:1-Zuordnung zwischen Schnittstellenfunktionen und den C-Funktionen, die diese implementieren. Es kann keine Konflikte durch die unmittelbare Generierung des Konvertierungscodes durch den Compiler geben.

Für diese Diplomarbeit sind Erweiterungen des Compilers zur Implementierung von Modulen nicht geplant. Einerseits ist der Änderungsaufwand am Compiler relativ hoch und andererseits sollte eine Portierung des Compilers eigentlich keine Spracherweiterungen erfordern. Die Implementierung von Modulen ist später jederzeit nachholbar, denn die Anforderungen der Erweiterung beeinflussen die Portierung von GNU C nicht.

Es gibt noch eine andere Möglichkeit, Teile der C-Bibliothek in ein Modul zu verlagern. Das Bibliotheksmodul könnte in MONADS-Pascal implementiert werden. Dann wäre keine Anpassung des C-Compilers oder Sprachänderung erforderlich. Dieser Weg erzwingt eine Neuimplementierungen aller C-Bibliotheksfunktionen, die in das Modul verlagert werden.

5.4 Speicheradressierung

Eine der wichtigsten Dinge für die Portierung des GNU C-Compilers ist die Definition einer einheitlichen Zeigerdarstellung. Der Compiler behandelt alle Speicherreferenzen auf die gleiche Weise und erwartet, daß es möglich ist, die Adresse von jedem Speicherobjekt gleich darzustellen. Beispielsweise bedeutet das, daß die Adresse einer lokalen Variable auf dem Stack oder einer Variable auf dem Heap gleich darstellbar sein muß. Die Beschaffenheit des Zeigeraufbaus ist irrelevant, nur die einheitliche Darstellung ist wichtig.

Der C-Standard definiert, daß jeder Zeigerwert in einen anderen konvertierbar sein muß. Das Ergebnis der Dereferenzierung dieses konvertierten Zeigers ist nicht definiert. Jedes Verhalten ist erlaubt, Programmabbruch eingeschlossen. Der Zeiger muß aber wieder in den ursprünglichen Typ zurückkonvertierbar sein, ohne daß Information verloren geht. In Bild 5.3 ist dies illustriert – der erste `printf`-Befehl muß das Ergebnis 5 ausgeben. Die Ausgabe des zweiten `printf`-Befehls ist hingegen undefiniert.

```
#include <math.h>

typedef double (*func) (double);

int main(void)
{
  func f, f2;
  int *p;

  f = sqrt; /* Zeiger auf die sqrt-Funktion */
  p = f;
  f2 = p;
  printf("sqrt(25) = %f\n", f2(25));
  printf("*p = %i\n", *p); /* nicht definiert */
}
```

Abbildung 5.3: Einheitliche Zeigerdarstellung in C

Alle in einem Programm zur Ausführungszeit vorhandenen Speicherobjekte müssen eindeutig identifizierbar sein, damit die in der Programmiersprache C üblichen Konstrukte (z.B. Operationen auf Zeigern, Bestimmung der Adresse einer beliebigen Variable oder Funktion) überhaupt implementiert werden können. Sie müssen deshalb in einem Adreßraum liegen. Der Begriff Adreßraum ist hier nicht im Sinne eines „Address Space“ der MONADS-Architektur benutzt, sondern bezeichnet den gesamten adressierbaren Speicher. Der Adreßraum eines laufenden Programms umfaßt also alle zum Programm gehörenden Speicherbereiche. Im Gegensatz dazu ist ein Address Space der MONADS-Architektur eine Strukturierung des Speichers in zusammengehörende Bereiche. Address Spaces enthalten z.B. sämtliche Instanzvariablen eines Moduls oder einen Stack.

Der Begriff des Adreßraums bietet die Möglichkeit, die Speicherbereiche von verschiedenen laufenden Programmen vollständig voneinander zu trennen. Diese strikte Trennung des adressierbaren Speichers verschiedener Programme stellt sicher, daß kein Programm Zugriff auf Daten erhält, die zu einem anderen Programm gehören. Dies ist die Basis des Zugriffsschutzes auf Architekturebene. Die einzige Möglichkeit des Datenaustauschs zwischen Programmen sind in der MONADS-Architektur die Instanzvariablen von Modulen. Module können mit verschiedenen Verfahren vor unberechtigtem Zugriff geschützt werden. Hier wird die Ähnlichkeit von Modulen in der MONADS-Architektur und Dateien in konventionellen Systemen besonders deutlich.

5.4.1 Segmentierte Adressen

Die auf der MONADS-Architektur vorgesehene Art, einen Address Space zu strukturieren, ist die Segmentierung. Die Architektur bietet Segmente beliebiger Größe in verschieden lang beibehaltenen Datenbereichen an. Näheres hierzu ist in Kapitel 2 zu finden.

Segmente enthalten typischerweise semantisch zusammengehörende Daten und können beliebig mit Verweisen auf andere Segmente zu komplexen Datenstrukturen kombiniert werden. Die einzige hierbei zu beachtende Einschränkung ist, daß kein Verweis in einer Datenstruktur auf eine andere Datenstruktur kürzerer Lebensdauer definiert werden kann. Dies würde potentiell zu Problemen bei späteren Zugriffen führen, wenn die Daten mit kürzerer Lebensdauer bereits gelöscht wurden.

5.4.2 Flache Adressen

Konventionelle Systeme unterstützen virtuellen Speicher auf Basis von Seiten fester Größe. Manche Architekturen setzen darauf noch Segmente auf. Segmente werden in diesen Architekturen praktisch immer zusätzlich zu seitenorientierter Speicherverwaltung unterstützt. Wenn Segmente angeboten werden, so benutzen die heute eingesetzten Betriebssysteme sie meist nicht. Es wird ein Segment pro Prozeßadreßraum definiert, in dem keine Strukturierung mehr vorgenommen wird. In vielen konventionellen Systemen wird wegen der Probleme, die bei Verwendung von segmentierten Adressierungsarten auftreten, ganz auf Segmentierung verzichtet. Die Zahl der Segmente ist meist stark begrenzt und die Zeigerdarstellung wird durch die zusätzlich notwendige Angabe des Segments aufwendiger.

Das Problem der seitenorientierten Verwaltung des Hauptspeichers ist ein starker Verlust von Robustheit. Ohne zusätzliche Mechanismen sind jetzt alle Adressen gültig und fehlerhafte Zeiger können nicht erkannt werden. Konstanten können ohne den Einsatz von Schutzkonzepten auf Seitenebene nicht vor versehentlichem Überschreiben geschützt werden. Ein weiteres Element der Robustheit

ginge so verloren. Die Verlagerung des Zugriffsschutzes auf die Seitenebene verschwendet aber auch Speicher, denn eine nicht beschreibbare Seite ist nur für Konstanten benutzbar.

Damit wenigstens eine gewisse Robustheit gegenüber falschen Zeigern erreicht wird, ist die erste Seite des Prozeßadreßraums meist nicht zugreifbar. Vom Betriebssystem wird eine Obergrenze des benutzbaren Speichers (d.h. die letzte zugreifbare Seite) mitgeführt, die von speziellen Aufrufen erhöht bzw. erniedrigt werden kann. Die eventuell vorhandenen Möglichkeiten, dies eleganter über Segmentierung zu lösen, werden also nicht genutzt. Bei Systemen, die Segmente und seitenorientierte Speicherverwaltung unterstützen, ist die Verwendung seitenorientierter Schutzkonzepte keine Ideallösung. Der Wegfall der Segmentierung verlangt eine unnötig komplexe Speicherverwaltung auf Seitenebene.

5.4.3 Segmentierung mit „flachen“ Zeigern

In den beiden vorigen Abschnitten wurden zwei verschiedene Arten der Strukturierung eines Adreßraums beschrieben. Die Verwendung von Segmenten bietet den Vorteil der problemorientierten Strukturierbarkeit des Speichers. Diese Art der Speicherverwaltung hat aber durch die zweigeteilte Zeigerdarstellung (Segment, Offset) Nachteile bei der effizienten Nutzung des Speicherzugriffs. Der Optimierer sollte möglichst oft erkennen, daß der Segmentteil zweier Zeiger identisch ist, um wiederholte Lade- bzw. Speicheroperationen von Segmentregistern zu vermeiden. Die frei verfügbaren C-Compiler unterstützen solche strukturierten Zeiger aber nicht. Sie müssen deshalb als unstrukturierte Zeiger implementiert werden. Dies macht die auf segmentierten Systemen unbedingt notwendigen Optimierungen der Segmentzugriffe unmöglich.

Die Idee der Segmentierung bietet aber auch bei eingeschränkter Nutzung noch genügend Vorteile, um sie nicht sofort aufzugeben. Es müßte ein Kompromiß zwischen den beiden Extremen gefunden werden. Die Beschränkung auf wenige Segmente (beispielsweise Stack, Daten/Heap und Konstanten) bietet zwar etwas weniger Robustheit, die Zugriffe auf die Segmentdeskriptoren können aber dadurch generell eliminiert werden. Sie könnten in der Initialisierungsphase des Programms in die CPU-Register geladen werden und für die gesamte Programmlaufzeit konstant bleiben.

Die Robustheit leidet etwas unter der groben Granularität der Segmente. Die Auswirkungen sind aber nicht so stark wie bei seitenorientierten Systemen. Die Zeiger dieser Mischform enthalten noch semantische Information über den Aufbau des Speichers. Die Eigenschaften segmentierter Systeme bleiben teilweise erhalten. Der Schutz vor Zugriffen über die Grenzen von Segmenten hinaus oder der Schutz vor dem Überschreiben von Konstanten ist weiterhin gewährleistet.

Auf dem MONADS-PC wäre es zusätzlich noch möglich, für den Heap und Stack zwar große Segmente zu reservieren, während der Ausführung des Programms

aber immer mit eingeschränkten Segmenten (refined segments) zu arbeiten. Eingeschränkte Segmente erlauben nur den Zugriff auf einen Teil des entsprechenden Segments. Der Bereich könnte durch den Stackzeiger oder die letzte reservierte Speicheradresse auf dem Heap begrenzt werden. Auf diese Weise können noch mehr falsche Zeiger erkannt werden, ohne die Verwaltungskosten nennenswert in die Höhe zu treiben.

5.5 Getrennte Übersetzung

Beinahe alle existierenden C-Programme erwarten implizit, daß Programme in einzelne Quelltexte aufgeteilt werden, die separat übersetzt werden können. Die übersetzten Teile des Programms müssen anschließend zu einem kompletten ausführbaren Programm gebunden werden. Diese Vorgehensweise bei der Programmentwicklung soll selbstverständlich unterstützt werden. Die Aufteilung von Programmen in Übersetzungseinheiten ist die einzige Strukturierungsmöglichkeit für Quelltexte in der Sprache C.

Der Compiler erzeugt meist keine kompletten Programme, sondern nur sogenannte Objektdateien. Die Objektdateien werden eigentlich vom Assembler erstellt, denn der Compiler selbst erzeugt nur Assemblerquelltext. Es wurde schon erwähnt, daß der existierende MONADS-Assembler nicht für solche Programmentwicklung geeignet ist. Es muß also ein System entwickelt werden, das erlaubt, getrennt übersetzte Programmteile abzuspeichern und später zu einem Programm zu binden.

Die Neuentwicklung eines Assemblers, der alle Anforderungen erfüllt und direkt in MONADS-PC-Maschinencode übersetzt, wurde von Anfang an als nicht praktikabel und unerwünscht verworfen. Der Zeitaufwand und die Komplexität wären zu hoch gewesen. Es muß ein Verfahren entwickelt werden, das einerseits die getrennte Übersetzung in einzelne Objektdateien unterstützt, aber andererseits einen Assembler benutzt, der dafür nicht geeignet ist. Die Lösung ist die Verlagerung des Aufrufs des MONADS-Assemblers hinter den Bindevorgang, wenn alle Teile zu einem kompletten Programm kombiniert sind. Die Abfolge der Verarbeitung, die sich hier ergibt, ist in Bild 5.4 dargestellt.

Es muß also ein Assembler neu entwickelt werden, der aus seinen Eingabequelltexten symbolische Informationen extrahiert und den ansonsten nahezu unveränderten Eingabequelltext mitsamt diesen Informationen in einer Objektdatei ablegt. Sämtliche definierten Symbole aus dem Eingabequelltext werden am Anfang der Objektdatei gebündelt abgelegt. Das vereinfacht den Bindevorgang. Bestimmte Teile des Eingabequelltexts, etwa die Deklaration eines Bezeichners als globales Symbol, können ebenfalls am Anfang der Objektdatei abgelegt werden. Diese Information ist nur für den Linker relevant und wird nicht an den MONADS-Assembler weitergegeben. Der MONADS-Assembler weiß nichts damit anzufangen, da die Eingabesyntax solche Deklarationen nicht vorsieht. Weiterhin muß der Eingabequelltext des neu erstellten Assemblers in Abschnitte zerlegt

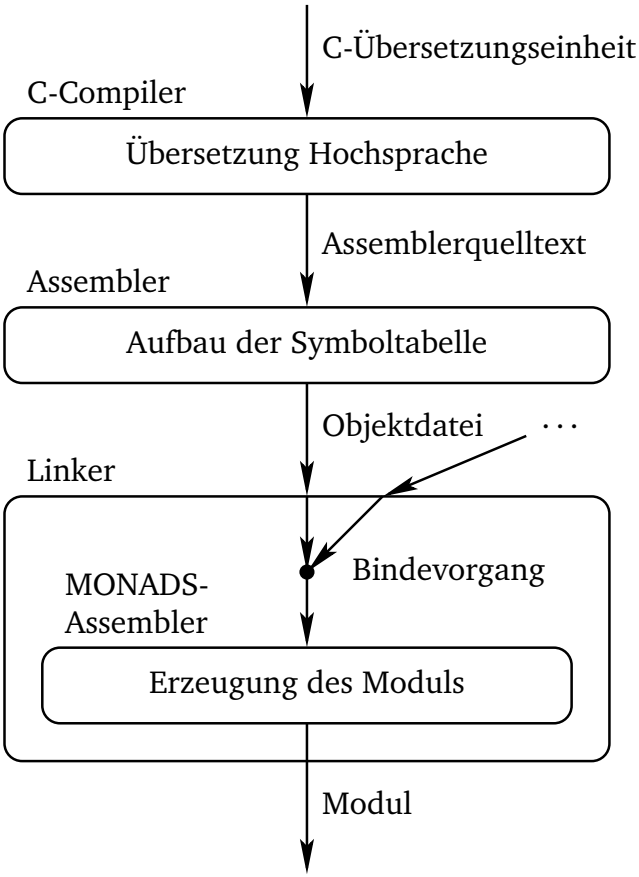

Abbildung 5.4: Schritte vom Quelltext zum Modul

werden, damit die verschiedenen Bestandteile einer Objektdatei (Daten, Konstanten, Programmcode) später in die richtige Reihenfolge gebracht werden können. Generell könnten diese Aufgaben direkt vom Linker übernommen werden, die Objektdateien wären dann identisch zu den Assemblerquelltexten. Von der Bearbeitung des Assemblerquelltexts profitiert aber nicht nur der Linker, sondern auch die Bibliotheksverwaltungsprogramme. Der zusätzliche Aufwand für die Implementierung eines Assemblers ist daher gerechtfertigt.

Der Linker hat die Aufgabe, mehrere Objektdateien zu kombinieren. Diese Aufgabe ist durch die textuelle Darstellung von Objektdateien etwas ungewöhnlich. Ein Linker ist normalerweise zur Auflösung globaler Symbole da, die in mehreren Objektdateien benutzt und in einer Objektdatei definiert werden. Dieser Fall ist bei den vom Assembler erstellten Objektdateien sehr einfach: globale Symbole benutzen in allen Objektdateien den gleichen Namen. Der MONADS-Assembler, der am Ende des Bindevorgangs aufgerufen wird, bewältigt diesen Fall problemlos. Das Problem liegt bei den lokalen, nur in einer Objektdatei sichtbaren Symbolen. Diese kommen potentiell mehrfach in Programmen vor, müssen aber für jede Objektdatei getrennt behandelt werden.

Die Erzeugung eindeutiger lokaler Symbole ist also das einzige verbleibende Problem. Eindeutige Symbole können an mehreren Stellen vergeben werden, aber

nicht alle sind dafür gleichermaßen geeignet. Der Assembler könnte eindeutige Symbole z.B. durch Anhängen der Uhrzeit generieren. Das birgt aber die Gefahr, daß unter bestimmten Umständen doch gleiche Symbole in verschiedenen Objektdateien vorhanden sein können. Das gebundene Programm wäre vom MONADS-Assembler nicht übersetzbar, weil Symbole mehrfach definiert sind. Das kann zwar durch Hinzufügen weiterer Informationen zu den Symbolnamen vermieden werden, die Symbolnamen werden dadurch aber extrem lang. Besser geeignet erscheint die Erzeugung eindeutiger Symbole im Linker, denn dieser hat Informationen über alle Objektdateien vorliegen. Eindeutige Symbole können durch einfaches Durchnumerieren der lokalen Symbole mit gleichem Namen erzeugt werden.

Zusätzlich zum Binden von Objektdateien sollte der Linker Bibliotheken unterstützen, damit die Benutzung von Funktionen aus der C-Bibliothek möglichst einfach wird. Bibliotheken werden als eine Sammlung von Objektdateien implementiert, die jedoch etwas anders behandelt werden als einzelne Objektdateien. Bei Bibliotheken werden ausschließlich die benötigten Objektdateien, die ein bisher undefiniertes Symbol definieren, zum Programm hinzugebunden. Einzelne Objektdateien dagegen werden ohne weitere Prüfungen zum fertigen Programm hinzugebunden. Die Sonderbehandlung von Bibliotheken reduziert die Größe des Teils einer Bibliothek, der zu einem Programm dazugebunden wird, auf das unbedingt nötige Minimum. Diese Optimierung ist bei einzelnen Objektdateien nicht erforderlich, da sie vom Entwickler des Programms explizit vorgegeben werden. Es ist daher sehr einfach, nicht benötigte Teile wegzulassen.

Die Erstellung von Bibliotheken erfordert einige kleine, vergleichsweise wenig komplexe Programme, die Objektdateien zu Bibliotheken hinzufügen bzw. löschen können. Die Bedienung der Programme soll den vergleichbaren UNIX-Programmen entsprechen.

5.6 Aufruf anderer Module

Der Aufruf anderer Module ist eine unbedingt notwendige Funktion auf dem MONADS-PC. Ohne diese Möglichkeit kann nicht mit dem Betriebssystem kommuniziert werden. Selbst so einfache Aufgaben wie Bildschirmausgaben sind ohne den Aufruf anderer Module unmöglich. Es muß also ein Weg gefunden werden, mit akzeptablem Aufwand und angemessener Geschwindigkeit Module aufrufen zu können.

Dabei entstehen einige Schwierigkeiten: der MONADS-PC verwendet eine andere Aufrufkonvention (siehe [8]) und spezielle Befehle, um Schnittstellenfunktionen externer Module aufzurufen. Es gibt auch einige Datenstrukturen, die ausschließlich für solche Aufgaben verwendet werden, z.B. Module Capabilities oder Module Call-Segmente. Da auf diesen Strukturen der Schutz basiert, können zur Manipulation nur bestimmte spezielle Maschinenbefehle eingesetzt werden, die nur die erlaubten Operationen durchführen. Der C-Compiler kann diese Befehle

nicht einsetzen, weil sie Datenstrukturen manipulieren, die ihm nicht bekannt sind.

Da der Compiler grundsätzlich nur eine Aufrufkonvention unterstützt, kann der Aufruf von anderen Modulen nicht direkt durch vom Compiler generierten Code erfolgen. Es ist zwar möglich, durch zusätzliche Angaben bei der Funktionsdeklaration kleine Änderungen am vom Compiler generierten Aufrufcode zu erreichen. Für die Erzeugung des Modulaufrufcodes reichen diese Möglichkeiten aber nicht aus. Das Haupthindernis ist die fehlende Adressierbarkeit der Parameter mit der in C-Programmen benutzten Adressierungsart.

Eine mögliche Lösung ist die Verwendung eines kleinen Unterprogramms für jeden Aufruf eines Moduls, dessen einzige Aufgabe es ist, die Aufrufkonvention zu konvertieren. Idealerweise erfolgt die Generierung dieses Unterprogramms automatisch, denn bei manueller Umsetzung ist das Risiko von Fehlern sehr hoch. Die Aufrufkonvention erfordert teilweise komplexe Datenstrukturen, beispielsweise um Strings zu repräsentieren. Der Compiler unterstützt z.B. die für Strings benutzte Adressierung von Segmenten nicht. Der Konvertierungscode muß also mindestens teilweise in Assembler implementiert werden.

Klassendefinitionen enthalten alle für die Generierung des Konvertierungscodes notwendigen Informationen. Sie wären eine geeignete Eingabe für ein Programm, das den Konvertierungscode automatisch erzeugt. Leider können Pascal-Klassendefinitionen auch noch andere Informationen enthalten, die schwer zu parsen sind und für die Erzeugung des Konvertierungscodes irrelevant sind. Darüberhinaus sollte ja auch für C eine Art Klassendefinition vorhanden sein, damit die Schnittstellenfunktionen und Parametertypen bekannt sind. Diese „C-Klassendefinitionen“, die viel weniger überflüssige Information enthalten, sind für die Erzeugung des Konvertierungscodes ausreichend. Es wurde wegen des daraus entstehenden Aufwandes (und der seltenen Änderung von Schnittstellen) darauf verzichtet, den in Bild 5.5 dargestellten ersten Konvertierungsschritt zu automatisieren. Dies kann später problemlos nachgeholt werden, wenn der Aufwand für die manuelle Konvertierung zwischen den Klassendefinitionen zu hoch wird.

Wie soll diese in Bild 5.5 dargestellte automatische Erzeugung erfolgen, wenn der Compiler weder die Befehle für Schnittstellenaufrufe, noch die Adressierungsarten für den Zugriff auf die Parameter kennt? Der fehlende Code muß ganz oder (bevorzugt) teilweise in Assembler implementiert werden.

Der GNU C-Compiler bietet für solche Aufgaben eine Spracherweiterung an, die es erlaubt, relativ bequem solche speziellen Operationen direkt als Assemblerquelltext zu schreiben. Die Unterstützung beschränkt sich nicht auf die bloße textuelle Einbindung, sondern es können im Assemblerquelltext symbolische Zugriffe auf Variablen und andere dem Compiler bekannte Objekte durchgeführt werden. Dies erlaubt beispielsweise, vom Compiler bestimmte Registerbelegungen erzeugen zu lassen, die der eingesetzte Code benötigt. Dies ist für optimierende Compiler sehr wichtig, denn es wird nicht garantiert, daß lokale Variablen

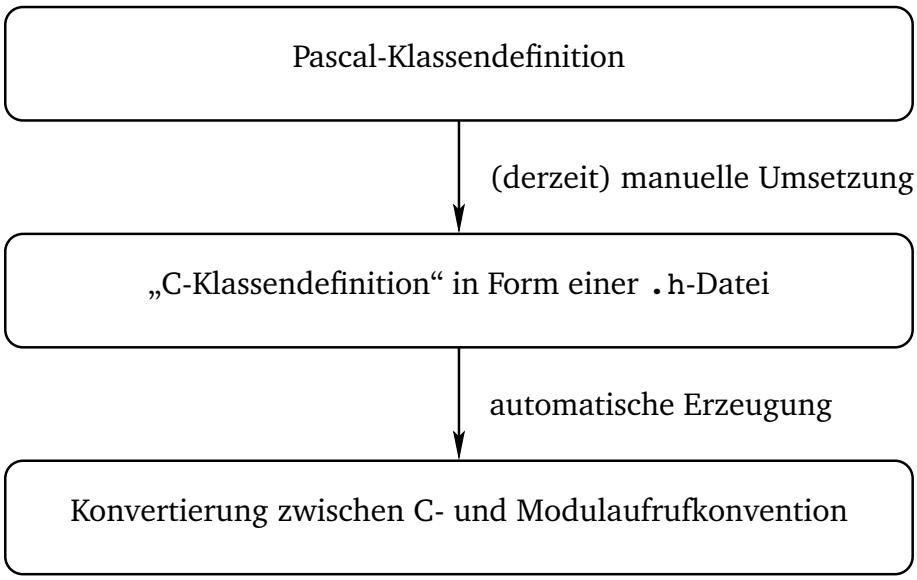

Abbildung 5.5: Erzeugung des Konvertierungscodes für Module

im Stack jederzeit auf dem aktuellen Stand sind. Es kann sogar sein, daß sie über die ganze Laufzeit in einem Register gehalten werden und auf dem Stack überhaupt kein Platz reserviert wurde. Der Compiler ist der einzige, der weiß, wie gerade auf Variablen zugegriffen werden muß. Andererseits kann er durch symbolische Verweise auch besser optimieren, denn die veränderten Register oder gemeinsamen Ausdrücke können zur Codeverbesserung herangezogen werden.

Diese Spracherweiterung des GNU C-Compilers wird durch zwei spezielle „Funktionen“, `asm()` und `asm volatile()` realisiert. Es sind keine Funktionen im üblichen Sinn, sondern Anweisungen an den Compiler, entsprechenden Code in den Assemblerquelltext zu schreiben. Die Angabe der Parameter (Ausgabeoperanden, Eingabeoperanden und geänderte Register) erfolgt durch Doppelpunkte abgetrennt. Die Funktion `asm` akzeptiert nur die ersten beiden Parameter, kann also nicht ausdrücken, daß andere Register als die Parameter benötigt bzw. verändert werden. Die veränderten Register können bei `asm volatile` im dritten Parameterspezifikationsteil vermerkt werden.

Beispiele:
```
asm("ldi %0,(c3)[%1]" : "=D" (x) : "D" (off));
asm volatile("call" : "=a" (r) :: "ax","i1","i2");
```

Das erste Beispiel zeigt dem Compiler an, daß der angegebene Maschinenbefehl je einen Ausgabeoperanden und einen Eingabeoperanden hat. Der Compiler wertet den Assemblertext nur für die Ersetzung der Operandenplatzhalter durch die entsprechenden realen Operanden aus. Operandenplatzhalter sind durch das Zeichen „`%`“ anzugeben, gefolgt von der Nummer des Operanden in der Parameterspezifikation. Der Ausdruck nach dem ersten Doppelpunkt gibt die Beschreibung der Ausgabeoperanden an (hier, daß ein Indexregister gesetzt wird), der Ausdruck nach dem zweiten Doppelpunkt ist die Beschreibung der Eingabeoperanden. Die Beschreibungen bestehen aus je zwei Teilen, der Bedingung und dem

Namen der Variablen, die gelesen oder geschrieben werden soll. Die Bedingung für die erlaubten Operandenarten wird exakt gleich wie in der Maschinenbeschreibung angegeben, für genauere Details siehe Abschnitt 6.1 und [10]. Das zweite Beispiel gibt analog einen Befehl an, nach dem der Compiler Code erzeugen soll, der den Wert des Akkumulators in der Variablen `r` ablegt. Die Register `ax`, `i1` und `i2` werden durch den Befehl zerstört, deshalb sorgt der Compiler dafür, daß diese Register nicht über diesen Befehl hinweg Daten aufbewahren.

Diese komfortable Möglichkeit, Assemblercode mit Hochsprachenelementen zu mischen, läßt sich sehr gewinnbringend für die automatische Erzeugung des Konvertierungscodes für die Parameterübergabe anderer Module einsetzen. Auf diese Weise läßt sich auch Code für Aufgaben schreiben, die in GNU C sonst nicht implementierbar wären. Ein Beispiel ist die in Bild 5.6 gezeigte automatisch generierte Konvertierungsfunktion für die Schnittstellenfunktion `readch` der Klasse `basic_text`.

Der vom Generatorprogramm erzeugte Code ist für den Compiler zwar nur eine „Black Box“, von der er nur die Eingabewerte und Ausgabewerte kennt. Diese abstrakte Sicht erspart aber u.a. die manuelle Implementierung der Zugriffe auf lokale Variablen. Es ist nicht notwendig, Kenntnisse über die Aufrufkonventionen von C-Funktionen in den erzeugten Konvertierungscode einfließen zu lassen. Der Compiler bleibt die einzige Instanz, die über seine eigenen Aufruf- und Parameterübergabekonventionen bescheid wissen muß. Veränderungen an der Stackorganisation des Compilers können ohne Rücksicht auf existierenden Code vorgenommen werden.

Bei der Generierung des Konvertierungscodes kann leicht Code für die effiziente Verwaltung der Module Call-Segmente hinzugefügt werden. Bei `open`-Aufrufen kann automatisch ein freies Module Call-Segment angelegt werden, das beim `close`-Aufruf wieder freigegeben wird.

Aufwendig ist hingegen die Verwaltung der Parametersegmente beim Aufruf von Schnittstellenfunktionen. Speziell die Übergabe von Strings ist vergleichsweise teuer, da wegen der anderen Stringdarstellung ein Umkopieren nicht vermieden werden kann. Strings können wegen der Benutzung von Segmentzeigern in der Repräsentation nicht im MONADS-Stack übergeben werden, deswegen muß auf den Heap ausgewichen werden. Damit aber nicht so viel Speicher verschwendet wird[6], muß eine eigene Segmentwiederverwendungsstrategie implementiert werden. Dieses Konzept wurde ursprünglich im Pascal-Compiler eingesetzt, bewährt sich aber auch an dieser Stelle. Bei Strings ist dieses Verfahren besonders aufwendig, da Segmente für Strings verschiedene Längen besitzen können. Dies verursacht beim Aufruf von Schnittstellenfunktionen mit Stringparametern einen nicht zu vernachlässigenden Zeitaufwand.

All diese Überlegungen werden in einem Programm zur Generierung der Konvertierungsfunktionen realisiert. Der Generator sollte für jede Schnittstellenfunktion eines Moduls eine eigene Datei mit dem Konvertierungscode dieser Schnittstel-

[6] Der vorgesehene Garbage Collector ist noch nicht implementiert.

```
#include <stdlib.h>
#include <string.h>
#define _MPC_STUB
#include <mpc/mpc.h>
int basic_text_readch(mcallseg mcs, int *p0)
{
  register int retval;
  asm("loadcb c11,#16");
  asm("loadc c13,XPmcs(c11)\n\t"
      "loadc c13,(c13)[%0]\n\t"
      "precall (c13),#4,c12" :: "D" ((unsigned)mcs&0xfffffff));
  switch ((unsigned)p0&0xf0000000) {
  case 0x10000000:
    asm("movc c0,c13");
    break;
  case 0x20000000:
    asm("movc c1,c13");
    break;
  case 0x30000000:
    asm("movc c2,c13");
    break;
  default:
    asm("invc c13");
  }
  asm("prefr #0,c13,%0,#4" :: "a" ((unsigned)p0&0xfffffff));
  asm volatile("sti i0,___i0save(cx)\n\t"
               "sti i3,___i3save(cx)\n\t"
               "call\n\t"
               "ldi i3,___i3save(cx)\n\t"
               "ldi i0,___i0save(cx)" : "=a" (retval)
                                      :: "ax","i1","i2");
  return retval;
}
```

Abbildung 5.6: Konvertierungscode für `readch` in Klasse `basic_text.h`

lenfunktion erzeugen. So können die übersetzten Schnittstellenfunktionen in einer Bibliothek abgelegt werden. Nur die wirklich benötigten Konvertierungsfunktionen werden zum Programm hinzugebunden.

5.7 Speicherverwaltung

Die Speicherverwaltung ist in groben Zügen durch die Kombination von Segmentierung und flachen Adressen festgelegt. Da diese zwar auf Segmenten basiert, in diesem Schema aber nur wenige Segmente erlaubt sind, kann nur eine sehr grobe Strukturierung des Adreßraums erfolgen.

Das Konzept der Aufteilung des Speichers (dargestellt in Bild 5.7) ist sehr einfach: es wird ein Segment für die Konstanten, ein Segment für globale Daten sowie den Heap und ebenso ein Segment für den Stack angelegt. Die grobe Aufteilung hat die oben diskutierten Nachteile bei der Robustheit, erlaubt aber schnellen Code durch die vollständige Eliminierung von Segmentoperationen. Der Compiler muß sich nicht mit der Verwaltung von Capability-Registern befassen. Es werden nur die von der Adressierungsart vorgegebenen Capability-Register benutzt, die übrigen sind frei verfügbar.

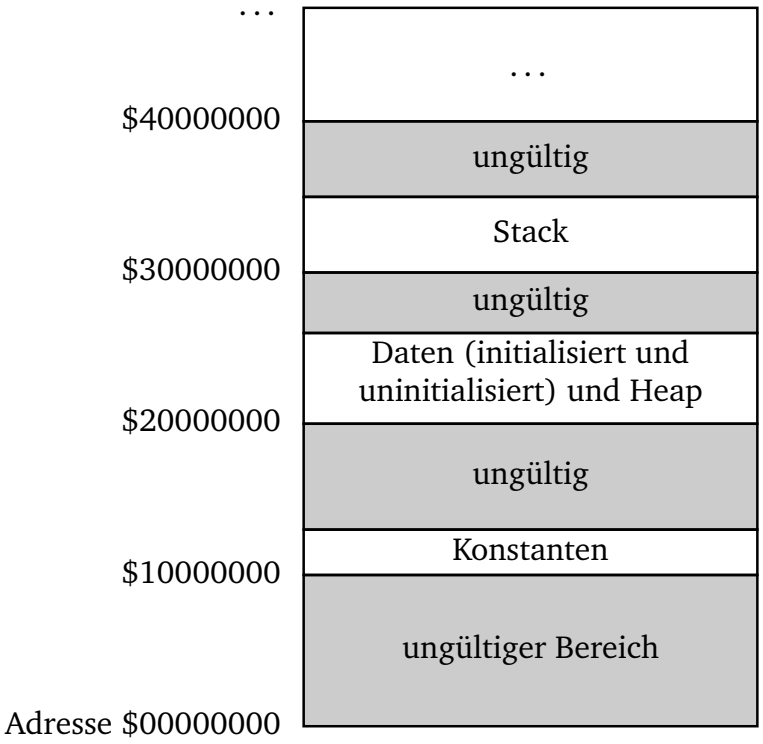

Abbildung 5.7: Speicheraufteilung in einem C-Programm

Der Heap wird auf konventionelle Weise verwaltet. Es ist also in der C-Bibliothek eine Verwaltung von Adreßbereichen implementiert, die dafür sorgen muß, daß sich die Fragmentierung in Grenzen hält.

Weiter oben wurde schon festgestellt, daß der Stack des MONADS-PC nicht für die Verwendung in C-Programmen geeignet ist, da potentiell Verweise auf unterschiedliche Stackrahmen in Variablen abgelegt sein können. Dies ist durch die sehr begrenzte Zahl der Segmente, die von der Adressierungsart vorgegeben wird, nicht möglich. Der Stack wird deshalb in einem eigenen Segment implementiert. Die Verwaltung wird vollständig durch vom Compiler generierten Code übernommen. Dies erfordert (potentiell) zwei Register, eines für den Zeiger auf das Ende des Stacks, und eines für den Zeiger auf den aktuellen Stackrahmen.

Durch die Verwaltung des Stacks durch den Compiler ist es leicht möglich, den Aufrufmechanismus für Funktionen weit effizienter zu gestalten als der von der

Architektur vorgesehene Mechanismus. Die freie Adressierbarkeit des Stacks erlaubt dem Compiler einige Optimierungen bei der Ausdruckauswertung, die bei Benutzung des von der Architektur vorgegebenen Berechnungsstacks nicht einsetzbar wären. Die MONADS-Architektur läßt nur Zugriffe auf das jeweils oberste Element auf dem Stack zu, gemeinsame Unterausdrücke können somit nur schwer eliminiert werden. GNU C kann Stacks, die keine direkte Adressierung zulassen, praktisch nicht einsetzen. Nur an wenigen Stellen könnte der Stack kurz, d.h. innerhalb eines „Befehls“ im Sinne der Maschinenbeschreibung, benutzt werden.

5.8 Startcode für C-Programme

Verschiedene Betriebssysteme haben jeweils andere Bedingungen, die beim Start von Programmen gelten. Es ist die Aufgabe des Startcodes, dies zu vereinheitlichen und dafür zu sorgen, daß die Funktion `main()` mit den notwendigen Parametern aufgerufen wird. Bei vielen Programmen, gerade bei denen aus der UNIX-Welt, wird angenommen, daß der Aufruf des Programms mit einer Kommandozeile erfolgt, die dem Programm übergeben wird.

Die Benutzung von Umgebungsvariablen, die in vielen Programmen benutzt werden, muß ermöglicht werden, auch wenn dies auf dem MONADS-PC nur schlecht in das vorhandene Betriebssystem paßt. Bei UNIX-basierten Systemen sind die Umgebungsvariablen ein Teil des Prozeßzustands, und die eindeutige Abbildung von Prozessen auf laufende Programme erlaubt die spezifische Weitergabe von einem Prozeß zum davon erzeugten Prozeß. Der MONADS-PC mit seinem im Betriebssystem verankerten In-Process-Modell muß daher dieses Verhalten simulieren.

Der Benutzer kann jedem Programm genau die relevante Umgebung mitgeben. Der Nachteil dieses Konzeptes ist die potentiell große Zahl verschiedener Sammlungen von Umgebungsvariablen. Auf dem MONADS-PC können Daten nur über Parameter des Schnittstellenaufrufs weitergegeben werden. Da die Angabe beliebig vieler Parameter nicht möglich ist, wurde auf die Kapselung der Umgebungsvariablen in einem Modul ausgewichen. Dieses Modul kann z.B. eine Textdatei sein. Alternativ könnte ein Modul des Kommandozeileninterpreters eingesetzt werden, genannt Variable Manager, das Variablen mit allen interaktiv einsetzbaren Typen verwalten kann.

Die Umgebungsvariablen können von einem Programm an ein anderes weitergegeben werden. Beim Aufruf eines C-Programms von einem anderen C-Programm müßte also eine neue Variable Manager-Instanz oder eine neue Textdatei angelegt werden, da das aufrufende Programm die Variablenwerte potentiell ändern kann. Die Erzeugung eines neuen Variable Managers ist mindestens ebenso aufwendig wie die Erzeugung einer neuen Textdatei. Der Einsatz des Variable Managers für die Implementierung von Umgebungsvariablen wurde verworfen, weil die MONADS-Konzepte strikt von den UNIX-Konzepten getrennt werden sollten.

Auch die Implementierung der standardmäßig vorhandenen Ein-, Ausgabe- und Fehlerausgabedateien muß durch explizit übergebene Textdateien simuliert werden. Die Hauptarbeit liegt bei der C-Bibliothek. Der Startcode muß nur dafür sorgen, daß die vom Benutzer angegebenen Module Capabilities aufbewahrt werden.

Aus praktischen Gründen werden die zu übergebenden Parameter in zwei Gruppen aufgeteilt (siehe Bild 5.8). Die erste Gruppe wird gleich beim Aufruf der `open`-Schnittstellenfunktion übergeben, die zweite erst beim Programmaufruf. Eine denkbare Erweiterung dieses Konzepts wäre die Unterstützung einer nachträglichen Änderung der bei `open` angegebenen Parameter, ohne das Modul schließen und neu öffnen zu müssen, z.B. mit `set_stdin` und ähnlichen Funktionen.

```
{#e-}
#ifndef _CPROGRAM_H
#define _CPROGRAM_H

{ Generic C program class definition supported by the GNU CC port.  }

type
  c_program = class

    procedure open({in} stdin:   modcap of basic_text {'stdin'};
                   {in} stdout:  modcap of basic_text {'stdout'};
                   {in} stderr:  modcap of basic_text {'stderr'};
                   {in} rootdir: modcap of directory  {'dir'};
                   {in} currdir: modcap of directory  {'dir'};
                   {in} env:     modcap of basic_text {'environ'});

    procedure close;

    function main({in} arg: string {''}): integer;

{ syntax

   COMMAND : /id(c_program)/' ' c_comm --> '#1.main "#1 #3"'
           | /id(c_program)/ --> '#1.main "#1"' .

   c_comm : chars c_rest --> '#1#2' .

   c_rest : '"'chars c_rest --> '\"#2#3'
          | .

   chars : /string_lib.scan_chars/.
}

  end; { c_program }

#endif
{#e+}
```

Abbildung 5.8: Schnittstellendefinition von C-Programmen

Die erste Gruppe enthält sechs Module Capabilities (Standardeingabe, Standardausgabe, Standardfehlerausgabe, das aktuelle Verzeichnis, das oberste für das Programm anzunehmende Verzeichnis des Verzeichnisbaums und die Datei mit den Umgebungsvariablen). Diese werden hauptsächlich von der C-Bibliothek benutzt und ändern sich selten. Weil sie für die meisten Programmaufrufe unverändert beibehalten werden können, kann das Programm mit denselben Module Capabilities mehrmals durch den Aufruf der Schnittstellenfunktion `main` gestartet werden.

Die Angabe des aktuellen und des obersten Verzeichnisses ist für die Erstellung der Dateizugriffsfunktionen der C-Bibliothek erforderlich. Das Betriebssystem des MONADS-PC beschränkt die Organisation von Verzeichnissen nicht auf Bäume, sondern läßt beliebige Graphen zu. Dies hat zur Folge, daß ein Startverzeichnis zur Auflösung von Dateinamen nicht automatisch ermittelbar ist. Die Angabe des obersten Verzeichnisses erlaubt es, absolute Verzeichnisnamen auch im MONADS-System zu unterstützen.

Jedes Programm kann auf diese Weise einen eigenen Namensraum erhalten. Der Benutzer muß nur die richtigen Verzeichnisse für das aktuelle und das oberste Verzeichnis angeben, um die zugreifbaren Dateien komplett zu trennen. Das erlaubt die vollständige Trennung verschiedener Programme, obwohl sie von einem Benutzer eingesetzt werden.

Der eigentliche Programmstart hat als einzigen Parameter die Kommandozeile, die als einfacher Stringparameter übergeben wird. Die Zerlegung der Kommandozeile in Worte erfolgt durch den Startcode anhand der auf nahezu jedem Betriebssystem dafür verwendeten Regeln: jedes Leer- oder Tabulatorzeichen ist ein Trenner zwischen den einzelnen Parametern, außer es steht in einem mit Anführungszeichen gekennzeichneten Teil oder es ist mit einem speziellen Zeichen (hier das \-Zeichen) gekennzeichnet. Durch diese Verteilung der Parameter und Aufgaben ist es leicht, ein Programm mehrere Male mit unterschiedlichen Kommandozeilen zu starten, ohne jedesmal die Module Capabilities wieder angeben zu müssen.

Die in Bild 5.8 angegebene Klassendefinition enthält auch noch andere Teile, die entweder für den MONADS-Pascal-Compiler relevant sind oder das Verhalten des MONADS-Kommandozeileninterpreters steuern. Für C-Programme wird dadurch eine Abkürzung definiert, die es erlaubt, nur durch Angabe des Modulnamens die Schnittstellenfunktion `main` aufzurufen.

Die Abkürzung erlaubt es, C-Programme auf dem MONADS-PC auszuführen, ohne die aufzurufende Schnittstellenfunktion `main` anzugeben. Auf ähnliche Weise werden alle angegebenen Kommandozeilenparameter in einen einzigen String zusammengefaßt, der an die Schnittstellenfunktion `main` übergeben wird. Bei Eingabe von `xyz Test 1` wird das Kommando

```
xyz.main "Test 1"
```

ausgeführt. Dies gleicht die Benutzung von C-Programmen der vielen Benutzern geläufigen UNIX-Konvention an.

Durch passende Definition der Ein- und Ausgabedateien lassen sich die bei UNIX-Kommandozeileninterpretern möglichen Umleitungen der `stdin`-, `stdout`- und `stderr`-Dateien nachbilden. Eventuell ist dies mit der Definition von Syntaxregeln für den Kommandozeileninterpreter möglich.

Der Startcode für den MONADS-PC muß folgende Arbeitsschritte durchführen:

- Der in C-Programmen übliche Speicheraufbau wird durch Anlegen von Segmenten für Daten und Stack hergestellt. Das Segment mit allen Konstanten wird zugreifbar gemacht.
- Die globalen Variablen werden mit den im Programm definierten Werten vorbesetzt und der Stackzeiger auf den Anfangswert gesetzt.
- Die übergebene Kommandozeile wird in einzelne Worte zerlegt.
- Die Umgebungsvariablen werden aus der angegebenen Datei eingelesen.
- Die Variablen für die Verwaltung des Heaps werden initialisiert.
- Die Initialisierungsfunktion der C-Bibliothek wird aufgerufen[7].
- Das C-Hauptprogramm `main()` wird aufgerufen.
- Das Programms wird durch Aufruf von `exit()` beendet.

[7]Die Initialisierungsfunktion der C-Bibliothek ruft zur passenden Zeit die C++-Konstruktoren auf.

Kapitel 6

Portierung des GNU C-Compilers

Die im Kapitel 5 beschriebenen Anforderungen und Eigenschaften müssen nun so aufbereitet werden, daß daraus ein neuer GNU C-Compiler samt zugehörigen Hilfsprogrammen erzeugt werden kann.

Im Rahmen dieser Arbeit wurden alle für einen funktionierenden C-Compiler benötigten Teile implementiert. Dazu gehört natürlich die Portierung des GNU C-Compilers mit den dazu erforderlichen Teilen der Maschinenbeschreibung. Für einen einsatzfähigen Compiler werden zusätzlich ein Assembler, Linker und die Programme zur Bibliotheksverwaltung und zur Generierung des Aufrufkonvertierungscodes benötigt. Alle diese Grundbestandteile müssen vorhanden sein, um den C-Compiler einsetzen zu können. Wenn diese Basis fertiggestellt ist, dann bildet die teilweise portierte C-Bibliothek den Abschluß des Gesamtsystems.

Die Konzepte zur Erstellung der Hilfsprogramme wurden bereits behandelt. Die Implementierung der Hilfsprogramme ist mit diesen Entwürfen leicht möglich. Im folgenden wird daher auf diese Programme nur noch dort eingegangen, wo spezielle Eigenschaften der Implementierung relevant sind.

Die Beschreibung einer GCC-Portierung besteht aus sechs Teilen, die unten mit den für die Portierung auf den MONADS-PC benutzten Dateinamen gekennzeichnet sind. Erst nach Erstellung der ersten drei kann überhaupt ein Compiler erzeugt werden. Meistens ist auch die vierte Datei nötig, um einen funktionsfähigen Compiler zu bekommen. Die letzten beiden Dateien sind derzeit noch nicht im Einsatz, weil der Compiler nicht auf dem MONADS-PC selbst lauffähig ist. Sie wurden implementiert, um die erwarteten Definitionen zu dokumentieren.

`mpc.h` Beschreibung der vom Prozessor vorgegebenen Maschineneigenschaften und der gewünschten Compilerdetails, z.B. das Stacklayout und die Registerbenutzung,

`mpc.md` Definition aller für den Compiler relevanten Maschinenbefehle, inklusive evtl. vorhandener Peephole-Optimierungen,

`mpc.c` Unterprogramme für die Ausgabe der Maschinenbefehle, wie z.B. die Ausgabe des Codes für den Funktionsanfang und das Funktionsende, die

Funktionen für die Ausgabe der Operanden der Maschinenbefehle und andere Funktionen, die spezifisch für den Prozessor sind, aber nicht in die `mpc.h`- oder `mpc.md`-Dateien passen,

`t-mpc` Makefile-Regeln, die spezifisch für die Portierung auf einen neuen Prozessor sind, z.B. ob Bibliotheksfunktionen für bestimmte Maschinenbefehle (wie Division) verwendet werden müssen,

`x-mpc` Makefile-Regeln, die bei einer Übersetzung auf dem MONADS-PC (d.h. wenn die Erzeugung des Compilers dort abläuft) benötigt werden,

`xm-mpc.h` Definition der auf dem MONADS-PC vorhandenen Umgebung, damit ein auf dem MONADS-PC lauffähiger Compiler auch auf anderen Plattformen erzeugt werden kann.

Der Großteil der dabei anfallenden Arbeit besteht darin, die Dateien anhand der im GNU C-Handbuch [10] angegebenen Anleitung zu erstellen. Die Aufgabe gleicht dem Ausfüllen eines Fragebogens. Viele Eigenschaften sind anhand [5, 7] leicht anzugeben. Einige Details wie die Kosten der Befehle mußten jedoch bei den MONADS-Entwicklern selbst nachgefragt werden.

6.1 Maschineneigenschaften

Die Datei `mpc.h` beschreibt alle möglichen Aspekte des Compilers, die von der Portierung beeinflußt werden. Die Beschreibung erfolgt in Form einer umfangreichen Sammlung von C-Makros. Bei der Erstellung wurde in der Reihenfolge der Beschreibungen im Handbuch vorgegangen. Diese Anordnung zieht leider an manchen Stellen logisch zusammengehörende Definitionen auseinander. Dem steht der Vorteil gegenüber, daß die Definitionen eine vorbestimmte Reihenfolge bekommen und so die Suche nach einer im Handbuch benachbarten Definition einfach wird.

Die ersten beiden Abschnitte definieren die Arbeitsweise des sogenannten Compiler Drivers `gcc` und der Optionsverarbeitung im C-Präprozessor `cpp` und im eigentlichen Compiler `cc1`. Hier werden die Optionen und ihre Werte definiert, die an die einzelnen Teile des Compilers weitergegeben werden. Die Koordinierung der Weitergabe von Optionen an die verschiedenen Teilprogramme des Compilers erlaubt ein einfaches Übersetzen von Programmen, die aus mehreren Übersetzungseinheiten bestehen. Generell können alle an den Compiler Driver übergebenen Optionen veranlassen, daß weitere Optionen an die aufgerufenen Teile übergeben werden. Ein Beispiel ist die Option `-p`, deren Bearbeitung in der Datei `mpc.h` explizit sichtbar ist. Wenn dem Compiler Driver die Option `-p` übergeben wurde, dann wird diese Option an den Compiler `cc1` weitergegeben. Die Option `-p` weist den Compiler an, zusätzlich für jede Funktion Code zur Untersuchung des Laufzeitverhaltens zu generieren. Dieser Code benötigt eine zusätzliche spezielle Bibliothek, die der Compiler Driver automatisch dem C-Compiler angibt.

Die folgenden Abschnitte in `mpc.h` legen die Anordnung der Daten im Speicher fest. Die Byteanordnung eines Wortes kann ebenso definiert werden wie die Wortbreite der Register. Der Compiler wird angewiesen, immer ganze Worte zur Übergabe von Funktionsparametern auf dem Stack zu verwenden. Auf diese Weise werden alle Datentypen in schnell verarbeitbare Portionen konvertiert.

Hier läßt sich die beim MONADS-PC wichtige Definition unterbringen, daß Wortzugriffe auf den Speicher nur mit durch 4 teilbaren Adressen zulässig sind. Auch Optimierungen des Speicheraufbaus, wie die bevorzugte Ausrichtung von Stringkonstanten auf Wortgrenzen, sind möglich.

Die Definition der Zahlendarstellung bei Gleitkommawerten und des jeweils benötigten Speicherplatzes erlaubt es, auch auf dem MONADS-PC ohne die Softwareemulation von IEEE-754 double precision auszukommen. Die Gleitkommabefehle des Prozessors arbeiten grundsätzlich mit einfacher Präzision. Der Compiler wird deshalb angewiesen, alle Gleitkommavariablen mit einfacher Präzision abzulegen, auch Variablen vom Typ `double` und `long double`. Wenn später einmal Berechnungen mit größerer Genauigkeit erforderlich werden, können entweder neue Maschinenbefehle definiert oder Funktionen zur Emulation benutzt werden. Der Compiler ruft für Befehle, die in der Maschinenbeschreibung nicht definiert sind, automatisch die zugehörigen Emulationsfunktionen auf. Die Emulationsfunktionen müssen separat erstellt werden. Für Maschinen, die keine Gleitkommaeinheit haben, ist eine Emulationsbibliothek verfügbar. Diese kann auch für Prozessoren benutzt werden, die nur single precision unterstützen.

Die Entscheidung, den Datentyp `short` generell als 32 Bit-Zahl darzustellen, beschleunigt und verkleinert den erzeugten Code deutlich. Nachteilig ist der höhere Platzverbrauch, verglichen mit der üblichen Realisierung als 16 Bit-Zahl. Nur noch wenige Programme setzen den Typ `short` häufig ein. Daher überwiegt der Geschwindigkeitsvorteil den Nachteil der Platzverschwendung. Die genaue Darstellung von `short` ist im ISO C-Standard nicht definiert.

Es existiert aber Code, der implizit annimmt, daß ein `short` 16 Bit breit ist. Solcher Code ist im Compiler selbst enthalten. Falls die Annahmen über die Breite des Typs `short` nicht erfüllt sind, wird der Code nicht übersetzt. Die Aufgabe dieses Codes ist die Durchführung von Gleitkommaberechnungen im Format des Zielsystems. Allgemein kann nicht angenommen werden, daß die Zahlendarstellungen des Zielsystems und des Übersetzungssystems identisch sind. Der Compiler benutzt daher für die Auswertung von konstanten Ausdrücken und ähnlichen Aufgaben eine Sammlung von Funktionen, die das Ergebnis in der Darstellung des Zielsystems berechnen. Wenn dieser Code nicht einsetzbar ist, weil `short` nicht 16 Bit breit ist, dann werden alle Berechnungen im Zahlenformat des Übersetzungssystems durchgeführt. Das bedeutet, daß ein auf dem MONADS-PC laufender Compiler nicht benutzt werden kann, um Code für Architekturen mit anderer Zahlendarstellung zu erzeugen. Dies ist keine große Einschränkung, denn derzeit ist es nicht möglich, den Compiler auf dem MONADS-PC laufen zu lassen. Darüberhinaus ist es unwahrscheinlich, daß auf dem MONADS-PC je ein Cross-Compiler auf eine Plattform mit anderer Zahlendarstellung eingesetzt wird.

Für den Compiler ist es wichtig zu wissen, welche Register zur Verfügung stehen und welche Eigenschaften sie haben. Die Register können in Klassen mit unterschiedlichen Eigenschaften eingeteilt werden. Diese Klassen werden später bei der Beschreibung der Maschinenbefehle verwendet, um die für die einzelnen Befehle erlaubten Register zu beschreiben. Der MONADS-Prozessor erfordert die Unterteilung der Register in drei Klassen. In `mpc.h` werden zwar mehr Klassen definiert (siehe auch Bild 6.1), dies dient aber nur zur Vereinfachung der Befehlsbeschreibung. Die unmittelbar notwendigen Klassen sind:

- Der Akkumulator `a`, das am universellsten verwendbare Register, abgesehen von der fehlenden Indizierungsmöglichkeit.

- Das Register `ax`, das für spezielle Aufgaben wie z.B. zur Aufnahme des Divisionsrests bei Divisionen dient. Es ist bei allen arithmetischen und logischen Operationen als Quellregister erlaubt, was es trotz der vorhandenen Einschränkungen zu einem vom Compiler gut benutzbaren Register macht.

- Die vier Indexregister `i0`, `i1`, `i2` und `i3`. Diese Register sind bei vielen Befehlen genauso wie der Akkumulator verwendbar. Im Gegensatz zum Akkumulator können diese Register zur Indizierung (d.h. berechneten Adressierung) benutzt werden. Sie werden deswegen vom Compiler bevorzugt für Zeigerwerte eingesetzt.

  Für den Compiler sind aber nicht alle Register frei verfügbar. Das Register `i0` wird als Stackzeiger benutzt und das Register `i3` als Rahmenzeiger. Bei Funktionen, die ohne den Rahmenzeiger auskommen, kann die Adressierung innerhalb des aktuellen Stackrahmens vom Stackzeiger aus erfolgen und `i3` als normales Register benutzt werden.

Das Register `i0` wurde durch die Überarbeitung bzw. Neuimplementierung des Mikrocodes verfügbar. Bisher wurde es vom Mikrocode für interne Zwecke verwendet und war deshalb im Registermodell nicht vorhanden.

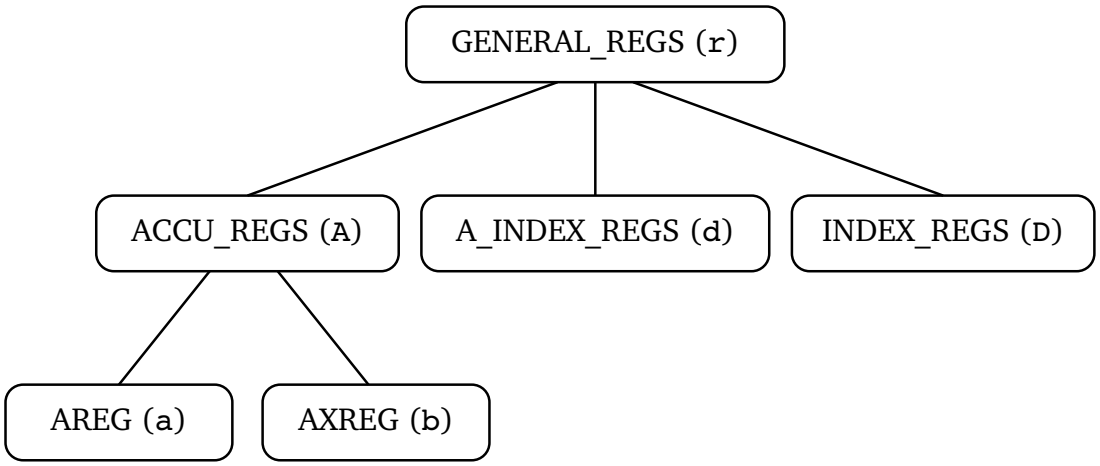

Abbildung 6.1: Registerklassen und Bezeichner für die Befehlsbeschreibung

Die Registergruppen AREG, AXREG und INDEX_REGS bilden den Kern der Registerklassen. Es wurden weitere Klassen gebildet, die nur eine Vereinigung zweier Registerklassen sind. Das vereinfacht die Maschinenbeschreibung und beschleunigt die Codegenerierung, da bei der Registervergabe weniger Alternativen betrachtet werden müssen. Die Klasse ACCU_REGS ist die Vereinigung von AREG und AXREG. Analog ist die Klasse A_INDEX_REGS die Vereinigung der Klassen AREG und INDEX_REGS.

Die sechs auf dem MONADS-PC zur Verfügung stehenden Register sind für eine effiziente Optimierung recht wenig. Es sind offenbar genügend, um in jeder Situation korrekten Code zu erzeugen. Der MONADS-PC ist bisher der Prozessor mit den wenigsten Registern von allen Prozessoren, auf die der GNU C-Compiler portiert wurde, ohne den Registersatz im Speicher zu simulieren.

Neben der statischen Bindung ist der größte Unterschied zur Diplomarbeit [11] der deutlich andere Speicheraufbau. Die Verwendung großer Segmente und die statische Bindung zur Übersetzungszeit erlaubt es, die im Prozessor vorhandenen Segmentregister während der gesamten Laufzeit eines Programms unverändert zu lassen. Da auch beim Unterprogrammaufruf die Segmentregister nicht geändert werden können[8], muß ein anderer Unterprogrammaufrufmechanismus verwendet werden. Der Stack wird vollständig selbst verwaltet.

Dieser Verzicht bietet auch eine Fülle von Vorteilen: der Stackaufbau kann ohne von der Architektur vorgegebene Zwänge festgelegt werden. Es ist möglich, Optimierungen zur Effizienzsteigerung des Unterprogrammaufrufs zu implementieren. Die Möglichkeit, beliebige Register als Funktionsparameter zuzulassen, wurde aber aufgrund der geringen Anzahl der Register nicht genutzt. Es wären maximal zwei Register überhaupt für Parameterübergabe benutzbar, denn zwei Register haben vorgegebene Funktionen (Stack- und Rahmenzeiger). Für Ladeoperationen sind teilweise bis zu zwei Temporärregister erforderlich.

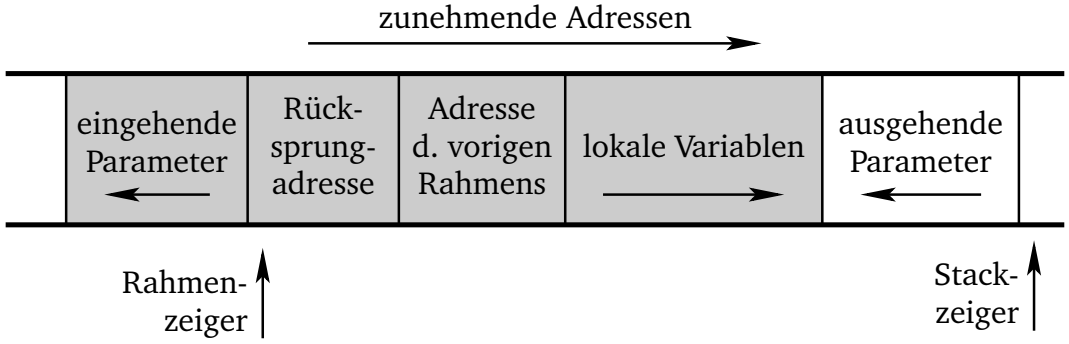

Abbildung 6.2: Aufbau eines Stackrahmens

Bild 6.2 zeigt den genauen Aufbau eines Stackrahmens. Der Bereich von den eingehenden Parametern bis zu den lokalen Variablen bildet den Stackrahmen der aktuellen Funktion. Der Bereich für ausgehende Parameter ist eine Optimierung

[8]Die anderen Stackrahmen wären sonst nicht mehr adressierbar.

des Stackaufbaus. Es wird gleich zum Beginn einer Funktion Platz für die längste benötigte Parameterliste für Funktionsaufrufe angelegt. Bei Funktionsaufrufen muß daher der Stackzeiger nicht mehr angepaßt werden. Es werden also potentiell mehrere Anpassungen am Stackzeiger (eine Anpassung pro Funktionsaufruf) durch eine einzige am Anfang der aufrufenden Funktion ersetzt. Die Kosten dieser Optimierung sind meist gering, denn beim Aufruf von Unterprogrammen ist nur etwas Speicherplatz auf dem Stack unbenutzt. Die Größe des temporär verlorenen Bereichs variiert mit der Länge des reservierten Bereichs und der Größe der Parameterliste des durchgeführten Aufrufs.

Für den Aufruf von Unterprogrammen ist die Verwendung dieser Optimierung der Parameterübergabe nicht direkt sichtbar, denn die Anordnung der Daten in der Parameterliste ist so gewählt, daß auch andere Implementierungen möglich sind. Der erste Parameter einer Funktion wird an der höchsten Adresse des reservierten Bereichs abgelegt, die weiteren Parameter liegen an niedrigeren Stackadressen. Die klassische Methode des Unterprogrammaufrufs (Platz für die Parameterliste reservieren, Parameter übergeben, Funktion aufrufen) kann durch die Entfernung eines Makros in der Maschinenbeschreibung erreicht werden.

Das einzige Register, das am Funktionsanfang und Funktionsende speziell behandelt werden muß, ist der Rahmenzeiger. Dieses kann als normales Register für Berechnungen dienen. Der alte Wert muß gespeichert werden, da sonst der Zustand einer aufrufenden Funktion zerstört wird. Der Rahmenzeiger kann nicht wie alle anderen Register vom Aufrufer gespeichert werden, denn dieser muß möglicherweise selbst relativ zum Rahmenzeiger auf die lokalen Variablen zugreifen. Nach dem Aufruf einer Funktion wäre es nicht mehr möglich, die Stelle wiederzufinden, an der der gewünschte Wert liegt. Der Wert des Rahmenzeigers im Register `i3` wäre unwiederbringlich zerstört.

Alle anderen Register, soweit sie nicht zum Aufruf von Funktionen bzw. zur Rückgabe von Funktionsergebnissen dienen, werden vom Aufrufer gespeichert. Dies führt einerseits dazu, daß GCC besseren Code erzeugt (dies wurde durch Versuche herausgefunden), andererseits ist gewährleistet, daß nur Register gespeichert werden, die sinnvolle Werte enthalten. Eine Alternative dazu wäre, am Anfang jeder Funktion die benutzten Register abzuspeichern und am Ende wiederherzustellen. Diese Möglichkeit ist aber weniger effizient, da wegen der Registerknappheit (auch in kleinen Funktionen) meist alle Register benutzt und damit zwischengespeichert werden. Die abgespeicherten Werte werden sehr oft nicht mehr gebraucht und die Speicherung ist damit verschwendete Zeit.

Der Unterprogrammaufruf ist wegen der wenigen zu bearbeitenden Daten sehr effizient. Es ist dabei von Vorteil, daß die derzeitige Sequenz nicht durch spezielle Maschinenbefehle, sondern durch mehrere normale Befehle implementiert ist. Änderungen sind so leicht möglich.

Einen großen Teil der Beschreibung der Maschineneigenschaften nimmt die Beschreibung der genauen Assemblersyntax ein. Dieser Teil ist ohne große Schwierigkeiten implementierbar, da sich die Assemblersprache des MONADS-PC nicht grundlegend von anderen Prozessoren unterscheidet.

6.2 Speicheradressierung

Um möglichst viele Vorteile der Segmentierung bei gleichzeitiger Erzeugung effizienten Codes für Speicherzugriffe zu erreichen, wurde beschlossen, Zeiger mit 32 Bit-Zahlen darzustellen, wobei die obersten Bits für die Adressierung verschiedener Segmente reserviert sind. Dies entspricht der Mischung zwischen Segmentierung und flacher Speicherorganisation. Die Verwendung dieser Adreßdarstellung hat gegenüber der Verwendung eines einzigen Segments, das alle Werte enthält, folgende Vorteile:

- Robustheit wird durch Aufteilung des Adreßraums in Abschnitte erreicht, die durch nicht adressierbare Bereiche getrennt sind. Auch Nullzeiger referenzieren nicht adressierbare Bereiche.
- Konstanten müssen nicht bei jedem Programmstart in das einzige adressierbare Segment kopiert werden und sind durch die Architektur vor Veränderung geschützt.
- Eine spätere Realisierung neuer Sprachen mit Unterstützung des Modulkonzepts bleibt möglich, da Instanzvariablen dann in einem eigenen Segment abgelegt werden können.
- Die Maschinenbefehle des MONADS-PC können maximal 16 Bit-Offsets in Segmente direkt im Befehl angeben. Größere Offsets müssen in ein Indexregister geladen werden. Reale Programme enthalten aber oft mehr als 64 KByte globale Daten und Konstanten.

Es gibt zwei Möglichkeiten, diese Adreßdarstellung zu implementieren. Eine offensichtliche Möglichkeit ist die Erzeugung einer Sequenz von Maschinenbefehlen, die aus der Adresse das anzusprechende Segment berechnen und den Zugriff passend durchführt. Diese Lösung wird zwangsläufig sehr langsam, da für jeden Speicherzugriff viele Maschinenbefehle ausgeführt werden müssen. Wegen der Beschränkung des Wertebereichs für unmittelbar in Maschinenbefehlen angegebene Offsets müßte die Realisierung eines effizienten Zugriffs auf größere Mengen globaler Variablen ähnlich wie bei vielen RISC-Prozessoren implementiert werden. Das würde Zugriffe auf globale Variablen weiter verlangsamen. Zugriffe auf den Heap wären hier nicht betroffen, da auf den Heap immer mit Zeigervariablen zugegriffen wird. Diese müssen ohnehin zuerst in Register geladen werden. Zugriffe auf den Stack sind bei kleinen Stackrahmen auch unkritisch, da immer relativ zum Stackzeiger oder Rahmenzeiger adressiert wird. Bei großen Stackrahmen muß aber zuerst die Adresse berechnet und dann zugegriffen werden, was ein zusätzliches Register erfordert.

Eine schnellere Lösung ist die Implementierung einer neuen Adressierungsart direkt in Mikrocode, was eine vergleichsweise effiziente Implementierung ergibt. Hier kann auch das Problem der beschränkten Offsetangabe der bestehenden Adressierungsarten beseitigt werden und so in vielen Fällen eines der ohnehin wenigen Register eingespart werden.

Die Adressierungsart wurde in Mikrocode realisiert, weil die Portierung des Compilers ohne spezielle Unterstützung ungleich aufwendiger und ineffizienter geworden wäre. Es darf aber nicht vergessen werden, daß die neue Adressierungsart langsamer als normale Segmentzugriffe ist, denn die Mikromaschine ist bei den intern benötigten Schiebebefehlen vergleichsweise langsam.

In Bild 6.3 ist die Implementierung der Adressierungsart genauer dargestellt. Der Zeiger wird in einen 4 Bit-Segmentindexteil und einen 28 Bit-Offsetteil aufgespalten. Diese Verteilung stellt keine Einschränkung des adressierbaren Bereichs dar, denn die maximale Segmentgröße des MONADS-PC beträgt 256 MByte. Diese Beschränkung stammt eigentlich von der Größenbeschränkung von Adreßräumen.

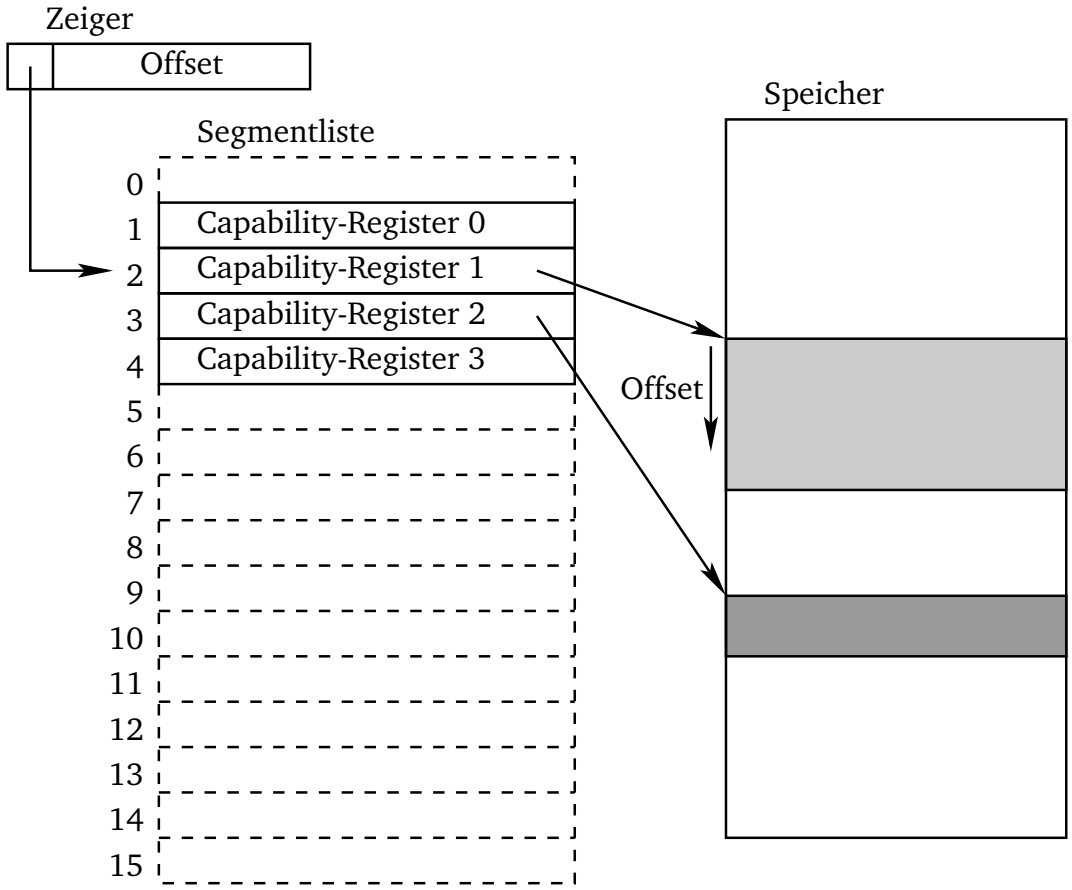

Abbildung 6.3: Adressierungsart für C-Programme

Die 4 Bit, die für den Segmentindexteil zur Verfügung stehen, werden nicht vollständig ausgenutzt, da bisher kein Bedarf dafür besteht und durch die Selbstbeschränkung auf wenige Segmentregister die verbleibenden ohne Probleme für andere Zwecke benutzt werden können. Es sind nur drei Segmente erforderlich, um alle Strukturen unterbringen zu können. Die Grenze wurde bei vier Segmentregistern gezogen. Das verbleibende Segmentregister kann später z.B. einen Verweis auf Instanzdaten enthalten.

Es ist vorgesehen, einen nicht adressierbaren Bereich am Anfang des Speichers zu haben, damit Zeiger mit diesen Werten niemals dereferenziert werden können. Für diesen Zweck wird der Eintrag `0` der Segmentliste reserviert. Die vier adressierbaren Segmente bekommen die daran anschließenden Werte `1`, `2`, `3` und `4`. Ihnen werden die Capability-Register 0 bis 3 zugeordnet. Alle weiteren möglichen Indexwerte sind ebenfalls ungültig. Diese Werte können z.B. zur Repräsentierung von Zeigern auf von C-Programmen nicht manipulierbare Datenstrukturen wie Module Capabilities, Module Call Segmente und Semaphoren benutzt werden.

Die neue Adressierungsart erlaubt darüberhinaus die direkte Angabe von kompletten 32 Bit-Adreßangaben in den Maschinenbefehlen, überwindet also die Einschränkungen des direkt adressierbaren Bereichs von Segmenten. Weil die Adressierungsart in jedem Fall ein eigenes Wort für den Offset benötigt, erhöht sich die durchschnittliche Befehlslänge gegenüber den Befehlen, die den 16 Bit-Offset im ersten Befehlswort unterbringen.

6.3 Beschreibung der Maschinenbefehle

Der folgende Abschnitt beschreibt einige für die Portierung des GNU C-Compilers relevante Details. Teile dieser Beschreibung greifen auf Eigenschaften des C-Compilers zurück, die im Handbuch des GNU C-Compilers [10] ausführlich erläutert werden. Die Kenntnis der speziellen Eigenschaften ist nur für einige wenige Teilaspekte erforderlich und wurde daher weggelassen.

Jeder für den Compiler relevante Maschinenbefehl muß in der Datei `mpc.md` beschrieben werden. Die Beschreibung erfolgt durch Spezifikation der durchgeführten Operation in einer formalen Schreibweise. Die Spezifikation der Operation wird auch Befehlsmuster genannt.

Befehlsmuster dienen zwei Aufgaben, der Erzeugung der entsprechenden RTL-Darstellung aus dem abstrakten Syntaxbaum des Frontend und der Optimierung der RTL. Die zwei Grundtypen der Befehlsmuster haben jeweils spezifische Eigenschaften, sie können aber beiden Aufgaben dienen:

Benannte Befehlsmuster sind für die Konstruktion der RTL-Repräsentation aus dem Syntaxbaum des Parsers vorgesehen. Der Compiler kennt diese Muster und benutzt sie zur Generierung der RTL als Ausgangsbasis für die nachfolgende Optimierung. Ein wichtiges Muster ist z.B. `movsi`, das 32 Bit-Werte zwischen Registern und dem Speicher austauscht. Die Semantik der benannten Befehlsmuster ist im GCC-Handbuch spezifiziert.

Unbenannte Befehlsmuster dienen zur anschließenden Optimierung der RTL-Darstellung. Wenn mehrere Befehle zu einem kombiniert werden, muß für den resultierenden Befehl ein Muster vorhanden sein, sonst wird die Optimierung nicht durchgeführt.

Es gibt darüberhinaus zwei verschiedene Typen der Definition von Befehlsmustern. Die Definition eines existierenden Befehls ist ein RTL-Ausdruck vom Typ `define_insn`. Die Definition einer Umsetzung in mehrere andere RTL-Muster geschieht mit `define_expand`.

Jedem Befehl ist eine textuelle Darstellung des RTL-Musters und des auszugebenden Assemblertexts zugeordnet. Für die Codegenerierung ist zunächst nur das RTL-Muster relevant. Der Assemblertext wird erst ganz am Ende der Übersetzung benutzt und wird ausschließlich vom RTL-Muster abgeleitet. Das ist besonders für mit `define_expand` definierte abgeleitete Befehlsmuster wichtig. Hier wird anhand des erzeugten RTL-Musters ein passender Befehl gesucht, der mit `define_insn` definiert ist.

Ein Beispiel für ein benanntes Befehlsmuster für einen auf der Maschine vorhandenen Befehl (die Berechnung der Quadratwurzel aus Gleitkommazahlen einfacher Genauigkeit) ist in Bild 6.4 dargestellt. Die Befehlsmuster sind selbst spezielle RTL-Ausdrücke. Die Muster für Maschinenbefehle des Prozessors bestehen aus vier Teilen:

1. der Name des Befehlsmusters (im Beispiel `"sqrtsf2"`), bei unbenannten Mustern ein Leerstring,

2. die RTL-Darstellung des Befehls in eckigen Klammern, mit Platzhaltern und Angaben zu den erlaubten Parameterausdrücken,

3. eine Bedingung, ob das Muster überhaupt anwendbar ist (hier `""`) und

4. ein Muster für die Ausgabe des Assemblerquelltexts. Die Ausgabe des Assemblerquelltexts ist in vielen Varianten möglich, bis hin zur Implementierung der Ausgabe in C. Im Beispiel wird durch das `@`-Symbol die automatische Auswahl der Alternativen in den folgenden Zeilen gewählt. Details hierzu sind in [10] nachzulesen.

Das Muster für die Quadratwurzelberechnung gibt an, daß es möglich ist, die Quadratwurzel aus dem Register `a` und allen Indexregistern (Registerklasse `D`) zu berechnen. Das Ergebnisregister und das Operandenregister müssen identisch sein. Die Angabe von `0` verweist auf den ersten Platzhalter, das Zielregister. Die genauen Bestandteile und der Aufbau der RTL-Muster sind im Compilerhandbuch ausführlich beschrieben.

```
(define_insn "sqrtsf2"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (sqrt:SF (match_operand:SF 1 "s_register_operand" "0,0")))]
  ""
  "@
  fpsqrt%0
  fpsqrti\\t%0")
```

Abbildung 6.4: Befehlsmuster für Quadratwurzelberechnung

Bei Befehlen, die einfacher durch eine Kombination von anderen Mustern ausgedrückt werden können, wird eine ähnliche Darstellung wie die von Maschinenbefehlen benutzt:

1. der Name des Befehlsmusters, bei unbenannten Mustern leer,
2. die RTL-Darstellung des Befehls in eckigen Klammern, mit Platzhaltern und Angaben zu den erlaubten Parameterausdrücken, potentiell eine Sequenz mehrerer RTL-Muster,
3. eine Bedingung, ob das Muster überhaupt anwendbar ist und
4. C-Code zur Ergänzung bzw. Generierung des RTL-Musters.

Die RTL-Muster, die von solchen Expander-Mustern erzeugt werden, müssen zu einem Muster passen, das mit `define_insn` deklariert ist. Sonst wäre es nicht möglich, Assemblerquelltext auszugeben. Ein Beispiel für einen solchen Befehl ist der Transfer von 16 Bit-Werten aus bzw. in den Speicher. Der MONADS-PC besitzt keinen Befehl, der diese Operation ausführt, deswegen werden mehrere Einzelbefehle erzeugt (siehe Bild 6.5).

```
(define_expand "movhi"
  [(set (match_operand:HI 0 "general_operand" "")
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "
{
  if (GET_CODE (operands[0]) == MEM) {
    operands[1] = force_reg (HImode, operands[1]);
  }

  if (GET_CODE (operands[0]) == MEM) {
    mpc_reload_outhi (operands[0], operands[1],
                      gen_reg_rtx (SImode));
    DONE;
  } else if (GET_CODE (operands[1]) == MEM) {
    mpc_reload_inhi (operands[0], operands[1],
                     gen_reg_rtx (SImode));
    DONE;
  }
}")
```

Abbildung 6.5: Beispiel für ein Expander-Befehlsmuster

Die Expander-Definition in Bild 6.5 benutzt für die Implementierung Funktionen, die in `mpc.c` definiert sind, nämlich `mpc_reload_outhi` und entsprechend `mpc_reload_inhi`. Diese legen die genaue Sequenz der RTL-Muster und somit die Maschinenbefehle fest. An den Stellen, die mit `DONE;` abgeschlossen werden, wird die im Befehlsmuster vorgegebene RTL verworfen, es werden also ausschließlich die explizit ausgegebenen Befehle in die RTL-Darstellung übernommen. In allen anderen Fällen wird das im Befehlsmuster angegebene RTL-Muster in die RTL aufgenommen. Im Beispiel ist dies der Fall, wenn nur Register als Operanden vorkommen.

Alternativ wäre es natürlich möglich, statt der Definition mit `define_expand` eine Definition mit `define_insn` anzugeben, die mehrere einzelne Assemblerbefehle ausgibt. Das macht diese Muster aber unnötig kompliziert und erfordert redundante Definitionen. Der Optimierer hat dann keine Gelegenheit, Teile des Befehls mit anderen zu kombinieren. Ein Beispiel für die Ausgabe mehrerer einzelner Befehle wurde zu Testzwecken in der Maschinenbeschreibung belassen und ist in Bild 6.6 zu sehen. Hier wird die Vorzeichenerweiterung einer 16 Bit-Zahl in eine 32 Bit-Zahl durch eine Kombination von Schiebebefehlen realisiert.

```
(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "=a,D")
        (sign_extend:SI
           (match_operand:HI 1 "register_operand" "0,0")))]
  ""
  "@
  ashft%0\\t#16\;ashft%0\\t#-16
  ashfti\\t%0,#16\;ashfti\\t%0,#-16")
```

Abbildung 6.6: Ausgabe mehrerer Befehle in einem `define_insn`-Muster

Es gibt noch exakt zwei weitere Definitionsarten, die in der Beschreibung der Maschinenbefehle verwendet werden können. Sie dienen zur Spezifikation von Peephole-Optimierungen und Aufspaltungsmöglichkeiten komplexer Befehle, die auf kein Befehlsmuster passen. Diese Definitionen wurden bei der Portierung bisher noch nicht eingesetzt, daher sei hier auf das GCC-Handbuch verwiesen.

6.4 Implementierung der `Makefile`-Fragmente

Die für eine Portierung zu erstellenden `Makefile`-Fragmente steuern, wie bestimmte Teile des Compilers übersetzt werden. Es existieren zwei Fragmente, da der zu erzeugende Compiler von zwei Hauptaspekten bestimmt wird, nämlich dem System, für das Code erzeugt wird, und dem System, auf dem der Compiler erzeugt wird. Die Fragmente werden je nach Konfiguration des Compilers in das `Makefile` zur Erzeugung des Compilers eingefügt.

Für die Beschreibung des Systems, auf dem der Compiler erzeugt werden soll, existieren nur wenige Einstellungsmöglichkeiten, da bei der Entwicklung des Compilers auf weitestgehende Unabhängigkeit geachtet wurde. Die wichtigste Einstellung ist die Angabe, welche Bibliotheken zusätzlich zur C-Bibliothek dazugebunden werden müssen, um einen lauffähigen Compiler zu erhalten. Darüberhinaus kann noch der Compiler für die Übersetzung von GNU C nicht bekannten Befehlen (z.B. die Division bei RISC-Prozessoren) angegeben werden. Dieser Compiler wird zur Erstellung einer Emulationsbibliothek der für GNU C unbekannten Befehle benutzt. Die für den MONADS-PC zu diesem Zweck erstellte `x-mpc`-Datei ist leer, da alle vordefinierten Einstellungen korrekt sind. Es werden keine zusätzlichen Bibliotheken benötigt, um einen lauffähigen Compiler zu bekommen – wenn die C-Bibliothek in ausreichendem Maß portiert ist.

Die Beschreibung der Übersetzungsparameter des Zielsystems in `t-mpc` sind umfangreicher. Es kann angegeben werden, mit welchen Optionen die verschiedenen Teile des Laufzeitsystems übersetzt werden müssen, bzw. ob sie überhaupt benötigt werden. Bei einigen Prozessoren ist z.B. keine Multiplikation oder Division vorhanden, dies muß dann durch Bibliotheken nachgebildet werden. Falls die Bibliotheken benötigt werden, wird der im anderen Fragment definierte C-Compiler zur Übersetzung benutzt. Im Fragment für den MONADS-PC wird der Startcode für C++-Programme (s.u.) ersetzt.

Falls GNU C je nach den angegebenen Optionen unterschiedlichen Code erzeugt, kann bei der Erzeugung arrangiert werden, daß mehrere Exemplare der Laufzeitumgebung generiert werden. Ein Beispiel sind Prozessoren, die eine unterschiedliche Anordnung der Bytes in einem Wort unterstützen. Der erzeugte Compiler wählt dann automatisch die zu den angegebenen Optionen passende Laufzeitbibliothek aus.

6.5 Startcode für C++-Programme

Zusätzlich zu den immer zu erstellenden Teilen wurde für den MONADS-PC auch der Startcode für C++ neu implementiert, weil die dem Compiler beiliegende generische Implementierung in C nicht genau dem entspricht, was auf dem MONADS-PC wünschenswert ist. Die generische Implementierung ist nicht falsch, nur ineffizient.

Die Implementierung ist für einen lauffähigen Compiler eigentlich nicht essentiell wichtig. Die Benutzer von GNU C gehen aber typischerweise davon aus, daß Programme problemlos funktionieren, die gemischt in C und C++ implementiert sind. Damit dies alles reibungslos klappt, muß auch bei C-Programmen die Liste der Konstruktoren von globalen Objekten vor dem Aufruf von `main()` abgearbeitet werden. Dies wird von der C-Bibliothek übernommen. Am Programmende müssen die entsprechenden Destruktoren aufgerufen werden.

Der hier zu implementierende Code ist sehr einfach und klein, deswegen wurde er direkt in Assembler (in den Dateien `crtbegin.s` und `crtend.s`) realisiert. Die Implementierung in Assembler verringert auch die Codegröße, da der bei der generischen C-Version unvermeidbare tote Code gespart wird.

Kapitel 7

Die C-Bibliothek

Die C-Bibliothek stellt eine große Zahl von Funktionen (etwa 2 000) und dazugehörende Typdeklarationen und Makros für alle möglichen Einsatzzwecke bereit. Die Schnittstelle bietet zahlreiche Funktionsgruppen an, die komfortables Programmieren ohne dauernde Neuimplementierung von Standardaufgaben erlauben.

Die C-Bibliothek bietet eine leistungsfähige Abstraktion vom eingesetzten Betriebssystem, dessen spezifische Eigenschaften in den Hintergrund treten. Es werden eine Vielzahl von Funktionen angeboten, angefangen bei Dateizugriffen über Sortierfunktionen, Konversion von Zahlen zwischen verschiedenen Zahlendarstellungen, dynamische Speicherverwaltung, Stringmanipulation, bis zu regulären Ausdrücken. Die vollständige Verlagerung jeglicher höherstehenden Funktionalität in eine externe Bibliothek ist eine Konsequenz der bei der Entwicklung der Sprache C getroffenen Entwurfsentscheidungen. Die Sprache sollte nur die unbedingt notwendigen Konstrukte enthalten, um den Compiler möglichst einfach zu halten.

Die notwendige Bibliothek war zunächst nicht standardisiert. Es bildete sich aber sehr bald ein de facto-Standard heraus, der sich stark an den in der C-Sprachdefinition [4] aufgeführten Beispielen orientierte. Später wurden dann von ANSI, IEEE und X/Open verschieden umfangreiche und verschieden portable Teile standardisiert, um die verbliebenen kleinen Unterschiede zu beseitigen. Heute haben sich diese Standards durchgesetzt. Es kommt auf modernen Systemen nur noch sehr selten vor, daß Funktionen inkompatibel sind.

Eine umfangreiche Suche hat ergeben, daß es nur zwei portable C-Bibliotheken gibt, die für dieses Projekt ohne Einschränkungen benutzbar sind. Die erste ist die GNU C Library, gedacht für Einzelrechner, z.B. Workstations oder PCs. Die zweite C-Bibliothek, genannt „NEWLIB“, wurde von Cygnus Support aus vielen Quellen zusammengestellt. Die NEWLIB ist vorwiegend für Embedded Systems mit ihren spezifischen Problemen gedacht. Sie stellt vergleichsweise geringe Anforderungen an den verfügbaren Hauptspeicher und die Prozessorgeschwindigkeit, bietet aber nicht den gesamten Funktionsumfang. Weitere frei verfügbare

C-Bibliotheken sind beispielsweise in den verschiedenen BSD-Varianten erhältlich, diese sind aber ausschließlich für UNIX-Derivate geeignet. Sie sind nur schwer auf andere Betriebssysteme portierbar. Kommerzielle C-Bibliotheken kamen nicht in Frage. Es ist fraglich, ob sie für Portierungen auf andere Prozessoren und Betriebssysteme überhaupt geeignet sind.

Die schließlich gewählte GNU C Library erwies sich als sehr geeignet für eine Portierung, da sie ähnlich dem GNU C-Compiler von Anfang an mit dem Ziel der Portabilität entwickelt wurde. Der Umfang der vorhandenen Funktionen deckt alle gebräuchlichen Funktionen einer C-Bibliothek ab. Mehrere UNIX-kompatible Betriebssysteme (darunter Linux) benutzen die GNU C Library oder daraus abgeleitete Bibliotheken und haben keinerlei Portabilitätsprobleme mit existierenden C-Programmen. Die derzeit aktuelle Version 2.0 der Bibliothek wurde zwar bisher nur auf wenige Prozessoren und Betriebssysteme portiert, die Version 1.0 dagegen auf eine große Zahl. Die sehr unterschiedliche Zahl der Portierungen der beiden Versionen liegt nicht in der verminderten Portabilität der Bibliothek, sondern am immensen Arbeitsaufwand, der für die Wartung der vielen Portierungen erforderlich ist. Angesichts der wenigen Freiwilligen wird der Einsatz auf die wichtigsten Portierungen konzentriert.

7.1 Entwurf ausgewählter Teile

Die GNU C-Bibliothek ist sehr portabel implementiert. Das wurde durch saubere Trennung von portablem Code und betriebssystem- und prozessorabhängigem Code erreicht. Durch die daraus resultierende Arbeitsersparnis wird es überhaupt erst möglich, mit einem vertretbaren Zeitaufwand eine funktionierende C-Bibliothek zu erhalten. Trotzdem verbleibt noch einiges zu tun, um die ganzen Abstraktionsstufen für den portablen Teil zu implementieren.

Es ist im Rahmen einer Diplomarbeit unmöglich, sowohl einen Compiler als auch eine vollständige C-Bibliothek zu portieren. Aus diesem Grund wurden nur die bei der Portierung der C-Bibliothek zu erwartenden Probleme analysiert und, wenn möglich, Lösungswege entwickelt. Eine Implementierung konnte wegen des Umfangs der zu erstellenden Teile meist nicht erfolgen. Die Implementierung ist daher hauptsächlich als Prototyp anzusehen, der zeigt, daß der gewählte Weg zum Ziel führt. Die Fertigstellung der Portierung bleibt späteren Arbeiten vorbehalten.

Die Anpassung der C-Bibliothek an die MONADS-Architektur verlangt die Verwendung von Module Capabilities und Module Call-Segmenten. Sie sind für den Aufruf anderer Module unbedingt notwendig. Die Sprache C kennt diese Datentypen nicht, sie müssen daher auf andere Weise realisiert werden. Die C-Bibliothek muß andere Module aufrufen können, beispielsweise um die Bildschirmausgabe durchführen zu können.

7.1.1 Verwaltung von Module Capabilities

Module Capabilities sind von der Architektur geschützte Datenstrukturen, die Informationen über die Zugriffsberechtigungen zu Modulen enthalten. Sie müssen auf dem MONADS-PC in Segmenten eines speziellen Typs abgelegt werden, um sie vor Manipulationen zu schützen. Ohne Module Capabilities (kurz MODCAPs) ist es unmöglich, andere Module anzusprechen. Es wäre also für C-Programme unmöglich, mit dem Rest des Systems Daten auszutauschen, z.B. könnte nicht einmal „Hello world“ ausgegeben werden. Dies ist natürlich nicht akzeptabel und bedarf einer Lösung.

Module Capabilities können schon wegen der Beschränkung der Zahl der erlaubten Segmente durch die Adressierungsart nicht direkt adressiert werden. Es werden deshalb Zeiger auf einen Bereich der möglichen Speicheradressen (aus offensichtlichen Gründen ein Bereich, der nicht zugreifbar ist) als Repräsentierung für jeweils eine Module Capability benutzt. Unzulässige Dereferenzierungen können so leicht von der Speicherverwaltungseinheit im Prozessor abgefangen werden. Die Repräsentierung von Module Capabilities ist trotzdem sehr einfach.

Als Repräsentierung einer Referenz auf eine Module Capability wird der in der Datei `include/mpc/mpc.h` definierte Datentyp `modcap` verwendet, der nur ein spezieller Zeigertyp (`void *`) ist. Die Module Capabilities selbst werden in einem Feld abgelegt, dessen Index dem Zeigerwert entspricht. Um sicherzustellen, daß Module Capabilities nicht mit normalen Zeigern verwechselt werden können, wird in den obersten 4 Bits eine Kennung eingetragen, der kein adressierbares Segment entspricht. So wird ein ungültiger Eintrag in der Segmentliste angesprochen. Zeiger auf Module Capabilities sind für C-Programme nicht direkt dereferenzierbar.

Es ist hier nicht relevant, daß die Identifikation von Module Capabilities in C-Programmen beliebig manipulierbar ist, denn innerhalb eines C-Programms ist ohnehin uneingeschränkter Zugriff auf alle Daten möglich.

Um alle auf dem MONADS-PC vorkommenden Fälle der Bearbeitung und Übergabe von Module Capabilities durchführen zu können, müssen folgende Operationen verfügbar sein:

- Reservieren eines Speicherbereichs für eine Module Capability,
- Freigeben des Speicherbereichs einer Module Capability,
- Ungültigmachen einer Module Capability,
- Kopieren einer Module Capability,
- Verändern der Rechte einer Module Capability und
- Abfrage diverser Informationen über eine Module Capability.

Diese Funktionen müssen für die Implementierung einer voll funktionsfähigen Portierung der C-Bibliothek verfügbar sein. Die Funktionen zur Verwaltung von

Module Capabilities kümmern sich um den genauen Aufbau der Tabelle. Sie kapseln die verwendete Datenstruktur.

Auf eine vollständige Implementierung dieser Funktionen wurde verzichtet. Damit die C-Bibliothek dennoch getestet werden kann, wurde im C-Startcode dafür gesorgt, daß die beim Öffnen des Programmoduls übergebenen Module Capabilities zugreifbar sind. So kann auch mit dieser eingeschränkten Implementierung auf Dateien zugegriffen werden.

Die Benutzung einer Module Capability zum Aufruf einer Schnittstellenfunktion eines Moduls wurde bisher nicht behandelt, denn diese Aufgabe wird ausschließlich durch Code durchgeführt, der vom Stub-Generator aus der Klassendefinition erzeugt wurde.

Der Zugriff auf eine Module Capability ist über den bei der Reservierung erhaltenen Index in die Liste aller Module Capabilities durchführbar. Der direkte Zugriff über Assemblercode ist die einzige Möglichkeit, denn Module Capabilities sind für C-Programme nicht adressierbar. Der Stub-Generator erzeugt deswegen eine kurze Sequenz von direkt angegebenen Maschinenbefehlen, um auf die passende Module Capability für den `create`- bzw. `open`-Aufruf zuzugreifen.

7.1.2 Verwaltung von Module Call-Segmenten

Die Unterstützung von Module Capabilities genügt noch nicht für den Aufruf anderer Module. Beim Öffnen eines Moduls wird eine Datenstruktur, genannt Module Call-Segment (MCS) ausgefüllt, um den Aufruf der einzelnen Schnittstellenfunktionen effizienter durchführen zu können.

Module Call-Segmente werden analog zu Module Capabilities verwaltet. Module Call-Segmente haben eine zusätzliche Eigenschaft, die ihre Verwaltung erleichtert: sie werden nur so lange benötigt, wie ein Modul geöffnet ist. Sie können also nach dem Schließen von Modulen wieder freigegeben werden.

Die Module Call-Segmente werden für C-Programme, d.h. die C-Bibliothek, mit einem speziellen Datentyp `mcallseg` dargestellt, der ein spezieller Zeigertyp (ebenfalls `void *`) ist. Der Wert entspricht der Nummer des Module Call-Segments in der Liste aller Module Call-Segmente, wobei die obersten 4 Bits ebenfalls eine Kennung enthalten. Die Kennungen von Module Capabilities und Module Call-Segmenten sollten sich unterscheiden, damit keine Verwechslungen auftreten.

Wenn ein freies Module Call-Segment benötigt wird, dann wird zuerst in der Freiliste nachgesehen, ob ein Module Call-Segment verfügbar ist. Wenn ja, dann wird es aus der Freiliste entfernt und dem Aufrufer zugeordnet, indem ihm die Nummer in der Liste mitgeteilt wird. Wenn kein Module Call-Segment in der Freiliste ist, dann wird ein neues angelegt. Das zugeordnete Module Call-Segment kann danach frei benutzt werden. Wenn es nicht mehr benötigt wird, kann es einfach wieder in die Freiliste eingetragen werden. Auf diese Weise ist es möglich,

ohne viel Zeitaufwand Module Call-Segmente zu verwalten und darüberhinaus Ressourcen zu sparen.

Die Verwaltung von Module Call-Segmenten erfolgt ausschließlich in vom Stub-Generator erzeugten Code, denn dies ist die einzige Stelle, an der Schnittstellenaufrufe durchgeführt werden. Es wurde auf die Bereitstellung von Funktionen eigens für die Verwaltung von Module Call-Segmenten verzichtet.

Auf sehr ähnliche Weise kann die Verwaltung und Benutzung anderer, von der Architektur geschützter Datentypen, wie Semaphor-Segmente o.ä. erfolgen. Die direkte Unterstützung auf Hochsprachenebene wäre eleganter als diese indirekte Implementierung über Bibliotheksfunktionen. Die direkte Unterstützung würde aber kaum Vorteile bringen. An der Sprache wären große Änderungen notwendig. Die Änderungen würden dem Konzept von C widersprechen, die Funktionalität soweit wie möglich in externe Bibliotheken zu verlagern.

7.1.3 Überlegungen zur Implementierung von C-Exceptions

Die C-Bibliothek enthält einige Funktionen, die es erlauben, Funktionen für die Bearbeitung von verschiedenen Ausnahmesituationen zu installieren. Ausnahmesituationen sind nicht nur auf Exceptions beschränkt, sondern umfassen auch Prozeßkommunikation und andere Dinge. Die Funktionen für die Behandlung gelten dann jeweils bis zur Entfernung bzw. Installation einer anderen Funktion. Wenn eine Ausnahmesituation auftritt, so wird die dafür angegebene Funktion wie ein normales Unterprogramm aufgerufen. In dieser Funktion kann entschieden werden, ob das Programm abgebrochen, an einer anderen Stelle fortgesetzt (mit `setjmp()` und `longjmp()`), oder der Fehler einfach ignoriert werden soll, indem zurück an die Fehlerstelle gesprungen wird.

Der MONADS-PC bietet einen komplexen Exception-Mechanismus bereits auf Maschinenebene an. Der Mechanismus ist in C nicht direkt benutzbar, denn er setzt speziell zur Behandlung von Exceptions gedachten Code in der von der Architektur vorgegebenen Funktionsstruktur voraus. Der Code muß entweder bei der Implementierung einer Funktion oder durch den speziellen Assemblerbefehl `setexh` angegeben werden. Dies wäre in dieser Form in C nur schwer erreichbar, denn das erfordert eine C-Spracherweiterung. Solche Änderungen sind für diese Arbeit nicht erwünscht, denn es geht um eine möglichst unveränderte Portierung von C.

MONADS-Exceptions können in der vom C-Laufzeitsystem bereitgestellten Umgebung nicht ohne weiteres benutzt werden: die Hardware-Implementierung von Exceptions basiert auf der von der Architektur definierten Stackverwaltung. Das von der Architektur vorgegebene Verfahren ist nicht geeignet, weil die Stackverwaltung in C-Programmen (siehe Abschnitt 5.7) unabhängig vom Stack der Architektur ist. Jedes C-Programm besteht aus Architektursicht nur aus einem einzigen Unterprogrammaufruf. Das bedeutet, daß alle Exceptions an einer Stelle behandelt werden müssen. Für einen Programmierer ist dieses Verhalten nicht

akzeptabel. Für die Implementierung einer benutzbaren Exception-Behandlung ist es aber besser als es zunächst aussieht, vor allem zur Implementierung der Funktionen der C-Bibliothek.

Die Hauptkonsequenz einer zentralen Stelle für die Fehlerbehandlung in einem C-Programm ist die problemlose Realisierbarkeit des Wiederaufsetzens an der Stelle, an der ein Fehler auftrat. Die Exception-Behandlung der Architektur sieht das gesamte C-Programm als ein riesiges Unterprogramm an. Deshalb wird das Modul nie verlassen, um die Exception zu behandeln. Da die Architektur von der Aufrufstruktur keine Kenntnis hat, bleibt der Aufrufstack erhalten. Es geht also keine Information durch abgebaute Stackrahmen verloren. Ein einfacher Sprung an die Stelle, an der unterbrochen wurde, ist neben der Wiederherstellung der Registerwerte alles, was für ein Wiederaufsetzen notwendig ist. Der in der C-Bibliothek vorgesehene Aufruf von Unterprogrammen zur Exception-Behandlung läßt sich einfach und ohne Änderungen am normalen Programmablauf implementieren. Es ist nur Fehlerbehandlungscode im Laufzeitsystem hinzuzufügen, der den Unterprogrammaufruf zur entsprechenden installierten Funktion durchführt. Der Programmierer von C-Programmen muß sich nicht mit anderen Mechanismen zur Exception-Behandlung beschäftigen.

Die gesamte Gruppe von Funktionen zur Exception-Behandlung um `signal()` ist auf diese Weise standardkonform implementierbar und die Details der Exceptions der Architektur können vollständig verborgen werden, ohne großen Aufwand treiben zu müssen.

Auf eine Implementierung wurde verzichtet, da diese Funktionen nur selten in portablem Code vorkommen. Für eine Prototypimplementierung sollte dies keine große Einschränkung sein.

7.1.4 Überlegungen zur Prozeßverwaltung

In der C-Bibliothek sind Aufrufe zur Prozeßverwaltung enthalten, die erlauben, Prozesse zu erzeugen, den Status abzufragen, Nachrichten auszutauschen und Prozesse wieder zu löschen. Das Hauptproblem tritt schon bei der Erzeugung von Prozessen auf. Das Betriebssystem des MONADS-PC kennt Prozesse und erlaubt die Erzeugung neuer Prozesse mit einfachen Mitteln. Die Semantik dieser Prozeßerzeugung entspricht aber nicht der in vorhandenen C-Programmen benötigten Semantik. Die Prozeßverwaltung der C-Bibliothek ist stark von den UNIX-Betriebssystemfunktionen beeinflußt. Viele C-Programme verwenden den `fork()`-Aufruf, um neue Prozesse zu erzeugen. Die vom UNIX-Kern vorgegebene und von vielen anderen Betriebssystemen kopierte Semantik dieses Aufrufs ist die Verdoppelung des derzeitigen Prozesses. Der einzige wesentliche Unterschied im Zustand der beiden Prozesse ist der Rückgabewert des `fork()`-Aufrufs. Aber genau diese Beibehaltung aller Speicherinhalte bietet der Aufruf zur Prozeßerzeugung auf dem MONADS-PC nicht. Die naheliegende Idee, einfach nachträglich die Daten zu kopieren, ist nicht sinnvoll realisierbar.

Dies liegt am gewählten Speicheraufbau. Die Segmente für Daten/Heap und Stack müssen sehr groß sein, damit ihre Größe keine Grenze für die benutzten C-Programme darstellt. Die Speicherverwaltung der Architektur stellt nur den wirklich benutzen Speicher bereit. So ist diese Realisierung ohne Ressourcenverschwendung möglich. Das größte Problem bei diesem Aufbau ist die Erstellung einer Kopie des aktuellen Prozesses. Man kann die großen Segmente nicht einfach kopieren, denn dies würde auf den gesamten reservierten Speicher zugreifen und die effiziente Speicherverwaltung zunichte machen. Außerdem würde eine vollständige Kopie des Prozesses angelegt. Das verschwendet weiteren Speicher, da der Betriebssystemkern keinen Mechanismus für das Kopieren nur der geänderten Seiten besitzt. Dieser „copy on write“-Mechanismus wird in UNIX-Systemen eingesetzt, um möglichst wenige Seiten kopieren zu müssen. Das Verfahren verbilligt den `fork()`-Aufruf erheblich.

Auf dem MONADS-PC muß also eine reale Kopie erstellt werden, die sehr teuer ist. Der Zeit- und Speicheraufwand ist in den meisten Fällen Verschwendung, da oft unmittelbar nach dem `fork()`-Aufruf ein neues Programm mit `exec()` aufgerufen wird. Der `exec()`-Aufruf verwirft alle Daten, die somit unnötig kopiert wurden.

Bei `fork()` müssen zusätzlich alle offenen Dateien (d.h. Module) auch dem neuen Prozeß zur Verfügung gestellt werden, strenggenommen im selben Zustand wie die Dateien des Vaterprozesses. Dies ist auf der MONADS-Architektur nicht ohne weiteres möglich, denn Module Call-Segmente sind nicht frei kopierbar. Außerdem würde eine Kopie die erforderliche Semantik der Kopie des gesamten Dateizustands nicht erfüllen. Es müßte eine Kopie der `retained`-Variablen aller offenen Module angelegt werden, was den Aufwand weiter steigert. Bei Typemanagern, die in den `open`-Schnittstellenaufrufen Einträge in die `instance`-Daten vornehmen, ist nicht mehr klar, welche Änderungen ein `fork()`-Aufruf durchführen müßte, um die Konsistenz zu wahren und trotzdem den Zustand des offenen Moduls zu kopieren.

Ohne Unterstützung durch den Betriebssystemkern ist es also kaum sinnvoll, `fork()` zu implementieren. Die dazu notwendigen Änderungen sind aber sehr umfangreich, denn ein `fork()`-Aufruf sollte möglichst billig sein. Die Prozeßkopie wird in den meisten Fällen nicht benötigt, sondern dient nur dazu, ein neues Programm aufzurufen. Dieses schwer zu lösende Problem ist das Haupthindernis für eine Ausführung des GNU C-Compiler auf dem MONADS-PC selbst, da der Compiler Driver `fork()` und `exec()` verwendet. Eine Änderung der Implementierung des Compiler Drivers ist nicht einfach, denn es werden viele Details der Prozeßverwaltung benutzt.

Viel einfacher ist hingegen die Implementierung von `system()`. Diese Funktion startet ein neues Programm und wartet auf dessen Beendigung, ruft also quasi ein Programm als Unterprogramm auf. Die `system()`-Funktion könnte auf dem MONADS-PC mit leichten, für normale Benutzung nicht relevanten Abstrichen implementiert werden. Bei einem In-Process-Betriebssystem kann ganz auf die Erzeugung eines neuen Prozesses verzichtet werden. Die Operation kann

sehr einfach auf einen Modulaufruf des gewünschten C-Programms abgebildet werden. Allgemeine Module sind nicht aufrufbar, da `system()` nur die Übergabe einer Kommandozeile vorsieht. Hier erlaubt die andere Konzeption des MONADS-Betriebssystems eine viel einfachere Implementierung als üblich. Unter UNIX muß dies mit `fork()` und `exec()` nachgebildet werden, denn nur über `exec()` ist die Ausführung von Code eines anderen Programms möglich. Da aber `exec()` den aktuellen Prozesses komplett ersetzt, muß vorher ein neuer Prozeß erzeugt werden.

7.2 Beispielimplementierung

Für den Test des Compilers ist es erforderlich, wenigstens einige grundlegende Funktionen der C-Bibliothek zu portieren. Für erste Versuche wurde im Rahmen der Arbeit eine einfache Unterstützung für Bildschirmeingabe und Bildschirmausgabe implementiert. Dies ist für den Test komplexerer Programme unbedingt notwendig, denn die meisten sinnvollen Programme benutzen diese Funktionen auf die eine oder andere Weise. Bildschirmausgabefunktionen erleichtern die Fehlersuche und sind, da die Gesamtimplementierung sehr komplex ist, auch selbst ein guter Test für die Korrektheit des Compilers. Die Funktion `printf` besitzt äußerst aufwendige Funktionen zur Ausgabe von Gleitkommawerten. Die Funktion `printf` ist portabel implementiert, nur die Funktionen, die einzelne Zeichen ein- und ausgeben, müssen implementiert werden.

Die Implementierung der Bildschirmeingabe und Bildschirmausgabe ist als Beispiel gedacht, um den Aufruf externer Module zu illustrieren. Es wird so aufzeigt, auf welche Weise eine komplette Implementierung erreicht werden kann. Die Verwaltung von Module Capabilities ist nur zum kleinsten Teil implementiert. Nur die unbedingt notwendigen Aufgaben werden direkt vom Startcode des C-Programms durchgeführt. Die sechs beim `open`-Aufruf übergebenen Module Capabilities werden in die unten beschriebene Datenstruktur eingetragen. Die für Modulaufrufe notwendige Verwaltung von Module Capabilities und Module Call-Segmenten ist weiter unten in diesem Abschnitt beschrieben.

Die Gleitkommaeingabe und Gleitkommaausgabe der C-Bibliothek bietet Funktionen zur Wandlung von Gleitkommawerten zwischen der IEEE-754-Darstellung und Textstrings an. Die C-Bibliothek unterstützt drei verschieden genaue Datenformate (IEEE-754 single und double precision, sowie IEEE-854 extended precision). Der MONADS-PC bietet nur ein Format an. Aus diesem Grund wurden zwei der drei Datenformate entfernt, um Platz zu sparen. Diese Reduktion wirkt sich deutlich auf die Codegröße von Programmen aus, da `printf` Aufrufe für die Umwandlung von Gleitkommazahlen in Text für alle unterstützten Formate enthält. Die Funktionalität der C-Bibliothek wird dabei nicht beeinträchtigt.

Für die Verwaltung des Heaps benötigt die C-Bibliothek einige Funktionen, die auf herkömmlichen Systemen Speicherbereiche vom Betriebssystem anfordern. Dies ist auf dem MONADS-PC nicht nötig, da schon im Startcode ein großer

Speicherbereich für den Heap reserviert wird. Die Funktionen zur Anforderung von Speicher vom Betriebssystem aktualisieren deswegen nur einige Variablen zu statistischen Zwecken, weitere Operationen sind nicht notwendig. Hier könnte die Aktualisierung der Segmentgrenzen des Heaps erfolgen, wenn eingeschränkte Segmente verwendet werden. Die höheren Heap-Verwaltungsfunktionen der C-Bibliothek (`malloc()` usw.) verwalten Teile dieses angeforderten Bereichs. Dazu gehören auch Bereiche, die mit `free()` wieder freigegeben wurden und wiederbenutzt werden können.

Die Beispielimplementierung definiert auch einige Datenstrukturen, die erforderlich sind, um die C-Bibliothek überhaupt übersetzen zu können. Diese müssen für eine vollständig funktionsfähige Portierung noch einmal überarbeitet werden.

Die Identifizierung einer Module Capability ist durch den Wert ihrer Zeigerrepräsentierung festgelegt. Es fehlt noch eine Abbildung des Verweises auf die eigentliche Module Capability. Dies wird über eine Umsetzungstabelle (siehe Bild 7.1) geregelt. Der Wert des Zeigers wird als Index benutzt. Die Tabelle ist nicht über die in C-Programmen verwendete Adressierungsart ansprechbar und muß daher über Assemblercode zugegriffen werden.

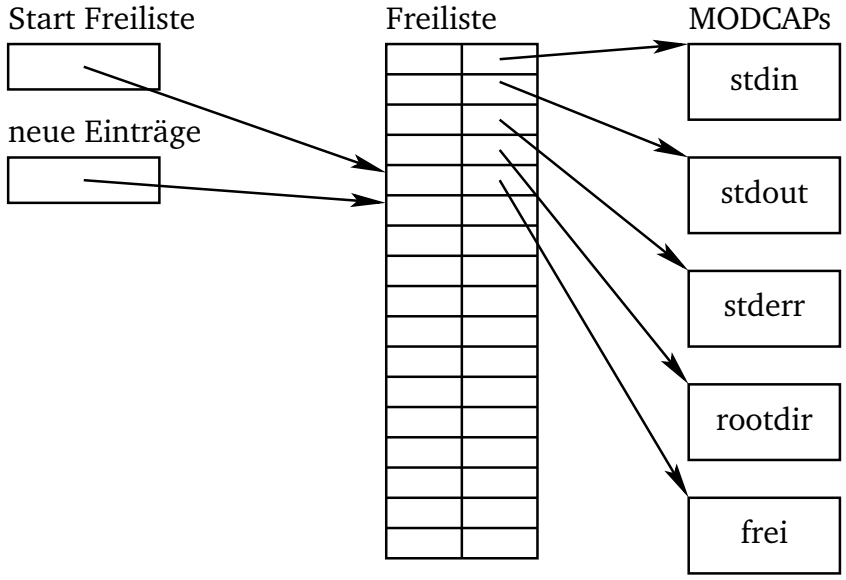

Abbildung 7.1: Verwaltung von Module Capabilities in C-Programmen

Da der MONADS-PC keinen Speicher explizit freigeben kann und noch keine Garbage Collection implementiert ist, werden momentan unbenutzte Module Capabilities in einer Freiliste aufbewahrt. So werden die Segmente automatisch nur dann belegt, wenn sie wirklich benötigt werden.

Für die Verwaltung der Module Capabilities wird ein einfacher Reservierungsalgorithmus mit Freilistenverwaltung verwendet, der alle benötigten Aufgaben schnell durchführen kann. Es wird ein Zeiger auf den Anfang der Freiliste und ein Zeiger auf den ersten unbenutzten Eintrag der Liste geführt. Das reicht aus,

denn vergebene Einträge sind durch die dem Aufrufer übergebene Identifikation lokalisierbar.

Analog zur Verwaltung von Module Capabilities wird die Verwaltung von Module Call-Segmenten implementiert. Derselbe Reservierungsalgorithmus wird in vom Stub-Generator erzeugten Code verwendet. Dies stellt sicher, daß nur wirklich benötigte Module Call-Segmente angelegt werden.

Die Situation in Bild 7.2 kann z.B. durch Reservierung von fünf Module-Call-Segmenten und anschließendem Freigeben des fünften, dritten, zweiten und ersten Module Call-Segments (in dieser Reihenfolge) erreicht werden.

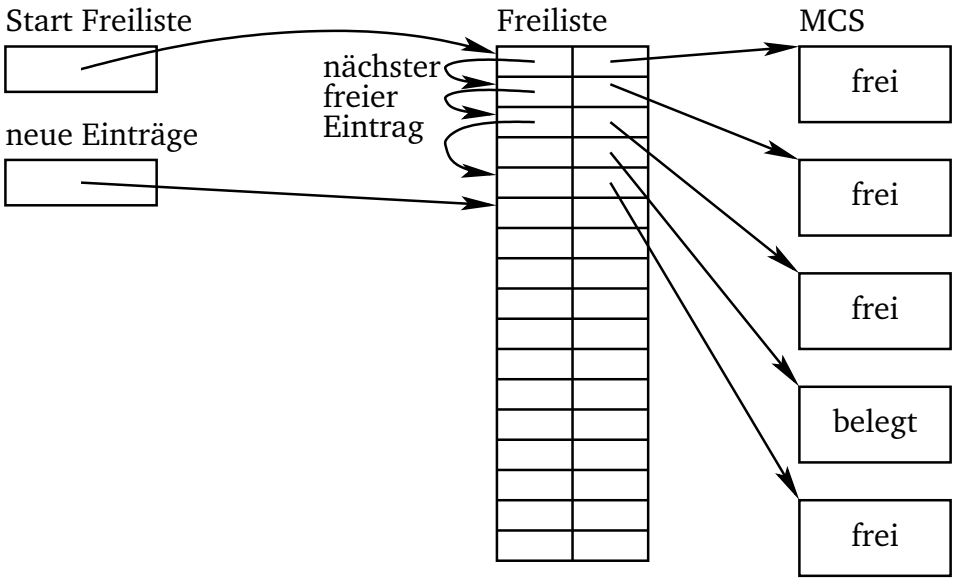

Abbildung 7.2: Verwaltung von Module Call-Segmenten

Kapitel 8

Test und Auswertung

Die Sicherstellung der Korrektheit von Compilern und anderen Entwicklungswerkzeugen ist eine der wichtigsten, aber auch schwierigsten Aufgaben ihrer Wartung. Die Entwicklungswerkzeuge werden für die Übersetzung aller anderen Programme benutzt. Fehler in den Werkzeugen betreffen also immer viele Programme. Die Fehler der Werkzeuge müssen so weit wie möglich verringert werden, um die von der Fehlerfortpflanzung verursachten Probleme in einem tolerablen Rahmen zu halten.

Beim Test von Programmen ist die Korrektheit der Werkzeuge eine oft getroffene Annahme. Ohne diese Annahme wird es schwer, auch nur die kleinsten Änderungen an den übersetzten Programmen ausreichend zu testen. Beim Test von Werkzeugen ist dagegen die Prämisse der Korrektheit der Werkzeuge nicht hilfreich. Der Test der Werkzeuge ist daher eine komplexe und aufwendige Aufgabe.

Das Problem des Tests von Compilern und Bibliotheken ist die sehr große Zahl der Fehlermöglichkeiten. Compiler sind sehr komplexe Softwaresysteme und C-Bibliotheken enthalten eine Vielzahl voneinander mehr oder weniger abhängiger Funktionen. Oft wird gesagt, daß Compiler die größten Programme sind, die genau einem Zweck dienen. Die Korrektheit läßt sich bei solchen Codemengen nur noch statistisch durch Programmtests ermitteln. Das bedeutet, daß nur Testfälle ausgewertet werden können. Die Güte der Testfälle bestimmt die Wahrscheinlichkeit, mit der Fehler in den Entwicklungswerkzeugen gefunden werden.

8.1 C-Compiler

Die Anforderungen an einen Compiler sind besonders hoch: er soll die Quellprogramme möglichst schnell in korrekten und schnellen Maschinencode übersetzen. Die Benutzer von Entwicklungssystemen sind meist noch zu bestimmten Abstrichen bei der Geschwindigkeit sowohl des Compilers als auch des erzeugten Codes bereit, die Korrektheit des erzeugten Codes ist dagegen äußerst wichtig. Compilerfehler sind meist schwer genau zu lokalisieren, wenn die Fehler sich

nicht direkt im Absturz des Compilers zeigen. Sie führen oft zu langer Fehlersuche im Programm des Benutzers, obwohl dort keine Fehler sind.

Compiler basieren zu großen Teilen auf formalen Spezifikationen, z.B. die Grammatik der zu übersetzenden Programmiersprache, die Codegenerierung, die einzelnen Optimierungsschritte und die Beschreibung des Zielrechners. Ideal wäre es, wenn anhand dieser Spezifikationen die Korrektheit automatisch verifiziert werden könnte. Dies scheitert bisher an der Gesamtkomplexität der Compiler, die heute verfügbare Verifikationsmethoden überfordert. Compiler werden den bekannten Testmethoden unterzogen, die nur die Präsenz von Fehlern nachweisen, aber nicht ihre Abwesenheit. Ist der Test normaler Programme schon eine komplexe Aufgabe, so ist der Test eines optimierenden Compilers eine fast unlösbare Aufgabe. Es müssen möglichst viele Ausführungspfade erreicht werden, damit der Test aussagekräftig ist. Bei Programmiersprachen mit ihren vom Benutzer geschriebenen Eingabedateien ist dies schon im Parser nur schwer zu erreichen. Ein großes Problem sind auch die verschiedenen Optimierungsschritte, die Abhängigkeiten zwischen den Anweisungen der Eingabedatei benutzen bzw. erzeugen. Hier ist es teilweise sehr schwer, eine Eingabe zu konstruieren, die bestimmte Fälle abprüft. Die Optimierungen beeinflussen sich oft gegenseitig.

Bei Compilern bietet es sich an, die Ergebnisse der Übersetzung von Testprogrammen mit der alten und neuen Version zu vergleichen. Falls sich die Ergebnisse unterscheiden, muß untersucht werden, ob die neue Version korrekt ist. Dieses Verfahren wird Regressionstest genannt. Regressionstests werden beim Test von GNU C meist nicht direkt in dieser ursprünglichen Funktion eingesetzt.

Der für GNU C am häufigsten ausgeführte Test ist die wiederholte Übersetzung des Compilers, wobei die erste Übersetzung mit einem vorhandenen Compiler durchgeführt wird, z.B. mit einer älteren Version von GNU C oder einem gekauften Compiler. Nach dieser ersten Übersetzung wird der dabei erzeugte Compiler dazu benutzt, sich selbst zu übersetzen. Der daraus resultierende Compiler wird mit dem Ergebnis einer weiteren Selbstübersetzung verglichen, die wiederum mit der im vorigen Durchgang erzeugten Version durchgeführt wird. Sind die beiden Versionen gleich, so ist es sehr wahrscheinlich, daß der Compiler keine offensichtlichen Fehler enthält. Die bei der Selbstübersetzung auftretenden Ausführungspfade decken wegen der Komplexität und Größe des Compilerquelltexts schon recht viele Fälle ab. In Compilern werden aber bestimmte Operationen kaum benutzt, z.B. Gleitkommaberechnungen.

Wenn ein Cross-Compiler erzeugt werden soll, dann kann dieses Verfahren nicht verwendet werden. Cross-Compiler erzeugen Programme, die auf einem anderen Zielsystem ablaufen. Die wiederholte Selbstübersetzung müßte teilweise auf dem Zielsystem durchgeführt werden. Beim derzeitigen Stand der Portierung der C-Bibliothek ist es aber nicht möglich, den GNU C-Compiler auf dem MONADS-PC auszuführen.

Weitere Tests liefern die bei verschiedenen Compilerversionen fehlgeschlagenen Übersetzungen von Programmen. Die aufgetretenen Probleme werden analysiert

und in verschiedene Gruppen eingeteilt. Beispielsweise gibt es Programme, die nicht übersetzbar sind, weil der Compiler abstürzt. Bei anderen ist der generierte Code falsch. Manchmal werden syntaktisch falsche Programme akzeptiert und übersetzt. Das weist auf Fehler im Parser hin.

Für GNU C gibt es eine umfangreiche Sammlung von Testprogrammen (der sogenannte C-Torture-Test), die einige hundert kleine und größere Testfälle in diesen Kategorien enthalten. Es gibt Testfälle, die nur übersetzt werden, um damit in den im Compiler enthaltenen Konsistenzprüfungen Fehler zu provozieren. Auch Abstürze des Compilers können bei der Übersetzung erkannt werden. Einen weiteren Satz Testfälle stellen syntaktisch falsche Programme dar, um sicherzustellen, daß ungültige Programme nicht akzeptiert werden. Die wichtigste Gruppe ist diejenige, die übersetzt und ausgeführt wird. Die Testfälle überprüfen hierbei selbst ihre Ergebnisse und melden erkannte Unterschiede zu den erwarteten Ergebnissen. Die Testfälle werden automatisch durchgegangen und ein Protokoll angefertigt. Bei Cross-Compilern ist die automatische Ausführung der Tests auf dem Zielrechner nicht möglich. Daher können die Ausführungstests nur übersetzt werden. Die Übertragung auf den MONADS-PC, der Start und die Auswertung der Ergebnisse erfolgen einzeln „von Hand“.

Von einigen GNU C-Entwicklern wird bereits die von Cygnus Support entwickelte DejaGnu-Testumgebung eingesetzt. DejaGnu ist nicht auf Tests von Compilern beschränkt, sondern stellt eine allgemeine automatisierte Testumgebung für Programme dar. Besonders die C++-Entwickler benutzen dieses Testsystem wegen der großen Zahl der C++-Testfälle, die mitgeliefert werden. Diese Testumgebung wurde nicht eingesetzt, denn die Tests für den C-Compiler wurden vom C-Torture-Test übernommen und bieten daher keine zusätzliche Information. Es wäre noch genauer zu untersuchen, ob die Verwendung von DejaGnu die automatische Durchführung und Auswertung der Ausführungstests ermöglicht. Dies wurde aus Zeitgründen nicht genauer untersucht, denn die Einarbeitung in die fortgeschrittenen Funktionen von DejaGnu hätte viel Zeit benötigt.

8.1.1 Erste Tests

In den ersten Monaten war es nicht notwendig, gezielt nach Fehlern zu suchen, denn die Übersetzung kleiner Testprogramme deckte genügend Fehler auf. Anfangs mußte der vom Compiler erzeugte Assemblercode durch Inspektion auf Fehler untersucht werden, weil noch keine C-Bibliothek vorhanden war. Ohne C-Bibliothek ist es in vielen Fällen nicht möglich, auch nur sehr einfache Testprogramme zu übersetzen, weil in C praktisch jede höherstehende Funktionalität in die Bibliothek verlagert wurde.

Selbst so einfache Programme wie in Bild 8.1 stellen erhebliche Anforderungen an das Entwicklungssystem. Der Compiler ist dabei eigentlich das kleinste Problem, wenn man davon absieht, daß alle Definitionen zum Prozeduraufrufme-

chanismus und der Parameterübergabe korrekt sein müssen. Die Hauptschwierigkeit liegt beim Startcode und der Bibliothek.

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Abbildung 8.1: Einfaches C-Programm

Die Aufgabe des Startcodes ist auf dem MONADS-PC umfangreicher als üblich. Der übergebene Kommandostring muß in die einzelnen Parameterworte zerlegt werden und das im Konzept des MONADS-PC nicht in dieser Form vorhandene Konzept des Environments muß über Textdateien simuliert werden.

Die Entwicklung des Startcodes wurde teilweise mit Hilfe des Compilers durchgeführt. Es wurde eine Version in C programmiert, die zwar nicht die ganze benötigte Funktionalität hatte, aber die wesentliche Struktur vorgab. Der nach der Übersetzung vorliegende Assemblerquelltext wurde als Ausgangsbasis des Startcodes benutzt. Dies war der erste Test des Compilers mit komplexeren Datenstrukturen. Es wurden mehrere Felder und Zeigerkonstrukte verwendet, was fast problemlos funktionierte. Dabei wurde entdeckt, daß in mehreren Befehlsmustern Tippfehler waren, die zu syntaktisch falschem Assemblertext führten. Diese kleinen Probleme waren leicht zu beseitigen.

Nachdem der Startcode fertig implementiert war, konnten erste Tests mit kompletten Programmen durchgeführt werden. Das Programm in Bild 8.2 wurde am häufigsten eingesetzt. Ohne eine teilweise funktionierende C-Bibliothek konnten noch keine Tests durchgeführt werden, die neben der übergebenen Kommandozeile und dem Rückgabewert von `main()` noch weitere Eingaben oder Ausgaben erforderten. Der Test des Kommandozeilenparsers benötigte keine solchen Funktionen, denn die Inhalte der Variablen wurden interaktiv direkt im Speicher überprüft, was mit einem speziell für den Test von Systemsoftware ausgelegten Mikrocode möglich war. Der Mikrocode erlaubte beispielsweise, die Ausführung des Programms an bestimmten Punkten anzuhalten und die Registerinhalte zu untersuchen. Diese Ergänzung der Debuggingmöglichkeiten war auch nötig, um die wenigen noch verbliebenen Fehler im Mikrocode aufzuspüren, die von den Umstrukturierungen des Befehlssatzes und der Implementierung einer neuen Adressierungsart resultierten.

Bei den ersten Tests wurden einige Fehler im Parser der Kommandozeile gefunden, die zu Programmabbrüchen wegen unerlaubter Speicherzugriffe führten. Einige davon gingen auf Fehler im Linker zurück, weil dieser manchmal falsche Segmente in die Variablendeklarationen eingetragen hat. Das Problem trat hauptsächlich bei C++-Konstruktoren und -Destruktoren auf.

```
int main(int argc, char *argv[])
{
    return argc;
}
```

Abbildung 8.2: Testprogramm für den Startcode

Es stellte sich recht bald das Problem des systematischen Tests, denn die wenigen bei der Portierung angefallenen Programme wurden bald korrekt übersetzt. Die kleine Zahl der bisher durchgeführten Tests legte aber die Vermutung nahe, daß noch einige Fehler vorhanden waren.

8.1.2 Der C-Torture-Test für GNU C

Als der Compiler bei den wenigen selbstgeschriebenen Testprogrammen korrekten Code produzierte, wurde begonnen, den C-Torture-Test zu benutzen. Die Ausführungstests konnten noch nicht durchgeführt werden, weil noch keine C-Bibliothek vorhanden war. Aber auch ohne diese Tests wurde eine große Zahl von kleineren Problemen gefunden, die teilweise nicht an der Portierung lagen. Die Beseitigung durch die GNU C-Entwickler ließ oft nicht lange auf sich warten oder war ohne großen Zeitaufwand selbst durchführbar. Beispielsweise war die Abfrage im Compiler, ob ein Zugriff außerhalb des Bereichs der lokalen Variablen durchgeführt wird, nicht korrekt. Eine Konsistenzprüfung hatte bei Objekten der Größe 0 einen Fehler. Datenobjekte der Größe 0 treten beispielsweise bei leeren `struct`-Deklarationen auf, die in C durchaus zulässig sind.

Der C-Torture-Test deckte auch noch umfassende Probleme des Compilers mit Zugriffen auf 16 Bit-Objekte auf. Der MONADS-PC-Prozessor hat keine Befehle dafür, die Zugriffe müssen daher mit anderen Befehlen simuliert werden. Eine Lösung dieser Probleme hat lange auf sich warten lassen, weil zunächst alle Versuche fehlschlugen, die bei der Realisierung mit mehreren Byte-Zugriffen nötige Erhöhung der Adresse zu realisieren. Die Implementierung des Unterprogramms des Codegenerators, das die Zugriffe in einzelne Maschinenbefehle zerlegt, ist nicht einfach, weil es in vielen Phasen des Compilers benutzt werden kann. Jede Phase stellt bestimmte Anforderungen an die Arbeitsweise und die erlaubten Modifikationen an den Befehlsmustern. Die genauen Anforderungen werden aber nicht im Handbuch dokumentiert. Speziell die Änderung von Adreßausdrücken in den späteren Phasen der Übersetzung ist problematisch, weil die Ausdrücke (bzw. Zeiger darauf) teilweise in lokalen Variablen der Optimierungsfunktionen gehalten werden. Erst durch die Analyse mehrerer bestehender Portierungen konnte eine Lösung gefunden werden, die in jeder Situation funktioniert.

Einige Fehler in der Maschinenbeschreibung wurden gefunden, bei denen der Compiler für die betreffende Operation nicht erlaubte Kombinationen von Registern benutzt hat. Der häufigste konzeptbedingte Fehler war die Benutzung eines einzigen Registers für die beiden Parameter der arithmetischen und logischen Be-

fehle, z.B. `adda a`. Diese Befehle sind im Befehlssatz dem MONADS-PC nicht erlaubt, weil sie auf andere Weise effizienter realisiert werden können. Diese Kombinationen wurden daher in der Befehlsbeschreibung ausgeschlossen, was in vielen Fällen nicht die Ideallösung ist. Die bessere Lösung wäre der Einsatz der entsprechenden Befehle (im vorigen Beispiel ein Schiebebefehl). Dies macht die Codegenerierung unübersichtlich, deshalb wurde dieser Weg noch nicht beschritten. Stattdessen wurde die verbotene Kombination von Registern in den Bedingungen der Befehlsmuster vermerkt.

Als alle automatisch durchführbaren Tests die gewünschten Ergebnisse lieferten, wurde die Portierung der GNU C-Bibliothek begonnen. Die ersten darauf folgenden Programmausführungen lokalisierten weitere, im allgemeinen einfach zu korrigierende Fehler. Teilweise gingen die Fehler auf den Linker zurück, der unter bestimmten Bedingungen lokale Symbole falsch auflöste.

Die große Sammlung von Programmen, die zur Laufzeit die Korrektheit verschiedener Codesequenzen ermitteln, wurde bisher nur in begrenztem Umfang zum Test benutzt, da diese Tests sehr zeitaufwendig sind. Der Grund dafür ist die Notwendigkeit, jeden Testfall einzeln auf den MONADS-PC zu übertragen und dort auszuführen. Es wäre sehr wünschenswert, wenn auch dies automatisierbar wäre. Beim derzeitigen Aufbau des Testsystems ist das unmöglich.

8.1.3 Selbstübersetzung von GNU C für MONADS-PC

Als alle bisher beschriebenen Tests keine Fehler mehr aufdeckten, wurde der Versuch unternommen, mit dem bis dahin erstellten Cross-Compiler GNU C zu übersetzen. Dies liefert einen C-Compiler, der auf dem MONADS-PC selbst lauffähig ist, wenn die C-Bibliothek in ausreichendem Maße implementiert wäre.

Bisher ist die C-Bibliothek zwar noch nicht uneingeschränkt einsetzbar, jede Funktion ist aber wenigstens als leere Hülle implementiert, damit Programme erfolgreich assembliert und gebunden werden können. Dies sollte es ermöglichen, zu Testzwecken den GNU C-Compiler mit sich selbst zu übersetzen.

Die ersten Versuche waren nicht sehr erfolgreich, was aber nicht am eigentlichen Compiler lag. Die `Makefile`-Datei hatte einige kleine Fehler, die nur bei der Übersetzung eines Compilers auftraten, der auf einem anderen Rechner laufen wird und für diesen Rechner Code erzeugen soll. Diese waren einfach zu korrigieren.

Die Übersetzung deckte dann noch einige Probleme auf, die bei der Verwendung von Bitfeldern auftraten. Der Compiler verwendet Bitfelder, um bestimmte Daten platzsparend abzuspeichern. Die Verwendung von Bitfeldern löste noch einige Abstürze des Cross-Compilers aus. Die Ursache lag nicht an der Portierung, sondern an Fehlern, die schon sehr lange unentdeckt im Compiler lagen. Die Korrektur war nicht ganz einfach, aber nachdem die richtige Stelle gefunden war, mußte nur sehr wenig geändert werden.

Die Selbstübersetzung verlief danach problemlos. Die Geschwindigkeit des Linkers ließ aber stark zu wünschen übrig. Die Implementierung mit einfach verketteten Listen war für mehrere 10 000 Bezeichner nicht mehr geeignet, da viel zu oft die gesamte Liste durchsucht werden mußte. Die Neuimplementierung der Symboltabelle mit einem Hashverfahren war innerhalb kurzer Zeit fertiggestellt, da bei der ursprünglichen Implementierung darauf geachtet wurde, die Schnittstellen möglichst universell zu halten. Die Ausführungszeit des Linkers (als Referenz wurde der Bindevorgang von `cc1` hergenommen) beschleunigte sich um den Faktor 20 auf ca. 45 Sekunden. Die Hashtabelle wird hierbei nicht optimal genutzt, weil viele vom Compiler erzeugte Bezeichner (z.B. lokale Sprungmarken) in jeder Datei vorkommen und daher prinzipbedingt viele Mehrfachtreffer auftreten. Die durchschnittliche Länge aller Ketten ist dadurch vergleichsweise groß.

Alle Symbole, auch die nur in einer einzigen Datei sichtbaren lokalen Symbole, werden bis zum Ende des Bindevorgangs behalten. So läßt sich die Generierung eindeutiger lokaler Symbole einfach implementieren. Es gäbe auch andere, effizientere Möglichkeiten, dies zu erreichen. Diese Lösung bietet aber den größten Spielraum für die Implementierung verschiedener Strategien der Erzeugung eindeutiger lokaler Symbole. Um den negativen Effekt vieler Mehrfachtreffer zu vermindern, werden die Ketten gleicher Hashwerte nach jeder bearbeiteten Datei umsortiert, damit die globalen Symbole immer am Anfang der Kette sind. Für die meisten Teile des Linkers sind nur die globalen Symbole relevant, somit müssen die vielen lokalen Symbole nur selten zugegriffen werden. Die effektive Länge der Ketten von Mehrfachtreffern verringert sich so stark.

8.2 C-Bibliothek

Der Test der C-Bibliothek ist wegen der prototyphaften Implementierung nicht systematisch betrieben worden. Das bisherige Kriterium war nur, daß einfache Programme lauffähig sein sollten. Die Frage nach einer korrekten Implementierung stellt sich erst dann, wenn die nur teilweise portierten Funktionen durch vollständige Implementierungen ersetzt worden sind.

Viele Funktionengruppen sind bisher noch nicht funktionsfähig, weil z.B. die Implementierung der Dateioperationen `open()`, `read()` usw. noch nicht, oder nur soweit es unbedingt nötig war, fertiggestellt wurde.

Getestet wurde der Startcode und Funktionen, die für jedes Programm implizit erforderlich sind. Die C-Bibliothek benötigt für die eigene Initialisierung schon eine Vielzahl von Funktionen, die größtenteils portabel sind. Dazu gehören auch komplexe Funktionen wie `malloc()`, die im Initialisierungscode verwendet werden. Auch sehr kleine Programme kommen so auf eine Größe von 60 KByte Code und Daten. Dies zeigt deutlich die Folgen statischer Bindung, denn der Anteil von Bibliotheksfunktionen ist bei vielen kleinen Programmen höher als der Code des Programms.

Als weiteres „Testprogramm“ diente GNU Go, eine Implementierung des japanischen Brettspiels Go. Dieses Programm wurde gewählt, da es nur sehr geringe Ansprüche an die C-Bibliothek stellt. Das Programm ist komplex genug, um eine Aussage über die Funktionsfähigkeit größerer übersetzter Codestücke zu erlauben. Die vorige Portierung von GNU C hatte mit solchen relativ lang laufenden Programmen Probleme, weil durch die häufige Reservierung von Segmenten auf dem Heap viel Platz verschwendet wurde. Es mußte nachträglich eine Wiederverwendung von Segmenten implementiert werden, um das Fehlen eines Garbage Collectors auszugleichen. Dieses Problem tritt bei der neuen Portierung nicht in diesem Maß auf, da für die Parameterübergabe und für lokale Variablen kein eigener Platz auf dem MONADS-Heap verwendet werden muß. Die Benutzung des MONADS-Heaps für solche Daten ist eine Folge der in C möglichen Speicherung von Verweisen auf diese Daten. Da bei der neuen Portierung nur beim Aufruf von anderen Modulen Strings in Segmente auf dem MONADS-Heap umkopiert werden müssen, ist das Fehlen des Garbage Collectors keine echte Einschränkung. Auch hier werden Segmente von Anfang an soweit irgendwie möglich wiederverwendet.

Bisher wurden keine Geschwindigkeitsmessungen des übersetzten Codes durchgeführt. Die genaue Ausführungszeit war noch nicht relevant, und es wurde nur wenig Wert auf die Effizienz des generierten Codes gelegt. Es wurde zwar möglichst effizienter Code für den Funktionsanfang und das Funktionsende erzeugt. Sonst wurden aber keine Optimierungen angebracht, die über die generischen, für alle GCC-Portierungen gemeinsamen Optimierungsverfahren hinausgehen. Daher wären absolute Geschwindigkeitsangaben eher ein Indiz für die Eignung der generischen Optimierungsverfahren, die wegen der Registerknappheit teilweise kaum optimieren können. Die Ausführungszeit der verschiedenen Testprogramme deutet auf generell akzeptablen Code hin.

8.3 Spät bemerkte Fehler

Nach der grundlegenden Implementierung der C-Bibliothek war der Compiler schnell zur Portierung größerer Programme geeignet. Die meisten Programme liefen auf Anhieb. Es sind aber zwei Fehler aufgetreten, als die Existenz solcher grundlegender Fehler eigentlich schon für unmöglich gehalten wurde. Dies zeigt deutlich, wie schwierig es ist, Entwicklungssysteme zu testen.

Der erste war eigentlich ein gemeinsamer Fehler im Assembler und Linker. Beide Programme erzeugten Code, der bei uninitialisierten Variablen die Bedingungen für die Ausrichtung von Variablen verletzte. Als Folge wurde mit Wortzugriffen auf nicht ausgerichtete Adressen zugegriffen. Auf dem MONADS-PC werden solche Fehler nicht erkannt, das Programm läuft mit undefinierten Werten weiter. Der Testrechner hat eine kleine Zusatzhardware eingebaut, der in diesem Fall den Rechner anhält. Nur deshalb konnte das Auftreten dieses Fehlers schnell erkannt werden.

Der Fehler konnte sich aus zwei Gründen lange halten:

- Programme verwenden fast ausschließlich Daten, deren Größe durch vier teilbar ist.
- Programme, die kleinere Datenobjekte enthalten, verwenden meist durch vier teilbare Gruppen solcher Variablen.

Der zweite Fehler lag in einer fehlerhaften Definition der Änderung des Prozessorbedingungscodes bei Vergleichsbefehlen. Der Optimierer entfernte Befehle, die für den korrekten Ablauf unbedingt erforderlich waren. Dieser Fehler tritt nur äußerst selten auf, da die Bedingungen, die eine Optimierung überhaupt erlauben, nur selten gegeben sind. In den meisten Fällen ist die Optimierung dennoch korrekt, denn die Bedingungen, die den Fehler auftreten lassen, sind nur sehr selten erfüllt.

Das Auftreten dieser Fehler zeigt, wie notwendig ein weiterer Test des Compilers durch umfangreichere Benutzung wäre. Nur so kann ein Zustand erreicht werden, an dem von Zuverlässigkeit ausgegangen werden kann.

Derzeit ist nur ein einziger Fehler des Compilers bekannt, der bei Verwendung von 64 Bit breiten Variablen auftritt. Der GNU C-Compiler bietet für solche Daten den Typ `long long int` an. In seltenen Fällen müssen solche Objekte aus dem Bereich für die ausgehenden Parameter in eine lokale Variable kopiert werden. Dies schlägt fehl, da der Compiler nur für Datengrößen, die der Prozessor direkt verarbeiten kann, korrekten Code erzeugt. Andere Portierungen haben das gleiche Problem. Sie verwenden als „Workaround“ entweder die von dieser Portierung benutzte Optimierung der Stackverwaltung nicht oder implementieren auch sämtliche Befehle für 64 Bit-Daten. Diese Lösung führt zu einer sehr großen und komplexen Befehlsbeschreibung, die daher schwer wartbar ist. Beide Lösungen sind nicht befriedigend. Das Problem sollte im prozessorunabhängigen Code des Compilers gelöst werden, damit alle Portierungen davon profitieren können.

8.4 Vergleich mit MONADS-Pascal

Ein Vergleich des GNU C-Compilers mit MONADS-Pascal kann selbstverständlich nur in bestimmten Bereichen erfolgen, denn die Möglichkeiten und Grenzen der Compiler sind zu unterschiedlich. Um eine Vorstellung von der Codegüte der beiden Compiler zu erhalten, wurde auf beiden Compilern ein einfaches Programm zur Primzahlberechnung übersetzt. Dieses Programm wurde auch schon in [11] zum Vergleich benutzt. In der Zwischenzeit wurde der MONADS-Pascal-Compiler deutlich verbessert, so daß die damals geltenden Grundlagen sich geändert haben. Die Meßlatte liegt daher höher. Andererseits wurde bei der Portierung des GNU C-Compilers kein Aufwand für die Verbesserung des generierten Codes getrieben. Der Vergleich zwischen den Compilern hat sich von einem praktisch nicht optimierenden Pascal-Compiler und einem recht gut optimierenden, aber

ebenfalls nicht ausgereizten C-Compiler zu einem ausgewogeneren Bild verschoben. Der Pascal-Compiler setzt mittlerweile leistungsfähige Optimierungen ein, der GNU C-Compiler wurde dagegen vergleichsweise wenig verbessert. Die neue Portierung des C-Compilers kann durch die Verwendung des neuen Befehlssatzes besser optimieren. Viele Asymmetrien im Befehlssatz wurden beseitigt und ein Register wurde zusätzlich verfügbar. Der Vergleich ist also bis zur Anpassung des Pascal-Compilers etwas unfair. Eine Übersetzung in einem Durchlauf kann die komplexen Algorithmen zur Registerverteilung nicht einsetzen, und so wird der Code immer etwas schlechter sein.

Die beiden Programme zur Berechnung von Primzahlen wurden möglichst ähnlich implementiert, wobei die Pascal-Implementierung die Vorlage lieferte. Der letzte Vergleich in [11] implementierte die Berechnungsfunktion zwar ebenfalls praktisch identisch, jedoch wurden Anpassungen an den Variablendeklarationen vorgenommen, die einen direkten Vergleich unmöglich machten. Die Quelltexte unterschieden sich durch die Benutzung globaler Variablen in der Pascal-Version bzw. lokaler Variablen in der C-Version. Der aktuelle MONADS-Pascal-Compiler akzeptiert keine Standard-Pascal-Programme mehr, was ein weiterer Grund für eine Neuimplementierung war. Die neue Implementierung benutzt für alle Daten lokale Variablen.

Bild 8.3 zeigt die beiden Implementierungen im Quelltextvergleich. Der direkte Vergleich zeigt, daß das C-Programm deutlich kürzer als das entsprechende MONADS-Pascal-Programm ist. Dies liegt hauptsächlich an der in C nicht notwendigen Klassendefinition und der Initialisierung der Ausgabe. Der Längenunterschied resultiert also weniger aus der umfangreicheren Implementierung als aus den erweiterten Möglichkeiten von MONADS-Pascal. Bei realem Code wird der Längenunterschied geringer ausfallen.

Die Schleife zum Überspringen von Zahlen, die keine Primzahlen sind, ließe sich in C viel kürzer programmieren. Einige eingefleischte C-Programmierer sind sicher der Überzeugung, daß kürzer Code immer übersichtlicher ist und würden `while (prim[++z]) ;` statt der `do/while`-Schleife in Bild 8.3 verwendeten. GNU C erzeugt aber für beide Varianten exakt den gleichen Code, es kann daher ohne Bedenken die (in meinen Augen leichter verständliche) längere Formulierung gewählt werden.

Die beiden Compiler mußten den Quelltext bei jeweils bestmöglicher Optimierung übersetzen, im Fall von MONADS-Pascal ohne Prüfung auf Bereichsüberschreitung, um Chancengleichheit zu gewährleisten. Ein Vergleich des erzeugten Codes erfolgte nur für den im Quelltext markierten Berechnungsteil. Die Ausgabefunktionen sind sehr unterschiedlich implementiert, ein Vergleich ist daher nicht sinnvoll. Ebenso macht ein Vergleich des Codes für den Prozeduranfang und das Prozedurende nur wenig Sinn, da diese indirekt vom Implementierungskonzept des Compilers vorgegeben werden. Bild 8.4 stellt diesen Hauptteil der Funktionen gegenüber. Der vom Pascal-Compiler erzeugte Assemblertext ist vereinfacht (ohne die erzeugten Befehle zu beeinflussen), denn der Compiler er-

```
primes module sieve;

type
  primes= class
    procedure open(tty: modcap);
    procedure PrimeNumbers;
  end;

const
  Size = 1000;
  Root = 32;

retained
  output: text;

interface

procedure open(tty: modcap);
begin
  mrewrite(output, tty);
end;

procedure PrimeNumbers;
var
  prim: array [0..Size] of boolean;
  n, z, mult: integer;
begin
  for n := 0 to Size-1 do
    prim[n] := true;

  z := 2;
  while z <= Root do begin
    mult := z + z;

    while mult < Size do begin
      prim[mult] := false;
      mult := mult + z;
    end;

    repeat
      z := succ(z);
    until prim[z];
  end;

  for n:=2 to Size-1 do
    if prim[n] then
      write(n, chr(9));

  writeln;
end;
.
```

```
#include <stdio.h>

#define Size 1000
#define Root 32

int main(void)
{
  unsigned char prim[Size];
  int n, z, mult;

  for (n = 0; n < Size; n++)
    prim[n] = 1;

  z = 2;
  while (z <= Root) {
    mult = z + z;

    while (mult < Size) {
      prim[mult] = 0;
      mult += z;
    }

    do {
      z++;
    } while (!prim[z]);
  }

  for (n = 2; n < Size; n++)
    if (prim[n])
      printf("%d\t", n);

  printf("\n");
  return 0;
}
```

Abbildung 8.3: Sieb des Erathostenes in MONADS-Pascal und C

zeugt oft mehrere Sprungmarken für eine Stelle im Code. Dies würde hier nur unnötig Platz kosten und den Vergleich erschweren.

Der Pascal-Compiler erzeugt 30 Maschinenbefehle, der C-Compiler 23 Maschinenbefehle. Die Codelängen berücksichtigen die `setl`-Befehle nicht. `setl`-Befehle dienen dem Debugging von Programmen und zur leichteren Übersicht. Die Unterschiede in der Zahl der Befehle gehen auf mehrere Faktoren zurück:

- Der Pascal-Compiler generiert oft Sprünge auf den Folgebefehl (im Beispiel dreimal). Dies ist ein durch die Einpass-Übersetzung nur schwer zu vermeidendes Problem. Der Pascal-Compiler erzeugte hier in einer älteren Version jedesmal eine Befehlssequenz, die mehrere Capability-Register mit bestimmten Werten vorbesetzt hat. Diese Ladebefehle sind relativ teuer und deswegen wurde der erzeugte Code so abgeändert, daß nur die vom Schleifenrumpf benötigten Werte geladen werden. Es ist aber erst nach vollständiger Übersetzung des Schleifenrumpfs bekannt, welche Register benötigt werden. Somit muß ein Einpass-Compiler einen Sprung auf den später erzeugten Ladecode in den Quelltext schreiben. Wenn aber keine Register geladen werden müssen, wird nicht einfach ein Sprung zurück an den eigentlichen Schleifenanfang generiert, sondern der Wert der ursprünglich angesprungenen Marke auf den Schleifenanfang gesetzt. Dies ist durch die Kürzung des Assemblertexts weniger offensichtlich. Das Verfahren ist die bestmögliche Lösung dieses Optimierungsproblems, die ohne Umordnung des erzeugten Codes auskommt.

- Der C-Compiler erkennt gemeinsame Berechnungen in verschiedenen Ausdrücken und kann die Addition von `+ z` im Ausdruck `mult = z + z` durch Wiederbenutzung eines Teils der Anweisung `mult += z` innerhalb der Schleife eliminieren. Somit muß für die Anweisung `mult = z + z` nur ein Ladebefehl und ein Sprung ans Ende der Schleife erzeugt werden.

- Der C-Compiler benötigt weniger Sprungbefehle, weil die Schleifen optimal angeordnet werden. Diese Optimierung ist aber nur in Mehrpass-Compilern möglich, da Programmteile umgeordnet werden.

Die Laufzeit des C-Programms dürfte deutlich besser sein, denn die Schleifen enthalten deutlich weniger Befehle. Die konsequente Benutzung aller verfügbaren Register dürfte für eine Verbesserung der Ablaufgeschwindigkeit sorgen. GNU C benötigt keine lokalen Variablen außer dem Feld `prim`. Die verbleibenden Variablen `mult` und `z` werden während ihrer gesamten Lebensdauer in Registern gehalten. Der C-Compiler entfernt sogar die sonst erforderliche Multiplikation (oder den Shift) in der Schleife, die Nicht-Primzahlen überspringt, zugunsten eines weiteren Registers, das direkt die Adresse des entsprechenden Feldelements enthält. Diese Optimierung spart zwar keinen Befehl, die Initialisierung der Schleife wird im Gegenteil sogar länger. Dies ersetzt aber einen Befehl in der Schleife durch einen sehr billigen.

Der ganze Aufwand, der im GNU C-Compiler betrieben wird, um maschinenunabhängige Optimierungen zu implementieren, zahlt sich offensichtlich aus. Der

```
    setl   #31
    lda    #2      ; z := 2
    sta    4(c1)
    setl   #32
    bunc   Z11
Z11:
           ; while z <= Root
    lda    4(c1)
    cmpai  #32
    bgt    Z13
Z12:
    setl   #33
    lda    4(c1)   ; mult := z + z
    adai   4(c1)
    sta    8(c1)
    setl   #35
    bunc   Z16
Z16:
           ; while mult < Size
    lda    8(c1)
    cmpai  #1000
    bge    Z18
    setl   #36
           ; prim[mult] := false
    lda    8(c1)
    scalea
    ldi    i1,a
    lda    #0
    sta    (c3)[i1]
    setl   #37
           ; mult := mult + z
    lda    8(c1)
    adai   4(c1)
    sta    8(c1)
    setl   #38
    bunc   Z16
Z18:
    setl   #40
    bunc   Z21
Z21:       ; repeat
    setl   #41
    inc    4(c1)   ; z := succ(z)
    setl   #42
           ; until prim[z]
    lda    4(c1)
    scalea
    ldi    i1,a
    lda    (c3)[i1]
    beq    Z21
    setl   #43
    bunc   Z11
Z13:
```

```
    setl   #31
    ldi    i2,#2   ; z = 2

    setl   #32
           ; while (z <= Root)
    ldi    i3,i0
    addi   i3,#-4012



LL14:
    setl   #33
    lda    i2      ; mult=z...
    setl   #35
    jump   #LL30



LL17:
    setl   #36
           ; prim[mult] = 0
    ldi    i1,ax
    ashfti i1,#2
    addi   i1,i3
    clr    (cx)[i1]
    setl   #37
           ; mult += z
    lda    ax
LL30:
    adda   i2
    ldax   a
    setl   #38
           ; while (mult < Size)
    cmpa   #999
    ble    LL17
    setl   #40
           ; do
    ldi    i1,i2
    ashfti i1,#2
    addi   i1,i3
LL19:
    setl   #41
    addi   i1,#4   ; z++
    addi   i2,#1
    setl   #42
           ; while (!prim[z])
    tst    (cx)[i1]
    beq    LL19
    setl   #43
           ; while (z <= Root)
    cmpi   i2,#32
    ble    LL14
```

Abbildung 8.4: Assemblercode von MONADS-Pascal und GNU C

erzeugte Code ist effizient und kompakt, speziell beim untersuchten Beispiel wird es schwer sein, eine effizientere Lösung zu finden. Das Beispiel ist keineswegs repräsentativ, denn die Codegröße ist sehr klein und der Algorithmus und die Abhängigkeit der Daten sind einfach. Der erzeugte Code ist auch bei größeren Funktionen und Programmen durchaus mit anderen Portierungen von GNU C vergleichbar. Als Repräsentant wurde ein Teil des Compilers selbst gewählt, `cc1`. Eine für SPARC-Prozessoren übersetzte Version (hier wurde schon viel Optimierungsarbeit geleistet) ist statisch gebunden etwa 2 MByte groß, die Version für den MONADS-PC ist mit etwa 2,5 MByte in der Größe vergleichbar. Der Unterschied dürfte hauptsächlich in der Anzahl der verfügbaren Register liegen, denn eine ausreichende Zahl Register erlaubt es, auch große Funktionen ohne dauerndes Zwischenspeichern von Registern zu übersetzen. Generell wird also Code von zufriedenstellender Güte erzeugt.

Kapitel 9

Ergebnisse

9.1 Ausblick

Es ist bei Projekten wie der Portierung eines Compilers unmöglich, innerhalb eines halben Jahres Perfektion in Form einer vollständigen Implementierung und uneingeschränkter Benutzbarkeit des gesamten Entwicklungssystems zu erreichen. Was schon bei einem vom Entwurf her äußerst portablen Compiler gilt, trifft bei der Portierung einer C-Bibliothek ebenso zu. Die Portierung einer C-Bibliothek ist eine sehr zeitaufwendige Arbeit. Die Bibliothek enthält viele Stellen, an denen bestimmte Eigenschaften des Betriebsystems bzw. des Prozessors berücksichtigt werden müssen. Der Schwerpunkt dieser Diplomarbeit lag daher bei der Schaffung einer soliden Basis für weitere Entwicklungen.

9.1.1 Verbesserung des Compilers

Bei der Portierung des Compilers wurde bisher kein größerer Aufwand getrieben, den erzeugten Code zu verbessern. Die Betonung lag auf der Zuverlässigkeit des erzeugten Codes. Es sind keine Fehler des Compilers bekannt, die spezifisch für die durchgeführte Portierung sind. Die Korrektheit ist also derzeit nicht widerlegt. Der Code ist aber teilweise weit von der Codegüte entfernt, die auf anderen Plattformen erreicht wird. Die vom Compiler angebotenen Möglichkeiten in Hinsicht auf Registerausnutzung und Wahl der optimalen Befehle werden nicht ausgeschöpft. Der Compiler hat enorme Schwierigkeiten, mit der sehr geringen Registeranzahl und den teilweise starken Einschränkungen der erlaubten Operanden zurechtzukommen. Wenn dem Compiler nur sehr wenig Register zur Verfügung stehen und diese nicht in allen Situationen einsetzbar sind, dann werden viele Transfers zwischen den Registern notwendig. Dies erhöht den Druck auf die Registerverteilung noch weiter.

Zuerst müßten die für viele Operationen geltenden Einschränkungen für das dritte Operandenregister entfernt werden, und dann die entsprechende Ersatzfunktion benutzt werden. Beispiel: ohne die in `mpc.md` beim Befehlsmuster `addsf3`

(Gleitkommaaddition) angegebene Einschränkung, daß der Akkumulator nicht gleichzeitig das Zielregister und das zweite Quellregister sein darf, würde der Compiler `fpadda a` als gültigen Befehl ansehen und ihn bei Bedarf in den Assemblerquelltext schreiben. Diesen Befehl kennt der MONADS-PC nicht. In vielen Situationen, in denen Beschränkungen für die Wahl des zweiten Operanden existieren, kann ein äquivalenter Ersatzbefehl angegeben werden. Beim oben gezeigten Beispiel wäre dies der Befehl `fpmula #$40000000`, der eine Multiplikation[9] mit `2.0` durchführt. In diesem Fall ist der Ersatzbefehl sogar deutlich schneller als der ursprüngliche Befehl. Diese Fälle sollten also nicht verboten, sondern erkannt und ausgenutzt werden. Dies erfordert eine Erweiterung der Beschreibung der Maschinenbefehle durch C-Code. Diese Erweiterung wurde bisher vermieden, um die Codeerzeugung möglichst lesbar und leicht verständlich zu halten.

Danach können noch verschiedene Peephole-Optimierungen für verbleibende, von den Standard-Optimierungen noch nicht hinreichend optimierte Fälle geschrieben werden. Dies sollte erlauben, die manchmal auftretenden sinnlosen Zwischenspeicherungen von Registern zu reduzieren. Der lokal manchmal sehr ineffiziente Code sollte auf diese Weise deutlich verbessert werden.

Eine weitere Einschränkung ist derzeit durch die direkte Verwendung bedingter Sprünge gegeben. Der MONADS-PC kann nur eine bestimmte Distanz vorwärts bzw. rückwärts springen, da die Befehle für bedingte Sprünge nur 22 Bit-Entfernungen (inkl. Vorzeichen) enthalten können. Bei sehr großen Schleifen kann dies dazu führen, daß C-Programme nicht übersetzbar sind. GNU C bietet eine komfortable Möglichkeit an, die Sprungentfernung abzuschätzen. So kann entschieden werden, ob der normale bedingte Sprung noch einsetzbar ist oder durch eine Kombination eines bedingten und eines unbedingten Sprungs ersetzt werden muß. Unbedingte Sprünge haben keine Einschränkungen bezüglich des Sprungziels.

9.1.2 Behebung der Einschränkungen der Portierung

Die Portierung berücksichtigt zwei, vom Compiler eigentlich angebotene Funktionalitäten nicht. Diese sind aber in C irrelevant, da sie nicht benutzt werden, außer von GNU C-spezifischen Spracherweiterungen.

Die erste ist die Implementierung von sogenannten Trampolinen. Die Aufgabe eines Trampolins ist die Unterstützung von Zeigervariablen, die Funktionsadressen enthalten, die auf anderen lexikalischen Ebenen liegen. Verschachtelte Funktionen sind eine GNU C-spezifische Erweiterung, die fast nie benutzt wird. Bei der Unterstützung von beliebiger Schachtelung von Funktionen ist es notwendig, den statischen Vorgänger korrekt zu setzen. Der Compiler verwendet die gleiche Darstellung für Funktionszeiger und für Zeiger auf Daten. Die Information über den statischen Vorgänger paßt nicht in den Zeiger selbst. Es wird zusätzlicher Code

[9] `40000000` ist die für IEEE-754 single precision gültige Hexadezimaldarstellung von `2.0`.

erzeugt, der bei der Bestimmung der Adresse von Funktionen, die in andere geschachtelt sind, die Adresse eines zur Laufzeit generierten Stücks Programmcode liefert. Dieser Programmcode besetzt den Verweis auf den statischen Vorgänger mit dem korrekten Wert und ruft danach die eigentliche Funktion auf, daher der Name Trampolin.

Eine solche Konstruktion ist auf dem MONADS-PC derzeit nicht möglich. Code kann nur in speziellen Adreßräumen ausgeführt werden, die aber vom Programm nicht selbst veränderbar sind. Daher ist die Implementierung von Trampolinen auf dem MONADS-PC unmöglich. Eine optimale Lösung setzt die Möglichkeit der Codeausführung auf dem Stack voraus, denn so kann die Freigabe des generierten Codes zum richtigen Zeitpunkt erfolgen. Ein Problem dieser Implementierung ist, daß potentiell nach dem Rücksprung aus der Funktion, die das Trampolin angelegt hat, dieses nicht mehr existiert. Es könnten immer noch Verweise darauf existieren. Dies ist bei den Sprachen, die solche Konzepte anbieten, verboten. Es gibt aber keine Möglichkeit, dies in allen Fällen zu garantieren. Bei fehlerhaften Programmen kann die Stelle auf dem Stack ausgeführt werden, obwohl der Code schon überschrieben wurde. Das führt zu schwer lokalisierbaren Folgefehlern. Dies beeinträchtigt aber nur die Robustheit eines Programms, nicht die Sicherheit des Gesamtsystems.

Die zweite ergibt sich als Konsequenz aus der ersten nicht implementierten Funktionalität: die komplette Nichtberücksichtigung des Verweises auf den statischen Vorgänger. Dies bedeutet, daß bisher in der Aufrufkonvention kein Platz für einen Verweis auf den statischen Vorgänger reserviert ist. Der Platz wäre bis zu einer Implementierung von Trampolinen in jedem Fall verschwendet. Die Änderung an der Maschinenbeschreibung ist sehr klein, deshalb wurde die Implementierung verschoben.

9.1.3 Implementierung von Modulen mit GNU C

Die bisherige Portierung des GNU C-Compilers übersetzt nur Programme, die mit der Standard-C-Semantik entwickelt wurden. Dies äußert sich in der feststehenden Schnittstelle von C-Programmen, die nur den Aufruf von `main()` vorsieht. Auf dem MONADS-PC wäre es wünschenswert, wenn andere Funktionen ebenfalls als Schnittstellenfunktion zugänglich wären. Damit könnte man allgemeine Code-Module erstmals auch in einer Art C implementieren. Die Änderungen am Compiler wären nur gering. Jedoch ist zu berücksichtigen, daß beispielsweise Aufrufe von Konstruktoren und Destruktoren (siehe auch Abschnitt 9.1.6) auf andere Weise als bisher organisiert werden müßten. Man könnte sie nicht mehr beim Aufruf von der Schnittstellenfunktion `main` aufrufen, sondern müßte schon beim Aufruf der `open`-Schnittstellenfunktion diesen Initialisierungscode ausführen. Dies würde auch Änderungen am Initialisierungszeitpunkt der C-Bibliothek erfordern. Die Konsequenz daraus wäre, daß ein wiederholter Aufruf von Programmen nur durch erneutes Öffnen möglich wäre. Das ist weniger komforta-

bel als die bisherige Implementierung, bei der einfach die Schnittstellenfunktion `main` erneut aufgerufen werden kann.

Die Implementierung allgemeiner Code-Module ist ohne die Realisierung eines Konzepts für die Spezifikation der Schnittstelle des Moduls unvollständig. Es ist unerwünscht, jede global sichtbare Funktion als Schnittstellenfunktion verfügbar zu haben. Dies würde umfangreiche Arbeiten erfordern, denn hierfür muß zuerst ein sauberes Sprachkonzept gefunden werden. Ob das für C überhaupt sinnvoll ist, muß bezweifelt werden. C++ und ähnliche Sprachen wären hierfür die bessere Ausgangsbasis.

Die Implementierung von Typemanagern in C ist dagegen schwieriger. Die Unterstützung der verschiedenen Lebensdauer von `instance`- und `retained`-Variablen ist z.B. durch eine Spracherweiterung realisierbar. Kleine Experimente mit einer Lösung, die auf einer in GNU C schon vorhandenen Spracherweiterung basiert, verliefen erfolgversprechend. Der GNU C-Compiler bietet die Möglichkeit, durch die Angabe von `attribute(())` Zusatzeigenschaften von Variablen und Funktionen festzulegen.

Alternativ könnte eine Implementierung über Bibliotheksfunktionen erfolgen. Analog zu `malloc` könnte es z.B. eine Funktion `instance_malloc()` geben, die für die Verwaltung von Instanzdaten zuständig ist. Diese Realisierung würde aber nur dynamische Speicheranforderung erlauben.

Ein weiteres Problem bei Typemanagern ist die Notwendigkeit, Zugriffe auf die Instanzdaten zu synchronisieren. Ein ähnliches Problem tritt bei der Unterstützung von Threads unter UNIX auf. Die GNU C Library unterstützt zwar Threads unter UNIX, aber dies muß noch auf Eignung für den MONADS-PC geprüft werden. Es kann auch zu Problemen beim Aufruf von anderen Modulen kommen, denn der bisher implementierte Stub-Generator für die Erzeugung des Konvertierungscodes für den Aufruf von Modulen produziert nicht wiedereintrittsfähige Funktionen (im Kontext von Typemanagern). Dies ist mit einigen Änderungen am Stub-Generator korrigierbar. Der nicht wiedereintrittsfähige Code ist einfacher und schneller als die wiedereintrittsfähige Lösung.

9.1.4 Verlagerung der C-Bibliothek in ein Modul

Bestimmte Funktionen der C-Bibliothek, z.B. die Ein- und Ausgabe von Gleitkommawerten und die Funktionen, die mit regulären Ausdrücken arbeiten, benötigen sehr viel Platz. Sie werden aber vergleichsweise selten verwendet, um die Codegröße zu rechtfertigen, die aus ihrer indirekten Referenzierung beim statischen Binden entsteht. Verwendet ein Programm beispielsweise die Funktion `printf()`, um Text auf dem Bildschirm auszugeben, so werden auf jeden Fall die Funktionen zur Ausgabe von Gleitkommawerten dazugebunden. Die Codegröße schnellt deshalb hoch. Selbst beim einfachen „Hello world“-Programm beträgt sie gut 200 KByte. Hier könnte ein ähnlicher Mechanismus, wie er im MONADS-Pascal-Compiler verwendet wird, die Codegröße verringern. Man

könnte in der C-Bibliothek einzelne Funktionen durch Stubs ersetzen, die entsprechende Funktionen in einem C-Bibliotheksmodul aufrufen. Dies entspricht einer dynamischen Bindung des Bibliothekscodes zur Laufzeit des Programms.

Dabei ist zu beachten, daß von einem C-Bibliotheksmodul keine Zugriffe auf globale Variablen möglich sind. Diese müßten als Parameter übergeben werden. Das ist bei geeigneter Verteilung der Aufgaben ein vergleichsweise kleines Problem.

Wenn der Compiler schon für die Unterstützung beliebiger Code-Manager erweitert wurde, ist dies sehr einfach. Die auszulagernden Funktionen müssen nur noch lokalisiert werden und durch Aufrufe an das Bibliotheksmodul ersetzt werden. Das Bibliotheksmodul wird dann ebenfalls vom C-Compiler übersetzt. Andernfalls ist es problemlos möglich, Teile der C-Bibliothek nach Pascal zu konvertieren und den MONADS-Pascal-Compiler zur Übersetzung der Bibliothek zu verwenden. Die Konvertierung von C nach Pascal ist nicht einfach, und die in Frage kommenden Funktionen sind gerade sehr komplex. Es ist daher wahrscheinlich einfacher, den C-Compiler zu erweitern.

9.1.5 Erweiterung des Assemblers

Bei der Implementierung der C-Bibliothek traten früher immer wieder Probleme auf, wenn ein Benutzerprogramm eine Funktion mit gleichem Namen wie eine Bibliotheksfunktion definierte. Es konnte vorkommen, daß ein Programm nicht gebunden werden konnte, weil es Kollisionen mit den Bezeichnern der Bibliotheksfunktionen enthielt. Dieses Problem wurde in der aktuellen ISO C-Standardisierung über einen Spezialmechanismus zur Lösung dieser Symbolkonflikte gelöst. Die C-Bibliothek definiert die Funktionen nie direkt, sondern immer mit zwei vorangestellten Unterstrichen. Solche Symbole dürfen nach dem C-Standard nicht in Benutzerprogrammen verwendet werden. Die Kompatibilität wird durch die Definition eines speziellen Alias hergestellt, der nur gültig ist, wenn das Symbol nicht vom Benutzerprogramm verwendet wird. Die C-Bibliothek verwendet intern grundsätzlich die Definition mit zwei Unterstrichen, damit nicht versehentlich Benutzercode statt einer Bibliotheksfunktion an eine Symbolbenutzung in der C-Bibliothek gebunden wird.

Weder der neu implementierte Assembler noch der MONADS-Assembler unterstützen dieses Verfahren. Somit ist die C-Bibliothek in der derzeitigen Form nicht vollständig standardkonform.

Die Realisierung dieser Eigenschaften wird über eine spezielle Symboldeklaration vorgenommen, sogenannte `weak`-Symbole. Ihr Verhalten unterscheidet sich bei Mehrfachdefinitionen bzw. fehlenden Definitionen von normalen globalen Symbolen. Ein als `weak` deklariertes Symbol wird, falls ein normales Symbol mit gleichem Namen existiert, einfach ignoriert und der Wert des normalen Symbols benutzt. Wenn ein `weak`-Symbol nur deklariert wird, aber keine Definition eines Wertes erfolgt, wird dem Symbol der Wert `0` zugeordnet.

Diese Erweiterung kann im Assembler und im Linker implementiert werden, die diese Referenzen ebenfalls statisch auflösen. Änderungen an den neu entwickelten Assembler- und Linkerprogrammen reichen dazu aus. Eine Änderung am MONADS-Assembler ist nicht erforderlich.

9.1.6 Einsatz von GNU C++

Die nächste Sprache, die nach GNU C zur Benutzung auf dem MONADS-PC in Betracht gezogen werden sollte, ist GNU C++. Einerseits bietet C++ deutlich mehr Möglichkeiten zur Strukturierung von Problemen, andererseits sollte der Unterschied klein genug sein, diese Sprache ohne Änderung an der bisherigen Portierung zu verwenden. Die Portierung des GNU C-Compilers wurde mit Hinblick auf den C++-Compiler durchgeführt. Es wurden aber wurden keine Tests der Funktionsfähigkeit durchgeführt. Die von C++ eingeführte Möglichkeit, statische Konstruktoren und Destruktoren zu definieren, die vor bzw. nach `main()` ausgeführt werden, ist bereits berücksichtigt. Es sollte untersucht werden, inwieweit die Möglichkeit, Module mit objektorientierten Methoden zu implementieren, die Wiederverwendbarkeit des Codes erhöht. MONADS-Module kennen bisher Vererbung nur auf Typebene. Wiederverwendung von Code bei der Implementierung von Subklassen ist nicht möglich.

Die Möglichkeit, kleine Objekte innerhalb der großen, von der Architektur vorgegebenen Objekte zu definieren, erlaubt beispielsweise die Verwendung komplexer Datentypen, ohne diese aus Effizienzgründen jedesmal neu implementieren zu müssen. Ein auf dem MONADS-PC häufiger auftretendes Problem ist die Implementierung von Listen aller Art. Man könnte einen Listen-Typemanager schreiben, der immer eingesetzt wird, wenn eine Liste benötigt wird. Oft sind Listen aber relativ klein und rechtfertigen den Aufwand für die Kapselung und den Schutz nicht. Viele Module implementieren selbst eine einfache Listenverwaltung und verzichten aus Effizienzgründen auf Modularisierung.

Dieser Ausweg wird ebenfalls oft gewählt, weil beim Design eines Moduls noch nicht bekannt ist, wie es am Ende zu implementieren ist. Da es aber ohne die entsprechenden Module Capabilities nicht möglich ist, externe Listen oder sonstige Datenstrukturen anzulegen, wird auf die Verwendung verzichtet, um die Schnittstelle der Klasse von Implementierungsdetails freizuhalten.

9.1.7 Andere Sprachen

Es ist noch zu untersuchen, inwieweit andere Sprachen der GNU-Sprachenfamilie mit der derzeitigen Portierung zurechtkommen. Die Portierung von GNU Pascal ist aus den oben genannten Gründen nicht einfach. Derzeit sind einige Funktionen noch nicht implementiert, die bei Sprachen benötigt werden, die lexikalische Ebenen unterstützen. Ein Vergleich zwischen MONADS-Pascal und GNU Pascal wäre sehr interessant.

Eine Vielzahl der Sprachen sollte einsetzbar sein, wenn auch teilweise mit Einschränkungen. Beispielsweise wäre der Einsatz von GNU Java interessant, denn diese Sprache wird in diesem Fall nicht in Byte Code für die Java Virtual Machine, sondern direkt in Maschinencode übersetzt. Damit fiele die Portabilität des Codes weg, aber die Portabilität des Quelltexts ist davon nicht beeinträchtigt. Die Benutzung der Java Virtual Machine würde aufgrund der geringen Geschwindigkeit des MONADS-PC zu einer unakzeptablen Ausführungsgeschwindigkeit führen.

9.2 Zusammenfassung

Beim Test des Compilers mit den in der C-Testsuite vorgegebenen Testfällen, der Übersetzung mit sich selbst und der Übersetzung der C-Bibliothek sind keine Fehler mehr aufgetreten. Da die C-Bibliothek nur teilweise funktionsfähig ist und auch noch nicht in größerem Umfang getestet wurde, ist es möglich, daß noch kleinere Fehler in der Portierung vorhanden sind. Es wurden nur eine begrenzte Anzahl von Ausführungstests auf dem MONADS-PC durchgeführt, da diese sehr zeitraubend sind.

Der Compiler ist sicherlich nicht fehlerfrei, denn in GCC sind noch einige bekannte (und wohl auch noch einige unbekannte) Fehler, die nur äußerst selten auftreten. Meist sind sie sehr schwer zu korrigieren, ohne das Risiko einzugehen, noch mehr und schwerwiegendere Fehler einzuführen.

Die Geschwindigkeit des Compilers ist sehr hoch, sie erlaubt eine Übersetzung von großen Projekten in akzeptabler Zeit. Wenn aufwendige Optimierungen verlangt werden, dann ist die Übersetzung teilweise sehr langsam und speicheraufwendig. Das ist nicht spezifisch für die Portierung auf den MONADS-PC, sondern liegt an an den Laufzeit- und Speicherbedarfseigenschaften der Optimierungen.

Die Übersetzung großer Projekte ist aus einem anderen Grund problematisch: der gewählte Ansatz des textuellen Bindens bedingt teilweise riesige Eingabedateien für den existierenden Assembler. Dieser ist nicht für solche Größenordnungen von Eingabedateien geeignet, weil er bei der Übersetzung in Maschinencode sehr viel Hauptspeicher benötigt. Dies läßt sich mit der Selbstübersetzung des C-Compilers illustrieren: bei `cc1`, dem eigentlichen C-Compiler, kommen mehr als 12 MByte Quelltext aus dem Linker, der damit noch gut zurechtkommt. Dem MONADS-Assembler reichten die 64 MByte des Rechners, auf dem die Portierung durchgeführt wurde, bei weitem nicht aus. Beim C++-Compiler, dem größten bisher übersetzten Programm (mehr als 16 MByte Quelltext), tritt Thrashing auf, da zu Spitzenzeiten mehr als 120 MByte Speicher verwendet werden.

In [11] wurde vermutet, daß eine statische Bindung zu sehr großen Programmen führen würde. Dies hat sich nur teilweise als richtig erwiesen, die Abschätzungen der erwarteten Codegröße bei statisch gebundenen Programmen sind deutlich zu hoch gewesen. Der größte Teil des von der C-Bibliothek eingebundenen Codes ist die Implementierung von `printf` und verwandten Funktionen. Dies liegt

daran, daß zur Bindezeit nicht entschieden werden kann, ob alle Ein- und Ausgabeunterfunktionen benötigt werden. Der Code wird dadurch recht groß, der Effekt könnte aber gerade hier sehr leicht durch Auslagerung in ein externes, dynamisch gebundenes Bibliotheksmodul verringert werden.

Der GNU C-Compiler erwies sich als sehr leicht anpaßbar an die spezifischen Probleme des MONADS-PC. Die wenigen, meist leicht behebbaren Probleme des Compilers mit den Eigenschaften des MONADS-PC lagen darin begründet, daß jede Architektur ihre kleinen Unterschiede hat, die an wenig getesteten Stellen des Compilers Fehler aufdeckt. Abgesehen davon ist der größte „Fehler“ des GNU C-Compilers die teilweise mangelhafte Dokumentierung seiner Abläufe und Eigenschaften. Oftmals konnte nur durch Untersuchung schon existierender Portierungen und des Compiler-Quelltexts ein Ansatz für die Implementierung der kritischen Befehle gefunden werden. Die Untersuchung des Compiler-Quelltexts ist durch die sehr saubere Implementierung und ausreichende Kommentierung nach einiger Einarbeitungszeit mit vertretbarem Aufwand möglich. Die ursprüngliche Hoffnung, Teile der früheren GCC-Portierung wiederverwenden zu können, erwies sich als falsch. Die Änderungen am Befehlssatz und die Auswirkungen des dynamischen Bindens in der früheren Portierung auf praktisch jedes Befehlsmuster erforderten zu viele Modifikationen, so daß ganz von vorne begonnen werden mußte. Auch die übrigen Teile der Maschinenbeschreibung konnten nicht übernommen werden, weil sich viel zwischen GCC Version 1.36 und 2.8 geändert hat.

Die Mikroprogrammierbarkeit des MONADS-PC war für die Portierung des Compilers sehr nützlich. Sie erlaubte die Erstellung einer neuen Adressierungsart, die nur durch großen Aufwand mit existierenden Maschinenbefehlen realisierbar gewesen wäre. Dieser Aufwand hätte die Vorteile der Kombination von 32 Bit-Zeigern und Segmentierung zunichte gemacht. Weitere Vorteile ergaben sich aus der schon seit längerem in Entwicklung befindlichen Neuimplementierung des Befehlssatzes, bei der viele sinnlose Einschränkungen wegfielen und dadurch effizienterer Code erzeugt werden konnte. Der neue Mikrocode hatte anfangs zwar einige Fehler, diese waren aber innerhalb kurzer Zeit behoben.

Die sehr strikte Einhaltung der für UNIX-Compiler üblichen Konventionen ermöglicht eine sehr leichte Portierung existierender C-Programme. In den allermeisten Fällen ist die Weiterbenutzung der vom Autor des C-Programms parallel erstellten `Makefile`-Datei möglich. Der Endanwender kann jetzt problemlos ein C-Programm für den MONADS-PC entwickeln oder portieren. Lediglich die C-Bibliothek muß noch in folgenden Arbeiten vervollständigt werden.

Anhang A

Kurzbeschreibung des C-Compilers

Der GNU C-Compiler ist flexibel für viele Plattformen konfigurierbar und mit umfangreichen Einstellungen versehen. Um diese Vielfalt übersichtlicher zu gestalten und um die genaue bei der Portierung eingesetzte Konfigurierung zu dokumentieren, dienen die folgenden Abschnitte. Dies kann einen Blick in das GCC-Handbuch nur soweit ersetzen, wie nicht von der vorgestellten Konfigurierung abgewichen wird. Eine C-Sprachbeschreibung ist in vielen Büchern zu finden und wird deshalb ebenfalls ausgeklammert.

A.1 Konfigurierung

Die Generierung des GNU C-Compilers wird von mehreren `Makefile`-Dateien gesteuert, die je nach benötigter Konfigurierung zusammengestellt werden. Die Zusammenstellung erfolgt durch ein Konfigurierungsskript, `configure`. Dieses Skript besitzt als wichtigste Einstellungsmöglichkeit die Angabe der Zielplattform[10], der Angabe der Plattform, auf dem der erzeugte Compiler laufen soll, und die Angabe der Plattform, auf der die Übersetzung des Compilers erfolgen soll. Diese Dreiteilung erlaubt es, auf einem einzigen Rechner beliebig konfigurierte Compiler zu erzeugen. Je nach der gewünschten Kombination müssen zuerst entsprechende Cross-Compiler erzeugt werden. Dies erfolgt über den gleichen Mechanismus.

Die Angabe der Übersetzungsplattform `--build=...` ist dabei nur in Ausnahmefällen notwendig, denn das Konfigurierungsskript kann auf den allermeisten Plattformen diese Information selbsttätig ermitteln.

Hingegen ist die Spezifikation der gewünschten Zielplattform `--target=...` für Cross-Compiler immer erforderlich, um die Standardkonfigurierung außer Kraft zu setzen. Ohne die Angabe würde ein Compiler erzeugt, der auf dem aktuellen

[10] Die Zielplattform ist die Architektur und das Betriebssystem, für die der erzeugte Compiler Code generieren soll.

z.B. `m68k-apple-aux` oder `alpha-dec-openvms5.5`

z.B. `powerpc-apple-linux-gnu`

Abbildung A.1: Konfigurationsnamen

System übersetzt wird, auf diesem System läuft und Code dafür erzeugt. Wenn ein Compiler zu erzeugen ist, der auf einem anderen System ablaufen soll, dann muß auch der dritte Parameter, `--host=...` entsprechend eingestellt werden.

Die Angaben der Plattformen sind ihrerseits in drei Bestandteile zerlegbar (siehe Bild A.1) und enthalten den Architekturnamen, den Rechnerhersteller und das Betriebssystem. In einigen Fällen ist die Angabe des Betriebssystems noch einmal in zwei Teile zerlegt, was insgesamt vier Teile ergibt. Die Aufteilung des Betriebssystems soll hier nicht die Benutzung zweier Betriebssysteme bedeuten, sondern herstellerunabhängige Gemeinsamkeiten kennzeichnen.

Bei der Konfigurierung sollte gleich das Installationsverzeichnis angegeben werden, denn im Compiler werden diese Pfade absolut abgespeichert. Dies geschieht mit dem Parameter `--prefix=...`, mit dem das Wurzelverzeichnis des Installationsbaums angegeben werden kann. Alle ausführbaren Programme werden im Unterverzeichnis `bin`, die Bibliotheken, die Bestandteile des Compilers und die Dokumentation werden in den Verzeichnissen `lib`, `info` und `man` abgelegt. Bei Bedarf können die voreingestellten Pfade auch später durch die Environmentvariable `GCC_EXEC_PREFIX` angepaßt werden.

Der Konfigurationsname des MONADS-PC mit dem derzeitigen Betriebssystem wurde auf `mpc-unknown-mpcos` festgelegt. Das Kommando (angenommen sei, daß `configure` im aktuellen Verzeichnis ist), um einen Cross-Compiler vom aktuellen System auf den MONADS-PC zu erhalten, ist:

```
./configure --target=mpc-unknown-mpcos --prefix=/monads
```

Das `configure`-Skript prüft, ob diese Konfigurierung überhaupt unterstützt wird und testet, ob bei dieser Konfigurierung fehlende oder nicht funktionierende Bibliotheksfunktionen nachgebildet werden müssen. Am Ende des Vorgangs werden aus den verschiedenen Teilen die `Makefile`-Datei und einige andere, konfigurierungsabhängige Dateien erstellt.

Es ist auch möglich, `configure` aus einem anderen Verzeichnis heraus aufzurufen, das dann die Objektdateien und andere bei der Erzeugung und Übersetzung des Compilers anfallende Dateien aufnimmt. So bleibt das Quelltextverzeichnis übersichtlich und es ist möglich, mehrere Konfigurationen gleichzeitig auf einem Rechner zu übersetzen. In diesem Fall muß natürlich der Pfad des `configure`-Programms entsprechend angegeben werden.

Bei der Diplomarbeit wurde immer der vollständige Name der Konfiguration verwendet. Es ist aber ebenso möglich, Abkürzungen zu verwenden wie `mpc` oder `mpc-mpcos`. In diesem Fall ändern sich allerdings auch die Namen der installierten Programme entsprechend, genaueres hierzu siehe Abschnitt A.2.

Für die Übersetzung des Compilers gibt es mehrere Varianten, die aber alle nur benutzerfreundlichere Varianten meist mehrerer `make`-Aufrufe mit entsprechenden Definitionen sind. Diese sind ausschließlich auf die Erzeugung von Native-Compilern beschränkt. Ein oft verwendetes Beispiel dieser benutzerfreundlichen Aufrufe ist `make bootstrap`, das zuerst mit dem vorhandenen C-Compiler ohne Optimierung übersetzt, dann zweimal mit dem jeweils im vorigen Durchgang erzeugten Compiler, diesmal mit eingeschalteter Optimierung. Insgesamt werden so drei Compiler erzeugt, wobei die letzten zwei identisch sein sollten. Zur Überprüfung dieser Gleichheit, die einen Hinweis auf die Korrektheit des Compilers gibt, dient `make compare`.

Diese benutzerfreundlichen Methoden sind für Cross-Compiler (um die geht es hier ja hauptsächlich) nicht anwendbar, da bereits der zweite erzeugte Compiler auf dem Zielsystem laufen würde. Der Cross-Compiler für den MONADS-PC wird daher mit folgendem Kommando übersetzt:

```
make CFLAGS="-O2 -g"
```

Dieses Kommando übersetzt alle drei Compiler (C, C++ und Objective C) in einem Durchgang. Wenn dies nicht erforderlich ist, kann durch Anhängen von `LANGUAGES="c c++"` z.B. der Objective C-Compiler weggelassen werden.

Auf dem Testsystem, einer Sun Ultra-2, dauert die Übersetzung aller drei Compiler knapp eine Viertelstunde[11]. Das System arbeitet dabei ausschließlich auf einer lokalen 9 GByte-Festplatte, um zeitraubende Netzwerkzugriffe zu vermeiden.

A.2 Installation

Die Installation ist sehr einfach, vorausgesetzt die Konfigurierung wurde korrekt vorgenommen. Der folgende Aufruf installiert den Compiler und alle notwendigen Dateien am dafür vorgesehenen Platz:

```
make install
```

Es sollte noch überprüft werden, ob das `bin`-Unterverzeichnis des Installationsverzeichnisses schon im Programmsuchpfad enthalten ist. Da Cross-Compiler niemals die Standardcompiler eines Systems sind, muß der neu installierte Cross-Compiler für den MONADS-PC entweder mit `mpc-unknown-mpcos-gcc` oder

[11] Die Zeitangabe ist ohne Benutzung des zweiten Prozessors zu verstehen. Der zweite Prozessor kann durch Angabe der `make`-Option `-j2` ebenfalls für parallele Übersetzung benutzt werden.

alternativ `gcc -b mpc-unknown-mpcos` aufgerufen werden. Dies könnte bei teilweiser Installation von Hand umgangen werden, indem das meist schon vorhandene `gcc`-Programm durch das neu erzeugte ersetzt wird.

Ein anderer möglicher Weg ist die Installation eines kürzeren symbolischen Verweises auf den installierten Compiler, z.B. unter dem Namen `mpc-gcc`. Die oben beschriebene Möglichkeit, eine Abkürzung bei den Konfigurationsnamen anzugeben, macht diese Anpassung überflüssig, wenn als Konfigurationsname `mpc` angegeben wird.

A.3 Benutzung

Die Benutzung der MONADS-PC-Portierung des GNU C-Compilers ist identisch zu allen anderen GNU C-Portierungen. Es sollte ohne größere Probleme möglich sein, existierende Programme einfach zu übersetzten, meistens kann sogar `Makefile` benutzt werden, das den Programmen oft beigefügt ist.

Die einzige zusätzliche Compileroption ist `-mlineno` (und natürlich analog auch `-mno-lineno`). Diese Option schaltet die Erzeugung von Zeilennummerninformationen im Assemblercode ein bzw. aus. Die Zeilennummerninformationen werden mit `setl`-Befehlen realisiert. Die Option ist nur wirksam, wenn gleichzeitig der Compiler angewiesen wurde, Debugging-Informationen zu erzeugen. Das Dateiformat des MONADS-PC sieht zwar keine Debugger-Unterstützung vor, die Angabe von Zeilennummern ist aber doch bei der Fehlersuche sehr hilfreich. Weil zusätzliche Befehle erzeugt werden, führt dies leider zur Vergrößerung des Codes und zu Geschwindigkeitseinbußen.

Beispiele für die Erzeugung von Objektdateien (die Aufrufe mit automatischem Aufruf des Linkers sind analog):

```
mpc-gcc -O -Wall -c -o eval.o eval.c
mpc-gcc -g -mlineno -O1 -c suicide.c
```

Das erste Beispiel übersetzt den Quelltext `eval.c` mit den Standardoptimierungen (`-O`) und allen für normale Programme sinnvollen Warnungen (`-Wall`). Die Option `-c` gibt an, daß nur eine Objektdatei erzeugt werden soll. Der Name der Objektdatei ist mit dem Parameter `-o eval.o` explizit angegeben. Das zweite Beispiel illustriert die Erzeugung von Zeilennummerninformationen.

Es ist sogar bei optimierender Übersetzung möglich, Zeilennummern in den Assemblerquelltext einzufügen. Es muß aber immer mit Umstellungen im Code gerechnet werden. Dies führt dazu, daß bestimmte Zeilen gar nicht bzw. mehrmals an verschiedenen Stellen in der Ausgabedatei vorkommen.

Bei der Erstellung von externem Assemblercode ist darauf zu achten, daß der Compiler allen vom Benutzer angegebenen Symbolen einen Unterstrich _ voranstellt. Die Funktion `printf` ist daher mit dem Namen `_printf` anzusprechen.

Anhang B

Handbuch der Hilfsprogramme

Für eine vollständige Entwicklungsumgebung ist die Portierung des Compilers die Basis. Ohne Assembler, Linker und Bibliotheksverwaltung (zusammen hier als „Hilfsprogramme“ bezeichnet) ist der Compiler nicht einsetzbar. Im folgenden sollen die verschiedenen Programme etwas näher beschrieben werden, um eine spätere Erweiterung oder Änderung zu erleichtern. Die genaue Arbeitsweise der Optionen wird ausführlich erklärt.

B.1 Konfigurierung

Die Konfigurierung erfolgt ähnlich wie beim C-Compiler mit dem `configure`-Kommando. Der Aufruf unterscheidet sich aber leicht von der Konfigurierung des GNU C-Compilers, denn es ist nur bei Compilern sinnvoll, drei Plattformen anzugeben. Bei den Hilfsprogrammen kann man deshalb nur das System, auf dem die Hilfsprogramme übersetzt werden sollen und das Zielsystem angeben. Es wäre im Fall der Hilfsprogramme für den MONADS-PC zwar überflüssig, das Zielsystem anzugeben. Aus Symmetriegründen wurde die Angabe beibehalten. Dies macht die Konfigurierung identisch mit den GNU Binutils, die für viele Plattformen die Hilfsprogramme bereithalten.

Die GNU Binutils waren für die Portierung auf den MONADS-PC ungeeignet, weil sie keine textuelle Bindung unterstützen. Als klassische Assembler- und Linkerimplementierungen werden nur globale Symbole in den Objektdateien verarbeitet. Der Linker für den MONADS-PC muß sich hauptsächlich um die Vermeidung von Kollisionen lokaler Symbole kümmern, paßt also nicht in dieses Schema.

Die Konfigurierung für den MONADS-PC erfolgt mit folgendem Kommando (die Angabe des Installationsverzeichnisses sollte angepaßt werden):

```
./configure --target=mpc-unknown-mpcos --prefix=/monads
```

Die Konfigurierung in einem anderen Verzeichnis für Objektdateien ist selbstverständlich möglich. Der Quelltext wird mit dem Kommando `make` übersetzt.

B.2 Installation

Die Hilfsprogramme werden wie gewohnt mit `make install` installiert. Es ist nicht möglich, später ein anderes Installationsverzeichnis anzugeben. Wenn die Programme in einem anderen Unterverzeichnis installiert werden sollen, ist eine Neuübersetzung zwingend erforderlich.

Die Programme werden an zwei Stellen installiert, um zwei komplementären Anforderungen gerecht zu werden. Ein Satz von Programmen wird im `bin`-Verzeichnis installiert. Um Kollisionen mit den schon vorhandenen Hilfsprogrammen für das auf dem Rechner installierte Betriebssystem zu vermeiden, wird jedem Programmnamen der Zusatz `mpc-unknown-mpcos-` vorangestellt. Alle Programme sind über diesen Namen direkt im Suchpfad ansprechbar. Das ermöglicht, die Hilfsprogramme für mehrere Zielsysteme gleichzeitig installiert und verfügbar zu haben. So können leicht Programme für verschiedene Plattformen erzeugt werden.

Die von GCC gestellte Anforderung ist dagegen die Verwendung der Standardnamen wie `as` und `ld`. Deswegen werden die Programme zusätzlich im Unterverzeichnis `mpc-unknown-mpcos/bin` unter ihrem normalen Namen abgelegt, um bei Aufgaben, die ausschließlich mit bestimmten Zielplattformen arbeiten, auch die kurzen Namen verwenden zu können. GCC durchsucht zuerst dieses Verzeichnis, bevor der Pfad durchsucht wird.

B.3 Dateiformate

Zuerst sollen das Datenformat der Objektdateien vorgestellt werden, auf denen die verschiedenen Hilfsprogramme arbeiten. Sie sind essentiell wichtig für das Verständnis der Arbeitsweise des Assemblers und des Linkers. Die Programme zur Bibliotheksverwaltung greifen ebenfalls auf Teile der Objektdateien zu und verarbeiten also neben dem Datenformat der Bibliotheksdateien auch Objektdateien.

Die Beschreibung der C-Klassenbeschreibung erfolgt im Abschnitt über den Stub-Generator, da dieses Format spezifisch auf ein einzelnes Programm zugeschnitten ist und bei Änderungen am Stub-Generator wahrscheinlich ebenfalls angepaßt wird. Das Format der C-Klassenbeschreibung ist bisher für den Benutzer sichtbar, da C-Klassenbeschreibungen noch von Hand geschrieben werden müssen.

B.3.1 Objektdateien

Für die Objektdateien wurden aus Effizienzgründen nicht direkt die Assemblerdateien benutzt, obwohl dies auch möglich gewesen wäre. Der neu implementierte Assembler wäre dann vollständig überflüssig gewesen.

```
:MPC object file 1.0
:source "hello.c"
:begin symbol
_printf U       0
_main   T       0
LLC0    c       0
:end symbol
:begin section
const   3
text    14
:end section
:end header
        align
LLC0:
        byte    "Hello world!",10,0
_main:
; prologue, locals: 0, outgoing: 4
        addi    i0,#12
        stax    -12(cx)[i0]
; end prologue
        lda     #LLC0
        sta     -4(cx)[i0]
        jsub    #_printf
        ldax    #$0
; epilogue
        lda     -12(cx)[i0]
        subi    i0,#12
        jump    a
; end epilogue
```

Abbildung B.1: Aufbau einer Objektdatei

Die gewählte Lösung einer aufbereiteten Objektdatei (siehe auch Bild B.1) bietet aber den Vorteil, daß der Linker und die Bibliotheksverwaltungsprogramme von einer leicht verarbeitbaren Datei profitieren. Speziell die Erstellung des Symbolindexes wird deutlich einfacher, siehe hierzu Abschnitt B.3.2. Ein weiterer Vorteil ist die frühe, allerdings recht grobe Syntaxprüfung, die half, viele Compilerfehler schnell zu erkennen.

Objektdateien bestehen aus fünf Teilen, die immer in der folgenden Reihenfolge vorkommen:

Kennung: Die Kennzeichnung von Objektdateien vermeidet die Bearbeitung beliebiger Dateien durch den Linker. Bei den Bibliotheksverwaltungsprogrammen dient sie zur Entscheidung, ob eine Datei eine Objektdatei ist, deren Symbole in den Index aufgenommen werden müssen.

Quelltextname: Bisher wird dieses Feld nur zu Debuggingzwecken benutzt, um die Zuordnung zum C-Quelltext herzustellen.

Symboltabelle: Hier wird jedes verwendete Symbol aufgeführt und angegeben, ob es lokal (Kleinbuchstaben) oder global ist. Die hierbei erlaubten Buchstaben sind:

- **B:** Symbol ist in den uninitialisierten Daten,
- **C:** Symbol ist in den Konstanten,
- **D:** Symbol ist in den initialisierten Daten,
- **T:** Symbol ist im Codebereich oder
- **U:** Symbol ist bisher unbekannt.

Jedes Symbol besitzt darüberhinaus eine Größe, die aber bisher nur bei Symbolen uninitialisierter Daten benutzt wird. Für diese Symbole werden erst vom Linker Speicherbereiche angelegt, daher muß diese Größe in den Objektdateien weitergegeben werden.

Verzeichnis der Abschnitte: Eine Objektdatei enthält in der Regel sowohl Daten, Konstanten und Code, die vom Linker in die richtige Reihenfolge für den MONADS-Assembler gebracht werden müssen. Die Objektdatei wird deswegen in mehrere Abschnitte unterteilt, die jeweils einer der Gruppen entsprechen. Für jeden Abschnitt wird die Zeilenzahl abgelegt, damit die Abschnitte effizient zugreifbar sind.

Assemblercode: Der Code aller Abschnitte wird hier in der Reihenfolge abgelegt, in der die Abschnitte im Verzeichnis eingetragen sind.

B.3.2 Bibliotheken

Die Bibliotheken werden im Standardformat abgelegt, das auf UNIX-Systemen üblich ist. Das Format ist zeilenorientiert, enthält die einzelnen Dateien aber als binäre Blöcke. Am Anfang ist die Kennung `!<arch>` und ein Zeilenvorschub abgelegt (siehe Bild B.2), damit nicht versehentlich andere Dateien bearbeitet werden. Der zeilenorientierte Aufbau kann auch ignoriert werden und die Datei als Binärdatei behandelt werden. Der Dateianfang läßt sich dann einfach als Sequenz von 8 Bytes behandeln.

```
!<arch>
__.SYMDEF       878049119   0     0     0       376       '
<376 Bytes Index>
printf.o        878044540   944   880   100600  1013      '
<1013 Bytes der Datei printf.o>

#1/16           878044651   944   880   100600  1328      '
argz-stringify.o<1312 Bytes der Datei argz-stringify.o>
```

Abbildung B.2: Aufbau einer Bibliothek

Die Dateien werden von einer Kopfzeile beschrieben (Länge, Datum der letzten Modifikation, Benutzer- und Gruppennummer, Zugriffsrechte), die vor den eigentlichen Daten abgelegt wird. In der Kopfzeile ist nur für maximal 15 Zeichen lange Dateinamen Platz. Längere Namen müssen auf andere Weise abgelegt werden. Die Dateinamen werden durch Leerzeichen abgeschlossen. Deswegen müssen auch diese Dateinamen speziell abgelegt werden, um die Leerzeichen zu erhalten. Für diese Speicherung nicht direkt benutzbarer Namen gibt es zwei Varianten. Die eine verwendet ein Verzeichnis aller vorkommenden speziellen Namen in einer am Dateianfang abgelegten Liste. Dieses Verzeichnis wird selbst wie eine normale Datei in der Bibliothek gespeichert. Die andere Variante legt den vollständigen Namen direkt nach der Kopfzeile ab. Der für den Dateinamen benötigte Platz wird in den eingetragenen Namen eincodiert (es wird die Zeichenkette `#1/` und die Länge der Datei verwendet) und der benötigte Platz der Datei zugeschlagen.

Es gibt noch weitere Varianten, hierbei sei auf die Dokumentierung der Implementierung in den GNU Binutils verwiesen. Die Datei `bfd/archive.c` in der GNU Binutils-Distribution Version 2.8 dokumentiert viele weitere Varianten, die es auf verschiedenen Systemen gibt.

Bibliotheken können einen Symbolindex enthalten, der für jedes globale Symbol angibt, in welcher Datei in der Bibliothek es definiert wird. Dieser Index beschleunigt den Bindevorgang, da nicht mehrmals über die Objektdateien in einer Bibliothek iteriert werden muß, um alle Referenzen aufzulösen. Der Index wird in einer nicht sichtbaren Datei[12] in der Bibliothek abgelegt, die immer die erste Datei in der Bibliothek ist. Der Symbolindex ist plattformunabhängig spezifiziert, er kann deswegen auf allen Systemen benutzt werden. Diese Eigenschaft wird meist nicht benötigt, denn dem Bibliotheksverwaltungsprogramm muß das Format der Objektdateien bekannt sein, um den Index aufzubauen oder zu benutzen.

Die Kopfzeilen werden immer auf gerade Dateipositionen in der Bibliothek ausgerichtet. Gegebenenfalls wird ein Zeilenvorschubzeichen `\n` nach einer Datei ungerader Länge eingefügt.

Die Variante, die implementiert wurde, entspricht dem Bibliotheksformat von BSD 4.4. Bei diesem Format werden die Dateinamen nach der Kopfzeile abgelegt. Ein um die eigentlichen Daten gekürztes Beispiel ist im Bild B.2 zu sehen. Bei diesem Format ist die Kompatibilität mit den GNU Binutils leider nur teilweise gegeben, denn diese können das BSD 4.4-Format nur lesen, aber nicht schreiben. Dies muß unbedingt beachtet werden, denn der Linker und die Bibliotheksverwaltungsprogramme für den MONADS-PC können die anderen Varianten nicht verarbeiten.

Ein anfangs häufig aufgetretener Fehler ist die Benutzung des falschen Bibliotheksverwaltungsprogramms. Dieser Fehler führt meist zu Fehlermeldungen des Linkers, wenn Bibliotheken hinzugebunden werden.

[12] Intern wird der Name `__.SYMDEF` verwendet.

B.4 Assembler

Der Assembler übersetzt handgeschriebenen oder von einem Compiler erzeugten Assemblercode in die Darstellung durch Objektdateien. Die Hauptaufgabe des Assemblers ist die Erstellung einer Liste aller definierten Symbole. Zu jedem wird abgespeichert, ob es lokal oder global definiert ist und die grobe Einteilung in Konstanten, initialisierte Daten, uninitialisierte Daten und Code vorgenommen. Dieser Verarbeitungsschritt erzeugt Objektdateien, die für den Linker und die Bibliotheksverwaltungsprogramme sehr leicht verarbeitbar sind.

B.4.1 Aufruf des Assemblers

Der Aufruf des Assemblers kann auf zwei Arten erfolgen. Die Namen der Objektdateien werden automatisch generiert oder vom Benutzer vorgegeben. Wenn der Benutzer den Namen der Objektdatei vorgibt, dann kann nur eine Assemblerdatei übersetzt werden. Sonst sind beliebig viele Assemblerdateien erlaubt.

Typische Aufrufe sehen wie folgt aus:

```
mpc-unknown-mpcos-as move.s weight.s board.s
mpc-unknown-mpcos-as -o ../test/hello.o hello.s
```

Zusätzlich ist es auch möglich, keine Quelldatei anzugeben. In diesem Fall ist die Angabe einer Ausgabedatei zwingend erforderlich. Der Quelltext wird von der Standardeingabe gelesen.

Die Assemblerquelltextdatei des Beispiels, das bei der Beschreibung von Objektdateien benutzt wurde, ist in Bild B.3 zu sehen. Der vom GNU C-Compiler erzeugte Code wurde um einige Kommentare gekürzt.

Alle verfügbaren Optionen des Assemblers sind:

`-n` Schaltet die Warnungen ab. Bisher gibt es keine Situationen, in denen Warnungen erzeugt werden, die Option dient nur der zukünftigen Verwendung und der Kompatibilität zu GNU as.

`-o` Spezifiziert einen Namen für die erzeugte Objektdatei. Wenn diese Option angegeben wird, kann nur ein einziger Assemblerquelltext bearbeitet werden. Ohne diese Option können beliebig viele Assemblerquelltexte angegeben werden. Der Name der Objektdatei wird dann durch Entfernen der Dateierweiterung und Hinzufügen von `.o` bestimmt.

`-V` Gibt die Version des Assemblers aus.

`-d` Schaltet den Debugging-Modus des Parsers ein. Die gerade bearbeiteten Tokens und der Stackzustand werden ausgegeben.

Der Assembler gibt eine kurze Zusammenfassung seiner Benutzung und die erlaubten Optionen aus, wenn er mit dem Parameter `-V` aufgerufen wird oder die Kommandozeile Fehler enthält.

```
        .file   "hello.c"

.section        .text

.section        .const
        align
LLC0:
        byte    "Hello world!",10,0

.section        .text

        .global _main
_main:
; prologue, locals: 0, outgoing: 4
        addi    i0,#12
        stax    -12(cx)[i0]
; end prologue
        lda     #LLC0
        sta     -4(cx)[i0]
        jsub    #_printf
        ldax    #$0
; epilogue
        lda     -12(cx)[i0]
        subi    i0,#12
        jump    a
; end epilogue
```

Abbildung B.3: Assemblerquelltext für `hello.c`

B.4.2 Neue Pseudobefehle

Für die Implementierung der zusätzlichen Funktionen des Assemblers sind einige neue Pseudobefehle notwendig, die der MONADS-Assembler nicht unterstützt. Alle zusätzlichen Pseudobefehle sind an dem vorangestellten Punkt erkennbar. Punkte sind in den Eingabedateien des MONADS-Assemblers nicht erlaubt. Es ist daher einfach, fundamentale Fehler bei der Erzeugung des endgültigen gebundenen Assemblercodes zu finden.

Die folgenden Pseudobefehle wurden hinzugefügt:

`.global`: Deklariert ein Symbol als globales Symbol.

`.common`: Definiert einen globalen Speicherbereich, der mit anderen so definierten Bereichen gleichen Namens beim Binden kombiniert wird.

`.reserv`: Definiert analog einen dateilokalen Bereich, der eine uninitialisierte Variable enthält.

`.section`: Beginnt einen neuen Abschnitt des Assemblercodes oder setzt einen Abschnitt fort, z.B. Daten, Konstanten oder Code.

`.file`: Gibt den Namen des Quelltexts an, der in der Objektdatei abgelegt wird.

`.text`: Dient nur der Kompatibilität mit GNU as. Jede Verwendung von `.text` ist äquivalent zu `.section .text`.

Die Erweiterungen beschränken sich hauptsächlich auf die Einführung dateilokaler und globaler Symbole, die getrennt behandelt werden müssen. Die Unterstützung beliebiger Abschnitte erlaubt, den Assemblerquelltext in beliebiger Reihenfolge anzugeben. Der Assembler sortiert die verschiedenen Teile eines Abschnitts zusammen.

B.5 Linker

Die Aufgabe des Linkers ist es, die verschiedenen Objektdateien und Bibliotheken zusammenzufügen, aus denen ein Programm besteht. Es wird dafür gesorgt, daß ein Symbol eindeutig zugeordnet wird, auch wenn die verschiedenen Objektdateien unterschiedliche Definitionen enthalten, die den gleichen Namen benutzen.

Der Linker kann mehrere Objektdateien und Bibliotheken binden und das Ergebnis in eine Objektdatei schreiben. Diese Objektdatei enthält üblicherweise deutlich weniger undefinierte Symbole, da eine Kombination mehrerer Objektdateien die Zahl der undefinierten Symbole meist verringert.

Ein Linkeraufruf erzeugt aber sehr häufig gleich ein vollständiges Programm. Programme enthalten keine undefinierten Referenzen mehr. Im Fall des Linkers für den MONADS-PC ist das Resultat des Bindevorgangs eine Textdatei. Da ein nachträglicher Aufruf des MONADS-Assemblers nicht in das Compiler-Konzept paßt, wird der Aufruf automatisch vom Linker durchgeführt. Das Endergebnis ist also wie auf allen Systemen eine ausführbare Datei.

B.5.1 Aufruf des Linkers

Es wurde schon erwähnt, daß der Linker zwei prinzipiell verschiedene Ausgabeformate unterstützt. Der Aufruf zur Erzeugung einer Objektdatei (Option `-r`) ist kürzer und übersichtlicher als die Erzeugung eines Programms. Der Grund dafür ist aber nur der komplexe Aufbau des C-Anfangscodes.

Die Benutzung von Bibliotheken bedingt ebenfalls meist die Angabe einiger Optionen, um die Suchpfade richtig einzustellen. Es muß dabei strikt zwischen explizit angegebenen Bibliotheken und solchen, die durch Suche im Pfad bestimmt werden, unterschieden werden. Explizit angegebene Bibliotheken (diese tauchen nur im ersten Beispiel auf) werden ohne Berücksichtigung der Suchpfade zugegriffen. Ihr Dateiname muß daher vollständig sein und bei Bedarf das Verzeichnis einschließen. Nur bei Bibliotheken, die mit der Option `-l` angegeben werden, werden der gerade gültige Suchpfad und die Regeln zur Ergänzung des Namens angewandt. Dies ist genauer bei der Beschreibung der Option `-l` erklärt.

Typische Aufrufe für beide Ausgabeformate sind:

```
mpc-unknown-mpcos-ld -r move.o weight.o io-func.a -o engine.o
mpc-unknown-mpcos-ld /monads/mpc-unknown-mpcos/lib/crt1.o     \
       /monads/mpc-unknown-mpcos/lib/crti.o                   \
       /monads/mpc-unknown-mpcos/lib/crtbegin.o               \
       -L/monads/lib/gcc-lib/mpc-unknown-mpcos/testgcc-2.7.90 \
       -L/monads/mpc-unknown-mpcos/lib                        \
       hello.o                                                \
       -lgcc -lc -lgcc                                        \
       /monads/mpc-unknown-mpcos/lib/crtend.o                 \
       /monads/mpc-unknown-mpcos/lib/crtn.o
```

Man sieht deutlich, wie aufwendig es ist, ein C-Programm zu binden. Dies ist der Grund, wieso zum Binden eines Programms der Compiler Driver benutzt werden sollte. Nur in sehr speziellen Fällen sollte der direkte Aufruf des Linkers in Betracht gezogen werden.

-r Weist den Linker an, das Ergebnis des Bindevorgangs als Objektdatei abzulegen. Diese Datei kann noch undefinierte Symbole enthalten, die von anschließenden Bindevorgängen aufgelöst werden.

-m Wird akzeptiert, um mit anderen Linkern kompatibel zu bleiben. Ursprünglich gibt diese Option ein Verzeichnis aller Abschnitte aus. Dies ist nicht implementiert.

-s Bei UNIX-Linkern wird mit dieser Option angegeben, daß die Ausgabedatei keine Debugging-Informationen enthalten soll. Die Option wird ignoriert, weil MONADS-Programme keine Informationen über den Quelltext enthalten, die entfernt werden könnten.

-t Weist den Linker an, keine Warnungen über die unterschiedliche Größe der Deklarationen ein und derselben globalen Variable auszugeben.

-L Fügt ein Verzeichnis dem Suchpfad für Bibliotheken hinzu.

-l Veranlaßt, daß die angegebene Bibliothek zum Programm gebunden wird. Der echte Name der Bibliothek wird durch Voranstellen von `lib` und Anhängen von `.a` ermittelt. Die Bibliothek wird in allen bisher angegebenen Verzeichnissen für Bibliotheken gesucht.

-o Spezifiziert den Namen der Ausgabedatei. Ohne die explizite Angabe dieser Option wird immer der Name `a.out` verwendet, um mit den UNIX-Linkern kompatibel zu sein.

-V Gibt die Version des Linkers aus.

-X Verhindert beim Binden eines vollständigen Programms den automatischen Start des MONADS-Assemblers. Anstelle der Adreßraumdatei wird der Assemblerquelltext, den der MONADS-Assembler erhalten würde, in die Ausgabedatei geschrieben. Nützlich für die Fehlersuche.

-z Gibt aus, in welcher Datei die Definition eines bisher undefinierten Symbols gefunden wurde. Dient zum Debugging des Linkers.

-d Schaltet den Debugging-Modus des Parsers ein. Die gerade bearbeiteten Tokens und der Stackzustand werden ausgegeben.

Der Linker gibt eine kurze Zusammenfassung seiner Benutzung und der erlaubten Optionen aus, wenn keine Datei angegeben wurde oder die Kommandozeile Fehler enthält. Die für spezielle Anwendungen vorbehaltenen Optionen `-X`, `-Z` und `-d` werden nicht ausgegeben.

Die Optionen des Linkers werden oft beim Start des Compilers angegeben. Hierfür muß den Optionen die Option `-Wl,` vorangestellt werden. Dies weist den Compiler Driver an, die entsprechende Option an den Linker weiterzugeben.

B.5.2 Auflösung lokaler Symbole

Der Algorithmus zur Auflösung lokaler Symbole muß die Erzeugung eindeutiger Bezeichner für mehrfach vorkommende lokale Symbole erledigen. Die kollidierenden Symbole werden durchnumeriert. Die generierten Symbole stellen diesem Symbol `N` und die zugewiesene Nummer voran. Kollisionen mit Symbolen, die vom Compiler generiert wurden, sind so ausgeschlossen. Bei handgeschriebenem Assemblercode muß selbst darauf geachtet werden, daß keine Kollisionen auftreten.

Die Verteilung der Nummern ist bisher nicht auf äußerste Effizienz getrimmt. Die Liste aller Symbole mit gleichem Namen (genauer: mit identischem Hashwert) muß durchlaufen werden, um das bisherige Maximum zu finden. Diese Speicherung aller Vorkommen erlaubt es, auch andere Verfahren zur Erzeugung eindeutiger Symbolnamen einzusetzen. Derzeit wird jedem lokalen Symbol die Nummer vorangestellt. Das wäre nicht notwendig, wenn nur ein einziges lokales oder globales Symbol mit diesem Namen im gesamten Programm vorkommt. Alternativ könnten schon in den Eingabeobjektdateien vorhandene Numerierungen lokaler Symbole soweit möglich beibehalten werden.

Beim Bindevorgang kompletter Programme wird zusätzlich jede Symboldefinition konvertiert, so daß die Werte denen entsprechen, die für den gewählten Speicheraufbau benötigt werden. Bis zu dieser Auflösung sind alle Symbole eines Programms über eine normale Label-Definition vereinbart, erkennbar an dem Doppelpunkt. Der MONADS-Assembler weist jedem so definierten Symbol den Offset im aktuellen Segment zu. Um die spätere Speicheradresse zu erhalten, müssen die Anfangsadressen der Segmente bei der Definition berücksichtigt werden. Der Doppelpunkt wird vom Linker durch eine Zuweisung des berechneten Werts ersetzt. Das „`:`“-Zeichen wird durch „`= offset()+...`“ ersetzt, wobei der entsprechende Korrekturwert angehängt wird.

Codesymbole werden etwas anders verarbeitet und erhalten als Wert die Wortadresse im Codesegment. Das muß bei relativen Sprüngen beachtet werden.

B.5.3 Suchalgorithmus bei Bibliotheken

Systemweit installierte Bibliotheken werden bei Linkern üblicherweise mit der Option `-l` und den verkürzten Namen angegeben. Dies erleichtert es, systemunabhängig die richtigen Dateien zu lokalisieren. Der Linker enthält einen Standardsuchpfad, der vom Benutzer beliebig erweiterbar ist. Der Standardsuchpfad beinhaltet bei der Beispielkonfigurierung für den MONADS-PC nur das Verzeichnis `/monads/mpc-unknown-mpcos/lib`.

Mit der Option `-l` angegebene Bibliotheken werden im gesamten derzeit gültigen Suchpfad gesucht. Die Optionen werden dabei strikt in ihrer Reihenfolge bearbeitet. Die Abfolge der `-L`-Optionen und der `-l`-Optionen kann das Ergebnis des Bindevorgangs verändern.

Wenn eine Bibliothek beim Auswerten der Kommandozeilenparameter nicht lokalisiert werden kann, wird eine Warnung ausgegeben und die Angabe ignoriert. Eine fehlende Bibliothek führt daher meist zu vielen Folgefehlern durch nicht definierte Symbole.

Bibliotheken müssen einen Index enthalten, sonst werden sie vom Linker ignoriert. Dieser Fall wird nicht durch eine Warnung angezeigt, da solche Bibliotheken sinnvoll sein können.

B.5.4 Anpassung an den MONADS-PC-Assembler

Der Linker unterstützt zwar die Verwendung beliebiger Abschnittsnamen, es werden aber nur bestimmte, dem Linker bekannte Abschnitte in die Programmdatei geschrieben. Die erlaubten Namen sind `const`, `ctor`, `dtor`, `data`, `udata`, `itext`, `init`, `fini` und `text`. Die ersten drei Abschnitte werden im Segment aller Konstanten abgelegt, die beiden folgenden ergeben zusammen die globalen Variablen des Programms. Die restlichen vier Abschnitte sind für Interfacecode, Initialisierungscode, Beendigungscode und die normalen Codeteile vorgesehen. Für spezielle Anwendungen des C-Startcodes gibt es für die Abschnitte `const`, `data`, `udata` und `text` zusätzliche Abschnitte, die immer vor bzw. nach allen anderen zugehörigen Abschnitten angeordnet werden. Die Abschnittsnamen werden durch Anhängen von `_start` und `_end` gebildet. Bei der Ausgabe von Objektdateien gibt es keine Vorgaben zu den Abschnittsnamen.

Der MONADS-Assembler benötigt vor und nach den verschieden Teilen des Programms einige Schlüsselworte zur Segmentdeklaration. Diese wurden in eine Objektdatei verlagert, die bei jedem Binden vollständiger Programme automatisch dazugebunden wird. Die Namen der Abschnitte werden aus den Abschnitten abgeleitet, die auf `_start` und `_end` enden. An diese Abschnittsnamen wird jeweils noch `_glue` angehängt. Die Abschnittsnamen sind dem Linker bekannt und sollten in Benutzercode niemals vorkommen.

Neben den Segmentdeklarationen für den MONADS-Assembler enthält die Objektdatei auch die Definitionen der Offsets, die für jedes Segment verwendet

werden müssen. So kann durch Ändern einer einzigen Datei der Speicheraufbau umgestellt werden. Änderungen müssen natürlich entsprechend im C-Startcode vorgenommen werden, um die Segmente korrekt anzulegen.

Der Assemblerquelltext dieser Datei ist in `devel/asmglue.s` gespeichert. Diese Datei wird beim Übersetzen der Hilfsprogramme assembliert. Bei der Installation wird die Objektdatei in das Systembibliotheksverzeichnis kopiert.

B.6 Bibliotheksverwaltung

Bibliotheken dienen der komfortablen Verwaltung zusammengehörender Objektdateien. Die im Kontext des C-Compilers am häufigsten benutzte Bibliothek ist die C-Bibliothek. Diese enthält eine große Zahl standardisierter Funktionen für viele Aufgaben. Bibliotheken sind nicht auf Objektdateien beschränkt, sondern für beliebige Dateien benutzbar. Es kommt aber nur selten vor, daß dies eingesetzt wird.

Die Verwaltung von Bibliotheken wird von den Programmen `ar` und `ranlib` vorgenommen. `ar` dient der allgemeinen Verwaltung von Bibliotheken, während `ranlib` nur für die Erstellung des Symbolindexes benötigt wird. `ranlib` existiert eigentlich nur noch aus historischen Gründen, denn `ar` kann mittlerweile ebenfalls mit dem Index umgehen.

B.6.1 Aufruf des `ar`-Programms

Das `ar`-Programm erlaubt komplexe Operationen auf Bibliotheken, darunter die Aktualisierung existierender Bibliotheken durch Ersetzen schon vorhandener Dateien durch eine geänderte Version. Natürlich ist es auch möglich, in einer Bibliothek abgelegte Dateien zu extrahieren.

Das `ar`-Programm hält sich aus historischen Gründen nicht völlig an die übliche Schreibweise der Optionen. Es ist aber ebenso möglich, das Optionskennzeichen „`-`“ voranzustellen. Die Implementierung verlangt aber immer noch, daß alle Optionen aneinandergereiht angegeben werden. Dies muß unbedingt beachtet werden, sonst wird versucht, auf die falschen Dateien zuzugreifen.

Die folgenden Beispiele zeigen einen Überblick über die möglichen Operationen des `ar`-Programms:

```
mpc-unknown-mpcos-ar -cq io-func.a print.o font.o table.o
mpc-unknown-mpcos-ar -r libc.a scanf.o printf.o exit.o
mpc-unknown-mpcos-ar -rsa scanf.o libc.a printf.o
mpc-unknown-mpcos-ar -t libc.a
mpc-unknown-mpcos-ar -x libc.a printf.o
mpc-unknown-mpcos-ar -dv libc.a printf.o scanf.o
```

Das dritte Beispiel ersetzt eine evtl. schon vorhandene Version von `printf.o` in der Bibliothek, plaziert die neue Version aber in jedem Fall hinter `scanf.o`. Die Kommandosyntax ist identisch zu den UNIX-Programmen. Die genaue Beschreibung wurde daher weggelassen.

Die von `ar` unterstützten Optionen sind:

`-d` Löscht alle aufgeführten Dateien in der Bibliothek.

`-m` Verschiebt alle aufgeführten Dateien in der Bibliothek. Die Angabe einer Zielposition mit den Optionen `-a` oder `-b` ist möglich. Die Standardeinstellung ist das Verschieben an das Dateiende. Nicht implementiert.

`-p` Gibt die aufgeführten Dateien auf dem Bildschirm (`stdout`) aus.

`-q` Hängt die angegebenen Dateien am Ende der Bibliothek an, ohne den Index zu aktualisieren.

`-r` Ersetzt die angegebenen Dateien durch die neuen Versionen. Die Angabe einer Zielposition mit den Optionen `-a` oder `-b` ist möglich. Standardeinstellung ist das Einfügen am Dateiende.

`-t` Gibt das Verzeichnis aller Dateien aus, die in der Bibliothek abgelegt sind.

`-x` Speichert eine in der Bibliothek vorhandene Datei als einzelne Datei ab.

`-a` Gibt an, daß das Ziel der durchzuführenden Operation nach der angegebenen Datei in der Bibliothek ist. Die Datei wird nicht vor dem Namen der Bibliothek in der Kommandozeile angegeben.

`-b` Analog für Einfügen vor der angegebenen Datei.

`-i` Aus Kompatibilitätsgründen unterstütztes Synonym für `-b`.

`-c` Gibt keine Warnung aus, wenn eine neue Bibliotheksdatei erstellt wird.

`-o` Gibt an, daß bei der Option `-x` abgespeicherte Dateien die Rechte bekommen sollen, die in der Bibliothek vermerkt sind.

`-s` Fordert `ar` dazu auf, den Symbolindex neu aufzubauen. Äquivalent zu einem getrennten Aufruf von `ranlib`. Diese Option ist nur erforderlich, wenn noch kein Index vorhanden ist oder mit der Option `q` Dateien angehängt wurden. In allen anderen Fällen wird bei jeder Änderung der Index automatisch aktualisiert.

`-u` Modifiziert das Verhalten der Option `-r`, so daß die schon vorhandene Kopie in der Bibliothek nur dann aktualisiert wird, wenn die neue Datei neuer ist. Nicht implementiert.

`-v` Gibt bei jeder Operation ausführlichere Informationen aus.

`-V` Fordert das Programm auf, die Versionsnummer auszugeben.

Der Aufruf ohne Parameter oder mit Fehlern in der Kommandozeile gibt eine kurze Übersicht der möglichen Optionen aus.

B.6.2 Aufruf des `ranlib`-Programms

Die Erstellung des Symbolindexes ist auch über das Programm `ranlib` möglich. Es akzeptiert als Parameter eine einzige Datei, deren Index aktualisiert wird. Der Aufruf von `ranlib` ist äquivalent zum Aufruf `ar s`. Ein Beispiel ist:

```
mpc-unknown-mpcos-ranlib libc.a
```

Das Programm `ranlib` hat deshalb nur eine unterstützte Option:

`-v` Fordert das Programm auf, die Versionsnummer auszugeben.

Das Programm `ranlib` gibt wie üblich eine kurze Zusammenfassung über seine Benutzung aus, wenn er mit der falschen Zahl von Parametern gestartet wurde oder die Kommandozeile sonstige Fehler enthält.

B.7 Stub-Generator

Der Stub-Generator dient dazu, den Konvertierungscode für die unterschiedlichen Aufrufkonventionen des C-Compilers und des MONADS-PC zu erzeugen. So können Schnittstellenfunktionen anderer Module aufgerufen werden, ohne den gesamten Aufruf in Assembler programmieren zu müssen. Die Umsetzung ist wegen der verschiedenen Aufrufkonvention innerhalb von C-Programmen und beim Aufruf von Modulen (siehe Kapitel 5) erforderlich. Die externe Lösung mittels eines Generatorprogramms, das den Konvertierungscode erzeugt, vermeidet die Implementierung einer C-Spracherweiterung. Darüberhinaus wird sichergestellt, daß für jede Schnittstellenkonvertierung nur einmal Code erzeugt wird.

B.7.1 Aufruf des Stub-Generators

Der Aufruf des Stub-Generators erfordert immer zwei Parameter. Der erste Parameter ist der Name der C-Klassendefinition (z.B. `basic_text.h`), der zweite Parameter ist der Basisname für die zu erzeugenden Stub-Funktionen. Der Basisname dient nur zur Bezeichnung der Dateinamen. Die Funktionsnamen werden grundsätzlich von der Klassendefinition abgeleitet (siehe Abschnitt B.7.4).

Ein typischer Aufruf sieht wie folgt aus:

```
mpc-unknown-mpcos-stubgen basic_text.h basic_text
```

Die im aktuellen Verzeichnis vorhandene C-Klassendefinition `basic_text.h` wird als Schnittstellenbeschreibung benutzt, die zur Erzeugung des Konvertierungscodes dient. Jede Funktion wird in einer eigenen Datei abgelegt. In diesem Fall sind das `basic_text.1.c`, `basic_text.2.c`, `basic_text.4.c`, ..., `basic_text.9.c`. Die Numerierung der C-Quelltexte entspricht den Funktionsnummern der Klassenschnittstelle. Die C-Klassendefinitionen dürfen nicht

mit den Pascal-Klassendefinitionen gleichen Namens verwechselt werden, denn ihr Inhalt und die Syntax sind sehr verschieden.

Der Stub-Generator hat zwei Optionen, von denen der Endbenutzer nur eine wirklich benötigt:

-V Fordert das Programm auf, die Versionsnummer auszugeben und

-d Schaltet den Debugging-Modus des Parsers ein, der die gerade bearbeiteten Tokens und der Stackzustand ausgibt.

Der Stub-Generator gibt eine kurze Zusammenfassung seiner Benutzung aus, wenn er ohne Parameter aufgerufen wird oder die Kommandozeile Fehler hat.

B.7.2 Format für C-„Klassendefinitionen“

C-Klassendefinitionen werden für zwei Aufgaben eingesetzt, die verschiedene Anforderungen stellen. Die erste ist die Deklaration von C-Prototypen für sämtliche ansprechbare Schnittstellenfunktionen einer Klasse, die andere ist eine Vorlage für die Generierung des Konvertierungscodes.

Der etwas merkwürdige Aufbau sollte durch einen Blick in die benutzte Headerdatei `mpc-unknown-mpcos/include/mpc/mpc.h` selbsterklärend sein. Weitere Informationen können dem Parser des Stub-Generators entnommen werden.

B.7.3 Erlaubte Datentypen

Die Auswahl der Datentypen ist auf die „einfachen“ Datentypen beschränkt. Es sind nicht nur einfache Datentypen, sondern noch einige mehr. Alle Datentypen können sowohl als Wert- als auch Referenzparameter übergeben werden. Funktionsrückgabewerte werden meist ähnlich behandelt wie Referenzparameter.

Die unterstützten Typen in der Darstellung der MONADS-Pascal-Typen und der zugeordneten C-Datentypen sind:

`integer/int`: Ganze Zahlen, werden auch für die Repräsentierung von booleschen Werten und Zeichen benutzt.

`real/float`: Gleitkommazahlen mit einfacher Genauigkeit.

`string/char *`: Zeichenketten beliebiger Länge, bei der Verwendung in C als mit dem ASCII-Zeichen NUL terminierte Zeichenketten, die mit `malloc` reserviert werden.

`modcap/modcap`: Repräsentierung von Module Capabilities in C-Programmen.

Diese Datentypen entsprechen exakt denen, die auf der Kommandozeilenebene verfügbar sind, so daß alle interaktiv aufrufbaren Schnittstellenfunktionen von C

aus benutzbar sind. Das bedeutet aber auch, daß alle Schnittstellenfunktionen, die Felder und Aggregattypen verlangen, nicht ohne Assemblerprogrammierung ansprechbar sind. Das ist kein großes Problem, denn nahezu alle Schnittstellen der existierenden Module beschränken sich auf die unterstützten Datentypen.

Bei Verwendung des Datentyps `string` ist unbedingt zu beachten, daß alle eventuell möglichen Rückgabewerte vom Konvertierungscode in einen eigenen Bereich kopiert werden, der mit `malloc` reserviert wird. Der Bereich wird jedesmal neu angefordert. Wenn das C-Programm einen Bereich nicht mehr benötigt, sollte er unverzüglich mit `free` wieder freigegeben werden.

B.7.4 Einbindung in Programme

Ein C-Quelltext, der MONADS-Module verwendet, sollte die C-Klassendefinition über `#include` einbinden. Auf diese Weise werden die Prototypen aller ansprechbaren Schnittstellenfunktionen deklariert. Der Compiler kann dann die Parametertypen auf Verträglichkeit prüfen.

Aus Sicht eines C-Programms werden ausschließlich die dazugebundenen Stub-Funktionen benutzt, die C-Aufrufkonventionen verwenden. Die Stub-Funktionen werden mit dem Klassennamen und dem Namen der Schnittstelle bezeichnet, die durch einen Unterstrich _ abgeteilt werden. Diese Benennung ist nicht eindeutig, da sowohl Klassennamen als auch Schnittstellennamen einen Unterstrich enthalten können. Bisher gibt es mit dieser Konvention keine Konflikte. Andere Konventionen verlängern aber die ohnehin schon langen Bezeichner nur noch mehr und beeinträchtigen die Lesbarkeit.

Aufrufe an Stub-Funktionen haben als ersten Parameter immer die Repräsentation eines Module Call-Segments, um die gewünschte Instanz zu kennzeichnen. Diese Repräsentation wird beim `open`-Aufruf angelegt. Hier ist die Variable für das Module Call-Segment als Referenz zu übergeben.

Die übrigen Parameter und der Rückgabewert werden entsprechend der Schnittstellenfunktion deklariert und übergeben.

Anhang C

Bemerkungen zur C-Bibliothek

Die Portierung der GNU C-Bibliothek ist die einzige Aufgabe, deren Basis unabhängig vom Prozessor ist. Bestimmte Teile müssen zwar mit Assemblercode implementiert werden, dies liegt aber nur daran, daß sich die exakte Aufgabe nicht in C selbst ausdrücken läßt.

Es müssen nur wenige, in der Hauptsache betriebssystemabhängige Funktionen implementiert werden, so daß der portable Teil der Bibliothek darauf aufbauen kann.

C.1 Konfigurierung und Installation

Die Konfigurierung erfolgt wiederum sehr ähnlich zu allen anderen Teilen des Entwicklungssystems. Zu beachten ist, daß die C-Bibliothek vom Cross-Compiler für den MONADS-PC übersetzt werden muß. Dies erfordert die Definition der Umgebungsvariable `CC` vor dem Start von `configure`. Die C-Bibliothek bietet viele Einstellungsmöglichkeiten an, auf die hier nicht näher eingegangen werden soll. Der MONADS-PC erfordert die Erstellung einer Bibliothek für statische Bindung. Die ebenfalls mögliche Erzeugung einer Bibliothek für die Untersuchung des Zeitverhaltens der Bibliotheksfunktionen wurde bisher weggelassen, da dieser Funktionskomplex z.B. vollständig implementierte Dateizugriffsfunktionen erfordert.

Die C-Bibliothek wurde mit den folgenden Kommandos konfiguriert:

```
CC="gcc -b mpc-unknown-mpcos"
export CC
./configure mpc-unknown-mpcos --build sparc-sun-solaris2.5.1 \
      --prefix=/monads/mpc-unknown-mpcos                      \
      --disable-shared --disable-profile --enable-static-nss
```

Die Übersetzung erfolgt wie gewohnt mit dem Aufruf von `make`. Die GNU C-Bibliothek verlangt die Verwendung von GNU make, da die komplexen Aufgaben

der `Makefile`-Dateien darauf zugeschnitten sind. Da die Übersetzung der C-Bibliothek sehr zeitintensiv ist, wurde meist auf die Möglichkeit der parallelen Übersetzung mehrerer Programme zurückgegriffen. Da im Rechner, der für die Übersetzung benutzt wurde, zwei Prozessoren vorhanden sind, wurde das Kommando `make PARALLELMFLAGS=-j2` verwendet. Auch mit dieser Option dauert die Übersetzung der C-Bibliothek etwa eine halbe Stunde. Die lange Übersetzungszeit liegt an der sehr großen Zahl der Quelltexte in der C-Bibliothek. Aus dem gleichen Grund sollte die Übersetzung der C-Bibliothek keinesfalls auf einem Netzwerklaufwerk erfolgen.

Die Installation erfolgt wie gewohnt mit `make install`. Die oben benutzte Einstellung legt alle Dateien gleich an der korrekten Stelle ab. Die Dokumentation zur C-Bibliothek wird im Unterverzeichnis `mpc-unknown-mpcos/info` abgelegt. Bei Bedarf kann das Verzeichnis für die Dokumentation mit der Option `--infodir=...` geändert werden.

C.2 Durchgeführte Arbeiten

Bisher wurden nur wenige Teile der C-Bibliothek portiert. Nur die Teile, die für den Test von Programmen unbedingt erforderlich sind, wurden begonnen. Der Startcode des C-Compilers mußte unbedingt vollständig implementiert werden, da sonst überhaupt kein Test des Codes erfolgen konnte. Die Datei `start.S` enthält den Assemblerquelltext. Die Aufgaben des Startcodes wurden schon im Hauptteil dieser Arbeit vorgestellt, dies soll hier nicht wiederholt werden.

Der wichtigste Teil der Portierung ist die begonnene Implementierung der Dateifunktionen. Die Dateien `open.c`, `read.c` und `write.c` enthalten eine stark vereinfachte Implementierung. Bisher können nur die vordefinierten Dateien `stdin`, `stdout` und `stderr` benutzt werden. Sie sind bereits vom Startcode berücksichtigt. Die Implementierung ist bisher nicht sehr effizient. Es sollte mit den in `read.c` und `write.c` nachzulesenden Beispielen aber klar sein, wie andere Module aufgerufen werden können. Ein Aufruf an die Schnittstellenfunktion `readch` der Klasse `basic_text` sieht etwa folgendermaßen aus:

```
status = basic_text_readch(text_mcs, 0, &character);
```

Weil die C-Bibliothek auf die Schnittstelle der Klasse `basic_text` zugreifen muß, sind alle Schnittstellenkonvertierungsfunktionen in der C-Bibliothek selbst vorhanden. Die C-Bibliothek wird praktisch als letzter Teil des Programms angegeben. Sie muß also alle neu auftretenden Symbole selbst definieren.

C.3 Weiterführung der Portierung

Die Fortsetzung der Portierung erfordert die vollständige Implementierung bisher nur angefangener Funktionen und die Neuimplementierung bisher nicht unter-

stützter Funktionen. Die bisher erstellten Dateien befinden sich in den Verzeichnissen `sysdeps/mpcos` und `sysdeps/mpc` des Bibliotheksquelltexts.

Das erste Verzeichnis nimmt die betriebssystemspezifischen Dateien auf, die für jeden Prozessor identisch sind. Bisher gibt es nur einen Prozessortyp. Die Unterscheidung zwischen Betriebssystem und Prozessor impliziert aber, daß im Verzeichnis `sysdeps/mpcos` kein Assemblercode vorkommen sollte. Das Verzeichnis `sysdeps/mpc` enthält die restlichen, prozessorspezifischen Dateien.

Viele der noch zu implementierenden Funktionen besitzen eine leere Definition in `sysdeps/stub`. Diese Implementierungen enthalten jeweils einen Vermerk, daß die Funktion nur eine leere Hülle darstellt. Die Information wird bei der Installation der Bibliothek zusammengestellt. Nach Installation der C-Bibliothek ist ein Verzeichnis der nicht implementierten Funktionen in der Datei `mpc-unknown-mpcos/include/stubs.h` abgelegt.

Die in `mpc/mpc.h` vorgesehenen Funktionen für die Verwaltung von Module Capabilities sind noch nicht implementiert. Dies wird für eine vollständige Implementierung der Dateifunktionen eine vordringliche Aufgabe sein.

C.4 Besonderheiten

Der MONADS-PC unterstützt nur eine einzige Genauigkeitsstufe für Gleitkommazahlen, IEEE-754 single precision. Die C-Bibliothek bietet Funktionen für drei verschiedene Genauigkeiten. Um den Code zu verkleinern, wurden die Funktionen zur Konvertierung zwischen der Textdarstellung und der internen Darstellung (für beide Richtungen) in das Verzeichnis `sysdeps/ieee` verschoben. Das erlaubt, portierungsspezifischen Ersatz für diese Funktionen anzugeben. Die Portierung definiert die nicht unterstützten Funktionen als Synonyme der `float`-Funktionen. Dies entspricht den Definitionen in der Portierung des Compilers, die `double` und `long double` auf den Typ `float` abbilden.

Eine weitere Besonderheit ist in der Datei `sysdeps/mpcos/sysdep.c` vorhanden. Da der Assembler und der Linker keine `weak`-Symbole unterstützen, muß dafür gesorgt werden, daß das Symbol `__gmon_start__` einen definierten Wert bekommt, wenn die C-Bibliothek zu einem Programm gebunden wird, das ohne Erzeugung von Profiling-Code übersetzt wurde. Diese nicht sehr übersichtliche Konstruktion könnte entfallen, wenn Assembler und Linker `weak`-Symbole unterstützen würden.

Der Einsatz der Segmentierung beschränkt die Möglichkeiten der Speicheranforderung etwas. Segmente haben eine feste Größe, die beim Anlegen festgelegt wird. Die Segmente für den Stack und Daten/Heap werden jeweils mit einer Größe von 8 MByte angelegt. Wenn dieser Speicherbereich nicht ausreicht, dann muß der Startcode angepaßt werden. Die niedrige Grenze erleichtert die Fehlersuche in Benutzerprogrammen, da sie eine Obergrenze für den Speicherverbrauch vorgibt.

Anhang D

Listings

D.1 Maschinenbeschreibung für den C-Compiler

Die folgenden Abschnitte sind Listings der verschiedenen Teile der Maschinenbeschreibung. Genauere Informationen über die Syntax bzw. die Funktion der verschiedenen Teile ist in [10] zu finden.

Der Code für den Aufruf der C++-Konstruktoren und -Destruktoren ist hier weggelassen, denn diese sind für diese Diplomarbeit nicht relevant. Sie entsprechen der generischen Implementierung in C bis auf einige kleine Optimierungen.

D.1.1 Maschinenbefehlsbeschreibung `mpc.md`

```
;; GCC machine description for MONADS-PC.
;; Written by Klaus Espenlaub (kespenla@hydra.informatik.uni-ulm.de).

;; This file is is derived from software covered by the terms of the
;; GNU General Public License as published by the Free Software Foundation.
;; Consequently this file is subject to these terms.

;; The original PO technology requires these to be ordered by speed,
;; so that assigner will pick the fastest.

;; See file "rtl.def" for documentation on define_insn, match_*, et. al.

;; Macro #define NOTICE_UPDATE_CC in file mpc.h handles condition code
;; updates for most instructions.

;; Macro REG_CLASS_FROM_LETTER in file mpc.h defines the register
;; constraint letters.

;; Here they are for quick reference:
;;   a -- the register A
;;   b -- the register AX
;;   A -- the register A or AX
;;   D -- any index register (I0-I3)
;;   d -- the register A or any index register (I0-I3)

;; Move instructions, one for each machine mode.
```

```
(define_insn "movqi"
  [(set (match_operand:QI 0 "general_operand" "=a,D,m,m,ab,D")
        (match_operand:QI 1 "general_operand" "m,m,a,D,ir,ir"))]
  ""
  "@
  ldb%0\\t%1
  ldbi\\t%0,%1
  stb%1\\t%0
  stbi\\t%1,%0
  ld%0\\t%1
  ldi\\t%0,%1")

; Handle the reload case (currently only MEM <-> AX)

(define_expand "reload_inqi"
  [(parallel [(match_operand:QI 0 "register_operand" "=b")
              (match_operand:QI 1 "reload_memory_operand" "m")
              (match_operand:QI 2 "register_operand" "=&d")])]
  ""
  "
{
  emit_insn (gen_movqi (operands[2], operands[1]));
  emit_insn (gen_movqi (operands[0], operands[2]));
  DONE;
}")

(define_expand "reload_outqi"
  [(parallel [(match_operand:QI 0 "reload_memory_operand" "=m")
              (match_operand:QI 1 "register_operand" "b")
              (match_operand:QI 2 "register_operand" "=&d")])]
  ""
  "
{
  emit_insn (gen_movqi (operands[2], operands[1]));
  emit_insn (gen_movqi (operands[0], operands[2]));
  DONE;
}")

; GCC insists on having all mov* patterns up to the largest data size supported
; by the machine.  There seems to be no reliable way to avoid HI moves to be
; generated at all.

(define_expand "movhi"
  [(set (match_operand:HI 0 "general_operand" "")
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "
{
  if (GET_CODE (operands[0]) == MEM) {
    operands[1] = force_reg (HImode, operands[1]);
  }

  if (GET_CODE (operands[0]) == MEM) {
    mpc_reload_outhi (operands[0], operands[1], gen_reg_rtx (SImode));
    DONE;
  } else if (GET_CODE (operands[1]) == MEM) {
    mpc_reload_inhi (operands[0], operands[1], gen_reg_rtx (SImode));
    DONE;
  }
}")

(define_insn ""
  [(set (match_operand:HI 0 "register_operand" "=A,D")
        (match_operand:HI 1 "general_operand" "ir,ir"))]
  ""
  "@
  ld%0\\t%1
  ldi\\t%0,%1")
```

```
(define_expand "reload_inhi"
  [(parallel [(match_operand:HI 0 "register_operand" "=d")
              (match_operand:HI 1 "reload_memory_operand" "m")
              (match_operand:SI 2 "register_operand" "=&d")])]
  ""
  "
{
  mpc_reload_inhi (operands[0], operands[1], operands[2]);
  DONE;
}")

(define_expand "reload_outhi"
  [(parallel [(match_operand:HI 0 "reload_memory_operand" "=m")
              (match_operand:HI 1 "register_operand" "d")
              (match_operand:SI 2 "register_operand" "=&d")])]
  ""
  "
{
  mpc_reload_outhi (operands[0], operands[1], operands[2]);
  DONE;
}")

(define_expand "movsi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "
{
  if (GET_CODE (operands[0]) == MEM) {
    operands[1] = force_reg (SImode, operands[1]);
  }
}")

(define_insn "movsi1"
  [(set (match_operand:SI 0 "general_operand" "=ab,D,m,m,m")
        (match_operand:SI 1 "general_operand" "g,g,ab,D,K"))]
  ""
  "@
  ld%0\\t%1
  ldi\\t%0,%1
  st%1\\t%0
  sti\\t%1,%0
  clr\\t%0")

(define_insn "movsf"
  [(set (match_operand:SF 0 "general_operand" "=A,D,m,m")
        (match_operand:SF 1 "general_operand" "g,g,A,D"))]
  ""
  "@
  ld%0\\t%1
  ldi\\t%0,%1
  st%1\\t%0
  sti\\t%1,%0")

;; Arithmetic operations

; FIXME: somewhere gets a set (mem:SI ***) (plus:...) emitted, which causes
; this pattern to be missed (it doesn't match). The general_operand constraint
; for operand 0/1 fixes this.
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "=a,D")
        (plus:SI (match_operand:SI 1 "general_operand" "%0,0")
                 (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  add%0\\t%2
  addi\\t%0,%2")
```

```
(define_insn "addsf3"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (plus:SF (match_operand:SF 1 "s_register_operand" "%0,0")
                 (match_operand:SF 2 "general_operand" "bDim,g")))]
  ""
  "@
  fpadd%0\\t%2
  fpaddi\\t%0,%2")

(define_insn "subsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (minus:SI (match_operand:SI 1 "s_register_operand" "0,0")
                  (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  sub%0\\t%2
  subi\\t%0,%2")

(define_insn "subsf3"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (minus:SF (match_operand:SF 1 "s_register_operand" "0,0")
                  (match_operand:SF 2 "general_operand" "g,g")))]
  ""
  "@
  fpsub%0\\t%2
  fpsubi\\t%0,%2")

(define_insn "mulsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (mult:SI (match_operand:SI 1 "s_register_operand" "%0,0")
                 (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  mul%0\\t%2
  muli\\t%0,%2")

(define_insn "mulsf3"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (mult:SF (match_operand:SF 1 "s_register_operand" "%0,0")
                 (match_operand:SF 2 "general_operand" "g,g")))]
  ""
  "@
  fpmul%0\\t%2
  fpmuli\\t%0,%2")

(define_insn "divmodsi4"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (div:SI (match_operand:SI 1 "s_register_operand" "0,0")
                (match_operand:SI 2 "general_operand" "g,g")))
   (set (match_operand:SI 3 "s_register_operand" "=b,b")
        (mod:SI (match_dup 1)
                (match_dup 2)))]
  ""
  "@
  sdiv%0\\t%2
  sdivi\\t%0,%2")

(define_insn "udivmodsi4"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (udiv:SI (match_operand:SI 1 "s_register_operand" "0,0")
                 (match_operand:SI 2 "general_operand" "g,g")))
   (set (match_operand:SI 3 "s_register_operand" "=b,b")
        (umod:SI (match_dup 1)
                 (match_dup 2)))]
  ""
  "@
  udiv%0\\t%2
```

```
  udivi\\t%0,%2")

(define_insn "divsf3"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (div:SF (match_operand:SF 1 "s_register_operand" "0,0")
                (match_operand:SF 2 "general_operand" "g,g")))]
  ""
  "@
  fpdiv%0\\t%2
  fpdivi\\t%0,%2")

(define_insn "negsi2"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (neg:SI (match_operand:SI 1 "s_register_operand" "0,0")))]
  ""
  "@
  neg%0
  negi\\t%0")

(define_insn "negsf2"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (neg:SF (match_operand:SF 1 "s_register_operand" "0,0")))]
  ""
  "@
  exor%0\\t#$80000000
  exori\\t%0,#$80000000")

(define_insn "sqrtsf2"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (sqrt:SF (match_operand:SF 1 "s_register_operand" "0,0")))]
  ""
  "@
  fpsqrt%0
  fpsqrti\\t%0")

;; Bit set operations.

(define_expand "ffssi2"
  [(set (match_operand:SI 0 "s_register_operand" "")
        (ffs:SI (match_operand:SI 1 "s_register_operand" "")))]
  ""
  "
{
  emit_insn (gen_ffs_1 (operands[0], operands[1]));
  emit_insn (gen_addsi3 (operands[1], operands[1], const1_rtx));
  DONE;
}")

(define_insn "ffs_1"
  [(set (match_operand:SI 0 "s_register_operand" "=b")
        (ffs:SI (match_operand:SI 1 "s_register_operand" "a")))]
  ""
  "ffbs")

;; Artithmetic shift operations.

; note that it is not allowed to have negative constant shift values
; in the RTL.  Thus CONST_INT are left alone when changing the sign of
; the shifting distance, but this is corrected when the insn is printed,
; because %n is specified in the output pattern.
(define_expand "ashrsi3"
  [(set (match_operand:SI 0 "s_register_operand" "")
        (ashiftrt:SI (match_operand:SI 1 "s_register_operand" "")
                     (match_operand:SI 2 "general_operand" "")))]
  ""
  "
{
  if (GET_CODE (operands[2]) != CONST_INT) {
```

```
    operands[2] = negate_rtx (SImode, operands[2]);
  }
}")

(define_insn ""
  [(set (match_operand:SI 0 "s_register_operand" "=a,a,D,D")
        (ashiftrt:SI (match_operand:SI 1 "s_register_operand" "0,0,0,0")
                     (match_operand:SI 2 "general_operand" "i,g,i,g")))]
  ""
  "@
  ashft%0\\t#%n2
  ashft%0\\t%2
  ashfti\\t%0,#%n2
  ashfti\\t%0,%2")

(define_insn "ashlsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (ashift:SI (match_operand:SI 1 "s_register_operand" "0,0")
                   (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  ashft%0\\t%2
  ashfti\\t%0,%2")

;; Logic operations.

(define_insn "andsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (and:SI (match_operand:SI 1 "s_register_operand" "%0,0")
                (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  and%0\\t%2
  andi\\t%0,%2")

(define_insn "iorsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,a,D")
        (ior:SI (match_operand:SI 1 "s_register_operand" "%0,0,0")
                (match_operand:SI 2 "general_operand" "bDim,0,g")))]
  ""
  "@
  or%0\\t%2
  cmp%0\\t#0
  ori\\t%0,%2")

(define_insn "xorsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (xor:SI (match_operand:SI 1 "s_register_operand" "%0,0")
                (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  exor%0\\t%2
  exori\\t%0,%2")

(define_insn "one_cmplsi2"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (not:SI (match_operand:SI 1 "s_register_operand" "0,0")))]
  ""
  "@
  exor%0\\t#$ffffffff
  exori\\t%0,#$ffffffff")

(define_insn "one_cmplhi2"
  [(set (match_operand:HI 0 "s_register_operand" "=a,D")
        (not:HI (match_operand:HI 1 "s_register_operand" "0,0")))]
  ""
  "@
  exor%0\\t#$ffff
```

```
  exori\\t%0,#$ffff")

(define_insn "one_cmplqi2"
  [(set (match_operand:QI 0 "s_register_operand" "=a,D")
        (not:QI (match_operand:QI 1 "s_register_operand" "0,0")))]
  ""
  "@
  exor%0\\t#$ff
  exori\\t%0,#$ff")

;; Logical shift instructions.

(define_expand "lshrsi3"
  [(set (match_operand:SI 0 "s_register_operand" "")
        (lshiftrt:SI (match_operand:SI 1 "s_register_operand" "")
                     (match_operand:SI 2 "general_operand" "")))]
  ""
  "
{
  if (GET_CODE (operands[2]) != CONST_INT) {
    operands[2] = negate_rtx (SImode, operands[2]);
  }
}")

(define_insn ""
  [(set (match_operand:SI 0 "s_register_operand" "=a,a,D,D")
        (lshiftrt:SI (match_operand:SI 1 "s_register_operand" "0,0,0,0")
                     (match_operand:SI 2 "general_operand" "i,g,i,g")))]
  ""
  "@
  lshft%0\\t#%n2
  lshft%0\\t%2
  lshfti\\t%0,#%n2
  lshfti\\t%0,%2")

;; Rotate instructions.

(define_expand "rotrsi3"
  [(set (match_operand:SI 0 "s_register_operand" "")
        (rotatert:SI (match_operand:SI 1 "s_register_operand" "")
                     (match_operand:SI 2 "general_operand" "")))]
  ""
  "
{
  if (GET_CODE (operands[2]) != CONST_INT) {
    operands[2] = negate_rtx (SImode, operands[2]);
  }
}")

(define_insn ""
  [(set (match_operand:SI 0 "s_register_operand" "=a,a,D,D")
        (rotatert:SI (match_operand:SI 1 "s_register_operand" "0,0,0,0")
                     (match_operand:SI 2 "general_operand" "i,g,i,g")))]
  ""
  "@
  rot%0\\t#%n2
  rot%0\\t%2
  roti\\t%0,#%n2
  roti\\t%0,%2")

(define_insn "rotlsi3"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (rotate:SI (match_operand:SI 1 "s_register_operand" "0,0")
                   (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "@
  rot%0\\t%2
  roti\\t%0,%2")
```

```
;; Test instructions.

(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "a,D,m"))]
  ""
  "@
  cmp%0\\t#0
  cmpi\\t%0,#0
  tst\\t%0")

;; Compare instructions.

(define_insn "cmpsi"
  [(set (cc0)
        (compare (match_operand:SI 0 "s_register_operand" "a,D")
                 (match_operand:SI 1 "nowiden_general_operand" "bDim,g")))]
  ""
  "@
  cmp%0\\t%1
  cmpi\\t%0,%1")

(define_insn "cmpsf"
  [(set (cc0)
        (compare (match_operand:SF 0 "s_register_operand" "a,D")
                 (match_operand:SF 1 "general_operand" "g,g")))]
  ""
  "@
  fpcmp%0\\t%1
  fpcmpi\\t%0,%1")

;; Store-flag instructions.

(define_insn "seq"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (eq (cc0) (const_int 0)))]
  ""
  "seq")

(define_insn "sne"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (ne (cc0) (const_int 0)))]
  ""
  "sne")

(define_insn "sgt"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (gt (cc0) (const_int 0)))]
  ""
  "sgt")

(define_insn "sgtu"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (gtu (cc0) (const_int 0)))]
  ""
  "shi")

(define_insn "slt"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (lt (cc0) (const_int 0)))]
  ""
  "slt")

(define_insn "sltu"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (ltu (cc0) (const_int 0)))]
  ""
```

```
  "slo")

(define_insn "sge"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (ge (cc0) (const_int 0)))]
  ""
  "sge")

(define_insn "sgeu"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (geu (cc0) (const_int 0)))]
  ""
  "shs")

(define_insn "sle"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (le (cc0) (const_int 0)))]
  ""
  "sle")

(define_insn "sleu"
  [(set (match_operand:SI 0 "s_register_operand" "=a")
        (leu (cc0) (const_int 0)))]
  ""
  "sls")

;; Basic conditional jump instructions.

(define_insn "beq"
  [(set (pc)
        (if_then_else (eq (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "beq\\t%l0*4")

(define_insn "bne"
  [(set (pc)
        (if_then_else (ne (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bne\\t%l0*4")

(define_insn "bgt"
  [(set (pc)
        (if_then_else (gt (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bgt\\t%l0*4")

(define_insn "bgtu"
  [(set (pc)
        (if_then_else (gtu (cc0)
                           (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bhi\\t%l0*4")

(define_insn "blt"
  [(set (pc)
        (if_then_else (lt (cc0)
                          (const_int 0))
```

```
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "blt\\t%l0*4")

(define_insn "bltu"
  [(set (pc)
        (if_then_else (ltu (cc0)
                           (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "blo\\t%l0*4")

(define_insn "bge"
  [(set (pc)
        (if_then_else (ge (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bge\\t%l0*4")

(define_insn "bgeu"
  [(set (pc)
        (if_then_else (geu (cc0)
                           (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bhs\\t%l0*4")

(define_insn "ble"
  [(set (pc)
        (if_then_else (le (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "ble\\t%l0*4")

(define_insn "bleu"
  [(set (pc)
        (if_then_else (leu (cc0)
                           (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "bls\\t%l0*4")

;; Negated conditional jump instructions.

(define_insn ""
  [(set (pc)
        (if_then_else (eq (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "bne\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (ne (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
```

```
  "beq\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (gt (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "ble\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (gtu (cc0)
                           (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "bls\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (lt (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "bge\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (ltu (cc0)
                           (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "bhs\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (ge (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "blt\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (geu (cc0)
                           (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "blo\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (le (cc0)
                          (const_int 0))
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "bgt\\t%l0*4")

(define_insn ""
  [(set (pc)
        (if_then_else (leu (cc0)
```

```
                        (const_int 0))
                    (pc)
                    (label_ref (match_operand 0 "" ""))))]
  ""
  "bhi\\t%l0*4")

;; Unconditional and other jump instructions.

(define_insn "jump"
  [(set (pc)
        (label_ref (match_operand 0 "" "")))]
  ""
  "jump\\t%a0")

(define_insn "indirect_jump"
  [(set (pc)
        (match_operand:SI 0 "general_operand" "g"))]
  ""
  "jump\\t%a0")

(define_insn "tablejump"
  [(set (pc)
        (match_operand:SI 0 "general_operand" "g"))
   (use (label_ref (match_operand 1 "" "")))]
  ""
  "jump\\t%a0")

;; Subroutine calls.

(define_expand "call"
  [(call (match_operand:QI 0 "memory_operand" "")
         (match_operand:QI 1 "general_operand" ""))
   (match_operand 2 "" "")
   (match_operand 3 "" "")]
  ""
  "
{
  emit_call_insn (gen_call_1 (operands[0], operands[1]));
  DONE;
}")

(define_insn "call_1"
  [(call (match_operand:QI 0 "memory_operand" "m")
         (match_operand:QI 1 "general_operand" "g"))]
  ""
  "jsub\\t%a0")

(define_expand "call_value"
  [(set (match_operand 0 "s_register_operand" "")
        (call (match_operand:QI 1 "memory_operand" "")
              (match_operand:QI 2 "general_operand" "")))
   (match_operand 3 "" "")
   (match_operand 4 "" "")]
  ""
  "
{
  emit_call_insn (gen_call_value_1 (operands[0],operands[1],operands[2]));
  DONE;
}")

(define_insn "call_value_1"
  [(set (match_operand 0 "s_register_operand" "=b")
        (call (match_operand:QI 1 "memory_operand" "m")
              (match_operand:QI 2 "general_operand" "g")))]
  ""
  "jsub\\t%a1")

(define_expand "untyped_call"
```

```
  [(parallel [(call (match_operand 0 "" "")
                    (const_int 0))
              (match_operand 1 "" "")
              (match_operand 2 "" "")])]
  ""
  "
{
  int i;

  emit_call_insn (gen_call (operands[0], const0_rtx, NULL, const0_rtx));

  for (i = 0; i < XVECLEN (operands[2], 0); i++) {
    rtx set = XVECEXP (operands[2], 0, i);
    emit_move_insn (SET_DEST (set), SET_SRC (set));
  }

  /* The optimizer does not know that the call sets the function value
     registers we stored in the result block.  We avoid problems by
     claiming that all hard registers are used and clobbered at this
     point.  */
  emit_insn (gen_blockage ());

  DONE;
}")

;; UNSPEC_VOLATILE is considered to use and clobber all hard registers and
;; all of memory.  This blocks insns from being moved across this point.

(define_insn "blockage"
  [(unspec_volatile [(const_int 0)] 0)]
  ""
  "")

;; Block operations.

(define_expand "movstrsi"
  [(set (match_operand:BLK 0 "general_operand" "")
        (match_operand:BLK 1 "general_operand" ""))
   (use (match_operand:SI 2 "general_operand" ""))
   (match_operand:SI 3 "" "")]
  ""
  "
{
  emit_insn (gen_movstrsi_1 (operands[0], operands[1], operands[2]));
  DONE;
}")

(define_insn "movstrsi_1"
  [(set (match_operand:BLK 0 "general_operand" "=g")
        (match_operand:BLK 1 "general_operand" "g"))
   (use (match_operand:SI 2 "general_operand" "a"))
   (clobber (match_dup 0))
   (clobber (match_dup 1))
   (clobber (match_dup 2))]
  ""
  "bmovb\\t%1,%0")

;; Conversion instructions.

(define_insn "truncsiqi2"
  [(set (match_operand:QI 0 "s_register_operand" "=a,D")
        (truncate:QI (match_operand:SI 1 "s_register_operand" "0,0")))]
  ""
  "@
  and%0\\t#$ff
  andi\\t%0,#$ff")

(define_insn "trunchiqi2"
```

```
  [(set (match_operand:QI 0 "s_register_operand" "=a,D")
        (truncate:QI (match_operand:HI 1 "s_register_operand" "0,0")))]
  ""
  "@
  and%0\\t#$ff
  andi\\t%0,#$ff")

(define_insn "truncsihi2"
  [(set (match_operand:HI 0 "s_register_operand" "=a,D")
        (truncate:HI (match_operand:SI 1 "s_register_operand" "0,0")))]
  ""
  "@
  and%0\\t#$ffff
  andi\\t%0,#$ffff")

(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "")
        (zero_extend:SI (match_operand:HI 1 "register_operand" "")))]
  ""
  "
{
  if (GET_CODE (operands[1]) == SUBREG) {
    operands[1] = gen_rtx (SUBREG, SImode, SUBREG_REG (operands[1]),
                           SUBREG_WORD (operands[1]));
  } else {
    operands[1] = gen_rtx (SUBREG, SImode, operands[1], 0);
  }
  emit_insn (gen_andsi3 (operands[0], operands[1], GEN_INT (65535)));
  DONE;
}")

(define_insn "zero_extendqisi2"
  [(set (match_operand:SI 0 "s_register_operand" "=a,a,D,D")
        (zero_extend:SI (match_operand:QI 1 "general_operand" "0,m,0,m")))]
  ""
  "@
  and%0\\t#$ff
  ldb%0\\t%1
  andi\\t%0,#$ff
  ldbi\\t%0,%1")

(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "=a,D")
        (sign_extend:SI (match_operand:HI 1 "register_operand" "0,0")))]
  ""
  "@
  ashft%0\\t#16\;ashft%0\\t#-16
  ashfti\\t%0,#16\;ashfti\\t%0,#-16")

(define_insn "extendqisi2"
  [(set (match_operand:SI 0 "register_operand" "=a,D")
        (sign_extend:SI (match_operand:QI 1 "register_operand" "0,0")))]
  ""
  "@
  extb%0
  extbi\\t%0")

(define_insn "floatsisf2"
  [(set (match_operand:SF 0 "s_register_operand" "=a,D")
        (float:SF (match_operand:SI 1 "s_register_operand" "0,0")))]
  ""
  "@
  conir%0
  coniri\\t%0")

(define_insn "fix_truncsfsi2"
  [(set (match_operand:SI 0 "s_register_operand" "=a,D")
        (fix:SI (fix:SF (match_operand:SF 1 "s_register_operand" "0,0"))))]
```

```
  ""
  "@
  conri%0
  conrii\\t%0")

;; The most important instruction on every machine.

(define_insn "nop"
  [(const_int 0)]
  ""
  "; NOP")
```

D.1.2 C-Makros `mpc.h`

```
/* Definitions of target machine for GNU compiler, for MONADS-PC.
   Written by Klaus Espenlaub (kespenla@student.informatik.uni-ulm.de).

This file is is derived from software covered by the terms of the
GNU General Public License as published by the Free Software Foundation.
Consequently this file is subject to these terms.  */

/* 6.1: Controlling the Compilation driver, 'gcc'.  */

#undef SWITCH_TAKES_ARG(CHAR)
#undef WORD_SWITCH_TAKES_ARG(STR)

#define CPP_SPEC                                                        \
  "-D__SHRT_MAX__=2147483647 %{posix:-D_POSIX_SOURCE}"

#undef CC1_SPEC
#undef CC1PLUS_SPEC

#define ASM_SPEC "%{v:-V}"

#undef ASM_FINAL_SPEC

#define LINK_SPEC "%{v:-V}"

#define LIB_SPEC                                                        \
  "%{mieee-fp:-lieee} %{p:-lgmon} %{pg:-lgmon} "                        \
  "%{profile:-lgmon -lc_p} %{!profile:-lc}"
#undef LIBGCC_SPEC

#define STARTFILE_SPEC                                                  \
  "%{pg:gcrt1.o%s} %{!pg:%{p:gcrt1.o%s} %{!p:crt1.o%s}} crti.o%s crtbegin.o%s"
#define ENDFILE_SPEC "crtend.o%s crtn.o%s"

/* 6.2: Run-time Target Specification.  */

/* Names to predefine in the preprocessor for this target machine.  */
#define CPP_PREDEFINES "-DMPC -DMPCOS -Acpu(mpc) -Amachine(mpc)"

/* Run-time compilation parameters selecting different hardware subsets.  */
extern int target_flags;

/* Macros used in the machine description to test the flags.  */

/* Masks for the -m switches.  */
#define MASK_LINENO     000000000001    /* generate SETL instrunctions.  */

/* Generate SETL instructions to aid debugging.  */
#define TARGET_LINENO (target_flags & MASK_LINENO)

/* Macro to define tables used to set the flags.
   This is a list in braces of pairs in braces,
   each pair being { "NAME", VALUE }
```

```
   where VALUE is the bits to set or minus the bits to clear.
   An empty string NAME is used to identify the default VALUE.  */
#define TARGET_SWITCHES                                                 \
  { {"lineno",     MASK_LINENO },                                       \
    {"no-lineno", -MASK_LINENO },                                       \
    {"",           0 } }

/* This macro is similar to 'TARGET_SWITCHES' but defines names of
   command options that have values.  Its definition is an
   initializer with a subgrouping for each command option.

   Each subgrouping contains a string constant, that defines the
   fixed part of the option name, and the address of a variable.
   The variable, type 'char *', is set to the variable part of the
   given option if the fixed part matches.  The actual option name
   is made by appending '-m' to the specified name.  */
#undef TARGET_OPTIONS

/* Print subsidiary information on the compiler version in use.  */
#define TARGET_VERSION fprintf (stderr, " (mpc)");

/* Sometimes certain combinations of command options do not make
   sense on a particular target machine.  You can define a macro
   'OVERRIDE_OPTIONS' to take account of this.  This macro, if
   defined, is executed once just after all the command options have
   been parsed.

   Don't use this macro to turn on various extra optimizations for
   '-O'.  That is what 'OPTIMIZATION_OPTIONS' is for.  */
#undef OVERRIDE_OPTIONS

/* Show we can debug even without a frame pointer.  */
#define CAN_DEBUG_WITHOUT_FP

/* 6.3: Storage Layout.  */

/* Define this if most significant bit is lowest numbered
   in instructions that operate on numbered bit-fields.  */
#define BITS_BIG_ENDIAN 0

/* Define this if most significant byte of a word is the lowest numbered.  */
#define BYTES_BIG_ENDIAN 0

/* Define this if most significant word of a multiword number is the lowest
   numbered.  */
#define WORDS_BIG_ENDIAN 0

/* number of bits in an addressable storage unit.  */
#define BITS_PER_UNIT 8

/* Width in bits of a "word", which is the contents of a machine register.
   Note that this is not necessarily the width of data type 'int';
   if using 16-bit ints on a 68000, this would still be 32.
   But on a machine with 16-bit registers, this would be 16.  */
#define BITS_PER_WORD 32

/* Width of a word, in units (bytes).  */
#define UNITS_PER_WORD 4

/* Width in bits of a pointer.  See also the macro 'Pmode' defined below.  */
#define POINTER_SIZE 32

/* Define this macro if it is advisable to hold scalars in registers
   in a wider mode than that declared by the program.  In such cases,
   the value is constrained to be within the bounds of the declared
   type, but kept valid in the wider mode.  The signedness of the
   extension may differ from that of the type.  */
#define PROMOTE_MODE(MODE,UNSIGNEDP,TYPE)                               \
```

```
  if (GET_MODE_CLASS (MODE) == MODE_INT &&                              \
      GET_MODE_SIZE (MODE) < UNITS_PER_WORD) {                          \
    (MODE) = SImode;                                                    \
  }

/* Define this if function arguments should also be promoted using
   the above procedure.  */
#define PROMOTE_FUNCTION_ARGS

/* Likewise, if the function return value is promoted.  */
#define PROMOTE_FUNCTION_RETURN

/* Allocation boundary (in *bits*) for storing arguments in argument list.  */
#define PARM_BOUNDARY 32

/* Boundary (in *bits*) on which stack pointer should be aligned.  */
#define STACK_BOUNDARY 32

/* Allocation boundary (in *bits*) for the code of a function.  */
/* On the MPC, code need not be aligned, the assembler takes care of that.  */
#define FUNCTION_BOUNDARY 8

/* No data type wants to be aligned rounder than this.  */
#define BIGGEST_ALIGNMENT 32

/* The best alignment to use in cases where we have a choice.  */
#define FASTEST_ALIGNMENT 32

/* Make arrays of chars and shorts word-aligned to make accesses faster.  */
#define DATA_ALIGNMENT(TYPE, ALIGN)                                     \
  ((TREE_CODE (TYPE) == ARRAY_TYPE)                                     \
   && ((TYPE_MODE (TREE_TYPE (TYPE)) == QImode)                         \
       || (TYPE_MODE (TREE_TYPE (TYPE)) == HImode))                     \
   && ((ALIGN) < FASTEST_ALIGNMENT) ? FASTEST_ALIGNMENT : (ALIGN))

/* Make strings word-aligned so strcpy from constants will be faster.  */
#define CONSTANT_ALIGNMENT(EXP, ALIGN)                                  \
  ((TREE_CODE (EXP) == STRING_CST && (ALIGN) < FASTEST_ALIGNMENT)       \
   ? FASTEST_ALIGNMENT : (ALIGN))

/* Alignment of field after 'int : 0' in a structure.  */
#define EMPTY_FIELD_BOUNDARY 32

/* Every structure's size must be a multiple of this.  */
#define STRUCTURE_SIZE_BOUNDARY 32

/* Set this nonzero if move instructions will actually fail to work
   when given unaligned data.  */
#define STRICT_ALIGNMENT 1

/* Our machine description cannot handle bit fields that overlap an
   alignment boundary.  */
#define PCC_BITFIELD_TYPE_MATTERS 1

/* An integer expression for the size in bits of the largest integer
   machine mode that should actually be used.  All integer machine modes
   of this size or smaller can be used for structures and unions with
   the appropriate sizes.  */
#define MAX_FIXED_MODE_SIZE 32

/* A code distinguishing the floating point format of the target
   machine.  There are three defined values: IEEE_FLOAT_FORMAT,
   VAX_FLOAT_FORMAT, and UNKNOWN_FLOAT_FORMAT.  */
#define TARGET_FLOAT_FORMAT IEEE_FLOAT_FORMAT

/* 6.4: Layout of Source Language Data Types.  */

#define SHORT_TYPE_SIZE INT_TYPE_SIZE
```

```
/* The sizes of the various supported floating point types.  */

#define FLOAT_TYPE_SIZE 32
#define DOUBLE_TYPE_SIZE 32
#define LONG_DOUBLE_TYPE_SIZE 32

/* Define this as 1 if 'char' should by default be signed; else as 0.  */
#define DEFAULT_SIGNED_CHAR 1

/* Define results of standard character escape sequences.  */
#define TARGET_BELL 007
#define TARGET_BS 010
#define TARGET_TAB 011
#define TARGET_NEWLINE 012
#define TARGET_VT 013
#define TARGET_FF 014
#define TARGET_CR 015

/* 6.5: Register Usage.  */

/* Number of actual hardware registers.
   The hardware registers are assigned numbers for the compiler
   from 0 to just below FIRST_PSEUDO_REGISTER.
   All registers that the compiler knows about must be given numbers,
   even those that are not normally considered general registers.

   The following register layout is used:
     0:   A
     1:   AX
     2:   I1
     3:   I2
     4:   I3    (arg and frame pointer, eliminable)
     5:   I0    (stack pointer) */
#define FIRST_PSEUDO_REGISTER 6

/* 1 for registers that have pervasive standard uses
   and are not available for the register allocator.  */
#define FIXED_REGISTERS {0, 0, 0, 0, 0, 1}

/* 1 for registers not available across function calls.
   These must include the FIXED_REGISTERS and also any
   registers that can be used without being saved.
   The latter must include the registers where values are returned
   and the register where structure-value addresses are passed.
   Aside from that, you can include as many other registers as you like.  */
#define CALL_USED_REGISTERS {1, 1, 1, 1, 0, 1}

/* Order in which to allocate registers.  Each register must be
   listed once, even those in FIXED_REGISTERS.  List frame pointer
   late and fixed registers last.  Note that, in general, we prefer
   registers listed in CALL_USED_REGISTERS, keeping the others
   available for storage of persistent values.  */
/*#define REG_ALLOC_ORDER {REG_A, REG_I1, REG_I2, REG_I3, REG_AX, REG_I0}*/

/* Return number of consecutive hard regs needed starting at reg REGNO
   to hold something of mode MODE.
   This is ordinarily the length in words of a value of mode MODE
   but can be less for certain modes in special long registers.  */
#define HARD_REGNO_NREGS(REGNO, MODE)                                   \
   ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* Value is 1 if hard register REGNO can hold a value of machine-mode MODE.  */
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1

/* Value is 1 if it is a good idea to tie two pseudo registers
   when one has mode MODE1 and one has mode MODE2.
   If HARD_REGNO_MODE_OK could produce different values for MODE1 and MODE2,
```

```
   for any hard reg, then this must be 0 for correct output.  */
#define MODES_TIEABLE_P(MODE1, MODE2) 1

/* 6.6: Register Classes.  */

/* Define the classes of registers for register constraints in the
   machine description.  Also define ranges of constants.

   One of the classes must always be named ALL_REGS and include all hard regs.
   If there is more than one class, another class must be named NO_REGS
   and contain no registers.

   The name GENERAL_REGS must be the name of a class (or an alias for
   another name such as ALL_REGS).  This is the class of registers
   that is allowed by "g" or "r" in a register constraint.
   Also, registers outside this class are allocated only when
   instructions express preferences for them.

   The classes must be numbered in nondecreasing order; that is,
   a larger-numbered class must never be contained completely
   in a smaller-numbered class.

   For any two classes, it is very desirable that there be another
   class that represents their union.  */
enum reg_class {
  NO_REGS,
  AREG,
  AXREG,
  ACCU_REGS,
  INDEX_REGS,
  A_INDEX_REGS,
  ALL_REGS, LIM_REG_CLASSES
};


#define GENERAL_REGS ALL_REGS

#define N_REG_CLASSES (int) LIM_REG_CLASSES

#define REG_A 0
#define REG_AX 1
#define REG_I1 2
#define REG_I2 3
#define REG_I3 4
#define REG_I0 5


/* Give names of register classes as strings for dump file.   */
#define REG_CLASS_NAMES                                                 \
{  "NO_REGS",                                                           \
   "AREG",                                                              \
   "AXREG",                                                             \
   "ACCU_REGS",                                                         \
   "INDEX_REGS",                                                        \
   "A_INDEX_REGS",                                                      \
   "ALL_REGS"}

/* Define which registers fit in which classes.
   This is an initializer for a vector of HARD_REG_SET
   of length N_REG_CLASSES.  */
#define REG_CLASS_CONTENTS                                              \
{    0x00,                                                              \
     0x01,                      /* AREG */                              \
     0x02,                      /* AXREG */                             \
     0x03,                      /* ACCU_REGS */                         \
     0x3c,                      /* INDEX_REGS */                        \
     0x3d,                      /* A_INDEX_REGS */                      \
     0x3f }                     /* ALL_REGS */
```

```
/* The same information, inverted:
   Return the class number of the smallest class containing
   reg number REGNO.  This could be a conditional expression
   or could index an array.  */
#define REGNO_REG_CLASS(REGNO)                                          \
  ((REGNO) == REG_A ? AREG :                                            \
   (REG_I1 <= (REGNO) && (REGNO) <= REG_I0) ? INDEX_REGS :              \
   (REGNO) == REG_AX ? AXREG : NO_REGS)

/* The class value for index registers, and the one for base regs.  */
#define BASE_REG_CLASS INDEX_REGS
#define INDEX_REG_CLASS NO_REGS

/* Get reg_class from a letter such as appears in the machine description.  */
#define REG_CLASS_FROM_LETTER(C)                                        \
  ((C) == 'a' ? AREG :                                                  \
   (C) == 'b' ? AXREG :                                                 \
   (C) == 'A' ? ACCU_REGS :                                             \
   (C) == 'D' ? INDEX_REGS :                                            \
   (C) == 'd' ? A_INDEX_REGS : NO_REGS)

/* These assume that REGNO is a hard or pseudo reg number.
   They give nonzero only if REGNO is a hard reg of the suitable class
   or a pseudo reg currently allocated to a suitable hard reg.
   Since they use reg_renumber, they are safe only once reg_renumber
   has been allocated, which happens in local-alloc.c.  */
#define REGNO_OK_FOR_BASE_P(REGNO)                                      \
  ((REG_I1 <= (REGNO) && (REGNO) <= REG_I0)                             \
   || (REG_I1 <= reg_renumber[REGNO] && reg_renumber[REGNO] <= REG_I0))
#define REGNO_OK_FOR_INDEX_P(REGNO) 0

/* Given an rtx X being reloaded into a reg required to be
   in class CLASS, return the class of reg to actually use.
   In general this is just CLASS; but on some machines
   in some cases it is preferable to use a more restrictive class.  */
#define PREFERRED_RELOAD_CLASS(X, CLASS) (CLASS)

/* Place additional restrictions on the register class to use when it
   is necessary to be able to hold a value of mode MODE in a reload
   register for which class CLASS would ordinarily be used. */
#define LIMIT_RELOAD_CLASS(MODE, CLASS)                                 \
  (((MODE) == QImode || (MODE) == HImode) && (CLASS)==ALL_REGS ? A_INDEX_REGS :\
   ((MODE) == QImode || (MODE) == HImode) && (CLASS) == ACCU_REGS ? AREG :\
   (CLASS))

/* Return the register class of a scratch register needed to copy IN into
   or out of a register in CLASS in MODE.  If it can be done directly,
   NO_REGS is returned.  */
#define SECONDARY_RELOAD_CLASS(CLASS, MODE, IN)                         \
  mpc_secondary_reload_class ((CLASS), (MODE), (IN))

/* When defined, the compiler allows registers explicitly used in the
   rtl to be used as spill registers but prevents the compiler from
   extending the lifetime of these registers.  */
#define SMALL_REGISTER_CLASSES 1

/* Return the maximum number of consecutive registers
   needed to represent mode MODE in a register of class CLASS.  */
#define CLASS_MAX_NREGS(CLASS, MODE)                                    \
  ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* FIXME: The letters I, J, K, L and M in a register constraint string
   can be used to stand for particular ranges of immediate operands.
   This macro defines what the ranges are.
   C is the letter, and VALUE is a constant value.
   Return 1 if VALUE is in the range specified by C.
```

```
   For MPC, 'I' is used for l16 values, as they fit in the instruction.
   'J' is the useable range of shifts, 'K' is the value 0, used for
   the clr mem insn.  */
#define CONST_OK_FOR_LETTER_P(VALUE, C)                                 \
  ((C) == 'I' ? -32768 <= (VALUE) && (VALUE) <= 32767 :                 \
   (C) == 'J' ? -31 <= (VALUE) && (VALUE) <= 31 :                       \
   (C) == 'K' ? (VALUE) == 0 :                                          \
   0)

/* Similar, but for floating constants, and defining letters G and H.
   Here VALUE is the CONST_DOUBLE rtx itself.  */
#define CONST_DOUBLE_OK_FOR_LETTER_P(VALUE, C) 1

/* 6.7: Stack Layout and Calling Conventions.  */

/* Define this if pushing a word on the stack
   makes the stack pointer a smaller address.  */
#undef STACK_GROWS_DOWNWARD

/* Define this if the nominal address of the stack frame
   is at the high-address end of the local variables;
   that is, each additional local variable allocated
   goes at a more negative offset in the frame.  */
#undef FRAME_GROWS_DOWNWARD

/* Define this if successive args to a function occupy decreasing
   addresses on the stack.  */
#define ARGS_GROW_DOWNWARD

/* Offset within stack frame to start allocating local variables at.
   If FRAME_GROWS_DOWNWARD, this is the offset to the END of the
   first local allocated.  Otherwise, it is the offset to the BEGINNING
   of the first local allocated.  */
#define STARTING_FRAME_OFFSET 8

/* Offset from the stack pointer to the first available location.  */
#define STACK_POINTER_OFFSET 0

/* Offset of first parameter from the argument pointer register value.  */
#define FIRST_PARM_OFFSET(FNDECL) 0

/* FIXME: Given an rtx for the address of a frame,
   return an rtx for the address of the word in the frame
   that holds the dynamic chain--the previous frame's address.  */
/* #define DYNAMIC_CHAIN_ADDRESS(FRAME) */

/* Register to use for pushing function arguments.  */
#define STACK_POINTER_REGNUM REG_I0

/* Base register for access to local variables of the function.  */
#define FRAME_POINTER_REGNUM REG_I3

/* Base register for access to arguments of the function.  */
#define ARG_POINTER_REGNUM FRAME_POINTER_REGNUM

/* Register in which static-chain is passed to a function.  */
#define STATIC_CHAIN_REGNUM REG_I1

/* Value should be nonzero if functions must have frame pointers.
   Zero means the frame pointer need not be set up (and parms
   may be accessed via the stack pointer) in functions that seem suitable.
   This is computed in 'reload', in reload1.c.  */
#define FRAME_POINTER_REQUIRED 0

/* C statement to store the difference between the frame pointer
   and the stack pointer values immediately after the function prologue.  */
#define INITIAL_FRAME_POINTER_OFFSET(VAR)                               \
  (VAR) = -compute_frame_size (get_frame_size ())
```

```
/* When a prototype says 'char' or 'short', really pass an 'int'.  */
#define PROMOTE_PROTOTYPES

/* Define this if the maximum size of all the outgoing args is to be
   accumulated and pushed during the prologue.  The amount can be
   found in the variable current_function_outgoing_args_size.  */
#define ACCUMULATE_OUTGOING_ARGS

/* Value is the number of bytes of arguments automatically
   popped when returning from a subroutine call.
   FUNDECL is the declaration node of the function (as a tree),
   FUNTYPE is the data type of the function (as a tree),
   or for a library call it is an identifier node for the subroutine name.
   SIZE is the number of bytes of arguments passed on the stack.  */
#define RETURN_POPS_ARGS(FUNDECL, FUNTYPE, SIZE) 0

/* Determine where to put an argument to a function.
   Value is zero to push the argument on the stack,
   or a hard register in which to store the argument.

   MODE is the argument's machine mode.
   TYPE is the data type of the argument (as a tree).
    This is null for libcalls where that information may
    not be available.
   CUM is a variable of type CUMULATIVE_ARGS which gives info about
    the preceding args and about the function being called.
   NAMED is nonzero if this argument is a named parameter
    (otherwise it is an extra parameter matching an ellipsis).  */
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) 0

/* For an arg passed partly in registers and partly in memory,
   this is the number of registers used.
   For args passed entirely in registers or entirely in memory, zero.  */
#define FUNCTION_ARG_PARTIAL_NREGS(CUM, MODE, TYPE, NAMED) 0

/* FIXME: A C expression that indicates when an argument must be passed by
   reference.  If nonzero for an argument, a copy of that argument is
   made in memory and a pointer to the argument is passed instead of
   the argument itself.  The pointer is passed in whatever way is
   appropriate for passing a pointer to that type.  */
#define FUNCTION_ARG_PASS_BY_REFERENCE(CUM, MODE, TYPE, NAMED) 0

/* FIXME: A C expression that indicates when it is the called function's
   responsibility to make copies of arguments passed by reference.
   If the callee can determine that the argument won't be modified, it can
   avoid the copy.  */
#define FUNCTION_ARG_CALLEE_COPIES(CUM, MODE, TYPE, NAMED) 0

/* Define a data type for recording info about an argument list
   during the scan of that argument list.  This data type should
   hold all necessary information about the function itself
   and about the args processed so far, enough to enable macros
   such as FUNCTION_ARG to determine where the next arg should go.  */
#define CUMULATIVE_ARGS int

/* Initialize a variable CUM of type CUMULATIVE_ARGS
   for a call to a function whose data type is FNTYPE.
   For a library call, FNTYPE is 0.  */
#define INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, INDIRECT)            \
  (CUM) = 0

/* Update the data in CUM to advance over an argument
   of mode MODE and data type TYPE.
   (TYPE is null for libcalls where that information may not be available.)  */
#define FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED)

/* 1 if N is a possible register number for function argument passing.  */
```

```
#define FUNCTION_ARG_REGNO_P(N) 0

/* Define how to find the value returned by a function.
   VALTYPE is the data type of the value (as a tree).
   If the precise function being called is known, FUNC is its FUNCTION_DECL;
   otherwise, FUNC is 0.  */
#define FUNCTION_VALUE(VALTYPE, FUNC)   \
  gen_rtx (REG, TYPE_MODE (VALTYPE), REG_AX)

/* Define how to find the value returned by a library function
   assuming the value has mode MODE.  */
#define LIBCALL_VALUE(MODE)                                             \
  gen_rtx (REG, MODE, REG_AX)

/* 1 if N is a possible register number for a function value.  */
#define FUNCTION_VALUE_REGNO_P(N)                                       \
  ((N) == REG_AX)

/* Return true if X should be returned in memory.  */
#define RETURN_IN_MEMORY(X)                                             \
  (GET_MODE_SIZE (TYPE_MODE (X)) > UNITS_PER_WORD ||                    \
   TYPE_MODE (X) == BLKmode)

/* RTX where the address to store structure values at is passed.
   0 means the address is passed as an 'invisible' first argument.  */
#define STRUCT_VALUE                                                    \
  gen_rtx (REG, Pmode, REG_A)

/* Indicate that caller save should be done by default if it looks
   profitable.  */
#define DEFAULT_CALLER_SAVES

/* This macro generates the assembly code for function entry.
   FILE is a stdio stream to output the code to.
   SIZE is an int: how many units of temporary storage to allocate.
   Refer to the array 'regs_ever_live' to determine which registers
   to save; 'regs_ever_live[I]' is nonzero if register number I
   is ever used in the function.  This macro is responsible for
   knowing which registers should not be saved even if used.  */
#define FUNCTION_PROLOGUE(FILE, SIZE)                                   \
  function_prologue (FILE, SIZE)

/* EXIT_IGNORE_STACK should be nonzero if, when returning from a function,
   the stack pointer does not matter.  The value is tested only in
   functions that have frame pointers.
   No definition is equivalent to always zero.  */
#define EXIT_IGNORE_STACK 1

/* This macro generates the assembly code for function exit,
   on machines that need it.  If FUNCTION_EPILOGUE is not defined
   then individual return instructions are generated for each
   return statement.  Args are same as for FUNCTION_PROLOGUE.

   The function epilogue should not depend on the current stack pointer!
   It should use the frame pointer only.  This is mandatory because
   of alloca; we also take advantage of it to omit stack adjustments
   before returning.

   If the last non-note insn in the function is a BARRIER, then there
   is no need to emit a function prologue, because control does not fall
   off the end.  This happens if the function ends in an "exit" call, or
   if a 'return' insn is emitted directly into the function.  */
#define FUNCTION_EPILOGUE(FILE, SIZE)                                   \
  do {                                                                  \
    rtx last = get_last_insn ();                                        \
    if (last && GET_CODE (last) == NOTE)                                \
      last = prev_nonnote_insn (last);                                  \
    if (! last || GET_CODE (last) != BARRIER)                           \
```

```
      function_epilogue (FILE, SIZE);                                   \
  } while (0)

/* Output assembler code to FILE to increment profiler label # LABELNO
   for profiling a function entry.  */
#define FUNCTION_PROFILER(FILE, LABELNO)                                \
  do {                                                                  \
    asm_fprintf (FILE, "\tlda\t%LLP%u(cx)\n", (LABELNO));               \
    asm_fprintf (FILE, "\tjsub\t%I%Umcount\n");                         \
  } while (0)

/* Output assembler code to FILE to initialize this source file's
   basic block profiling info, if that has not already been done.  */
#define FUNCTION_BLOCK_PROFILER(FILE, LABELNO)                          \
  do {                                                                  \
    asm_fprintf (FILE, "\tlda\t%LLPBX0(cx)\n");                         \
    asm_fprintf (FILE, "\tbne\t%LLPI%u\n", LABELNO);                    \
    asm_fprintf (FILE, "\taddi\t%s,%Iu\n",                              \
                       reg_names[STACK_POINTER_REGNUM], UNITS_PER_WORD);\
    asm_fprintf (FILE, "\tlda\t%LLPBX0(cx)\n");                         \
    asm_fprintf (FILE, "\tsta\t%i(%Rcx)[%s]\n, -UNITS_PER_WORD,         \
                       reg_names[STACK_POINTER_REGNUM]");               \
    asm_fprintf (FILE, "\tjsub\t%I%U__bb_init_func\n");                 \
    asm_fprintf (FILE, "\tsubi\t%s,%I%u\n",                             \
                       reg_names[STACK_POINTER_REGNUM], UNITS_PER_WORD);\
    asm_fprintf (FILE, "%LLPI%u:\n", LABELNO);                          \
  } while (0)

/* Output assembler code to FILE to increment the entry-count for
   the BLOCKNO'th basic block in this source file.  */
#define BLOCK_PROFILER(FILE, BLOCKNO)                                   \
    asm_fprintf (FILE, "\tinc\t%LLPBX2+%u(%Rcx)\n", 4*BLOCKNO)

/* 6.8: Implementing the Varargs Macros.  */

/* The only thing we need to do is to make sure that the upwards growing
   stack with downwards growing args is handled properly.  The implementation
   is in ginclude/va-mpc.h.  */

/* 6.9: Trampolines for Nested Functions.  */

/* Output assembler code for a block containing the constant parts
   of a trampoline, leaving space for the variable parts.

   In our case this would be VERY difficult to implement, as code can not
   be executed in the data area.  The transfer routine must be called
   somehow when the trampoline is used.  But since we need not support
   all crazy C extension gcc comes up with, we can wait for a working
   implementation until someone wants to port a language that really
   supports nested functions (eg. Pascal).  */
#define TRAMPOLINE_TEMPLATE(FILE)                                       \
  fatal ("trampolines are not supported on this platform");

/* Length in units of the trampoline for entering a nested function.  */
#define TRAMPOLINE_SIZE 0

/* Emit RTL insns to initialize the variable parts of a trampoline.
   FNADDR is an RTX for the address of the function's pure code.
   CXT is an RTX for the static chain value for the function.  */
#define INITIALIZE_TRAMPOLINE(TRAMP, FNADDR, CXT)                       \
  fatal ("trampoline usage not supported on this platform");


/* 6.10: Implicit Calls to Library Routines.  */

/* Implicit library calls should use memcpy, not bcopy, etc.  */
#define TARGET_MEM_FUNCTIONS
```

```
#define FLOAT_VALUE_TYPE float
#define INTIFY(FLOATVAL) FLOATVAL

/* 6.11: Addressing Modes.  */

#undef HAVE_POST_INCREMENT
#undef HAVE_POST_DECREMENT

#undef HAVE_PRE_DECREMENT
#undef HAVE_PRE_INCREMENT

/* Recognize any constant value that is a valid address.  */
#define CONSTANT_ADDRESS_P(X)                                          \
  (GET_CODE (X) == LABEL_REF || GET_CODE (X) == SYMBOL_REF ||          \
   GET_CODE (X) == CONST_INT || GET_CODE (X) == CONST)

/* Maximum number of registers that can appear in a valid memory address.  */
#define MAX_REGS_PER_ADDRESS 1

/* GO_IF_LEGITIMATE_ADDRESS recognizes an RTL expression
   that is a valid memory address for an instruction.
   The MODE argument is the machine mode for the MEM expression
   that wants to use this address.  */
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)                        \
  {                                                                    \
    if (CONSTANT_ADDRESS_P (X)) goto ADDR;                             \
    if (REG_P (X) && REG_OK_FOR_BASE_P (X)) goto ADDR;                 \
    if (GET_CODE (X) == PLUS                                           \
        && REG_P (XEXP (X, 0))                                         \
        && REG_OK_FOR_BASE_P (XEXP (X, 0))                             \
        && CONSTANT_ADDRESS_P (XEXP (X, 1))) goto ADDR;                \
  }

/* The macros REG_OK_FOR..._P assume that the arg is a REG rtx
   and check its validity for a certain class.
   We have two alternate definitions for each of them.
   The usual definition accepts all pseudo regs; the other rejects
   them unless they have been allocated suitable hard regs.
   The symbol REG_OK_STRICT causes the latter definition to be used.

   Most source files want to accept pseudo regs in the hope that
   they will get allocated to the class that the insn wants them to be in.
   Source files for reload pass need to be strict.
   After reload, it makes no difference, since pseudo regs have
   been eliminated by then.  */

#ifndef REG_OK_STRICT

/* Nonzero if X is a hard reg that can be used as a base reg
   or if it is a pseudo reg.  */
#define REG_OK_FOR_BASE_P(X)                                           \
  ((REG_I1 <= REGNO (X) && REGNO (X) <= REG_I0) ||                     \
   REGNO (X) >= FIRST_PSEUDO_REGISTER)

/* Nonzero if X is a hard reg that can be used as an index
   or if it is a pseudo reg.  */
#define REG_OK_FOR_INDEX_P(X) 0

#else /* REG_OK_STRICT */

/* Nonzero if X is a hard reg that can be used as a base reg.  */
#define REG_OK_FOR_BASE_P(X) REGNO_OK_FOR_BASE_P (REGNO (X))

/* Nonzero if X is a hard reg that can be used as an index.  */
#define REG_OK_FOR_INDEX_P(X) REGNO_OK_FOR_INDEX_P (REGNO (X))

#endif /* REG_OK_STRICT */
```

```
/* Try machine-dependent ways of modifying an illegitimate address
   to be legitimate.  If we find one, return the new, valid address.
   This macro is used in only one place: 'memory_address' in explow.c.

   OLDX is the address as it was before break_out_memory_refs was called.
   In some cases it is useful to look at this to decide what needs to be done.

   MODE and WIN are passed so that this macro can use
   GO_IF_LEGITIMATE_ADDRESS.

   It is always safe for this macro to do nothing.  It exists to recognize
   opportunities to optimize the output.  */
#define LEGITIMIZE_ADDRESS(X, OLDX, MODE, WIN)

/* Go to LABEL if ADDR (a legitimate address expression)
   has an effect that depends on the machine mode it is used for.  */
#define GO_IF_MODE_DEPENDENT_ADDRESS(ADDR, LABEL)

/* Nonzero if the constant value X is a legitimate general operand.
   It is given that X satisfies CONSTANT_P or is a CONST_DOUBLE.  */
#define LEGITIMATE_CONSTANT_P(X) 1

/* 6.12: Condition Code Status.  */

/* Store in cc_status the expressions that the condition codes will
   describe after execution of an instruction whose pattern is EXP.
   Do not alter them if the instruction would not alter the cc's.  */
#define NOTICE_UPDATE_CC(EXP, INSN)                                     \
  notice_update_cc(EXP)

/* 6.13: Describing Relative Costs of Operations.  */

/* FIXME: Compute the cost of computing a constant rtl expression RTX
   whose rtx-code is CODE.  The body of this macro is a portion
   of a switch statement.  If the code is computed here,
   return it with a return statement.  Otherwise, break from the switch.  */
#define CONST_COSTS(RTX, CODE, OUTER_CODE)                              \
  case CONST_INT:                                                       \
    if (INTVAL (RTX) >= -32768 && INTVAL (RTX) < 32767) return 0;       \
    return 1;                                                           \
  case CONST:                                                           \
  case CONST_DOUBLE:                                                    \
  case LABEL_REF:                                                       \
  case SYMBOL_REF:                                                      \
    return 1;

/* FIXME: Provide the costs of a rtl expression.  This is in the body of a
   switch on CODE.  */
#define RTX_COSTS(X, CODE, OUTER_CODE)                                  \
  case MULT:                                                            \
    return COSTS_N_INSNS (20);                                          \
  case DIV:                                                             \
  case MOD:                                                             \
    return COSTS_N_INSNS (40);                                          \
  case UDIV:                                                            \
  case UMOD:                                                            \
    return COSTS_N_INSNS (40);                                          \
  case FFS:                                                             \
    return COSTS_N_INSNS (10);                                          \
  case ASHIFTRT:                                                        \
  case LSHIFTRT:                                                        \
  case ASHIFT:                                                          \
    return COSTS_N_INSNS (15);                                          \
  /* Make floating point conversion very expensive, so that CSE will    \
     favor using CONST_DOUBLE.  */                                      \
  case FLOAT:                                                           \
  case FIX:                                                             \
    return COSTS_N_INSNS (200);
```

```
/* FIXME: Compute the cost of an address.  This is meant to approximate the size
   and/or execution delay of an insn using that address.  If the cost is
   approximated by the RTL complexity, including CONST_COSTS above, as
   is usually the case for CISC machines, this macro should not be defined.
   For aggressively RISCy machines, only one insn format is allowed, so
   this macro should be a constant.  The value of this macro only matters
   for valid addresses.  */
/* #define ADDRESS_COST(RTX) 1 */

/* A C expression returning the cost of moving data from a register of class
   CLASS1 to one of CLASS2.
   A value != 2 allows the reload pass to verify the constraints.  */
#define REGISTER_MOVE_COST(CLASS1, CLASS2) 4

/* A C expressions returning the cost of moving data of MODE from
   a register to or from memory.  */
#define MEMORY_MOVE_COST(MODE) 8

/* Specify the cost of a branch insn; roughly the number of extra insns that
   should be added to avoid a branch.  */
#define BRANCH_COST 4

/* Nonzero if access to memory by bytes is no faster than for words.
   Also non-zero if doing byte operations (specifically shifts) in registers
   is undesirable.  */
#define SLOW_BYTE_ACCESS 1

/* Define this if addresses of constant functions
   shouldn't be put through pseudo regs where they can be cse'd.  */
#define NO_FUNCTION_CSE

/* 6.14 Dividing the Output into Sections (Texts, Data, ...).  */

/* Output before instructions.  */
#define TEXT_SECTION_ASM_OP "\n.section\t.text"

/* Output before writable data.  */
#define DATA_SECTION_ASM_OP "\n.section\t.data"

/* Output before init segment.  */
#define INIT_SECTION_ASM_OP "\n.section\t.init"

/* Output before fini segment.  */
#define FINI_SECTION_ASM_OP "\n.section\t.fini"

/* Output before read-only data.  */
#define CONST_SECTION_ASM_OP "\n.section\t.const"

/* Output before uninitialised data.  */
#define UDATA_SECTION_ASM_OP "\n.section\t.udata"

/* Output before constructor segment.  */
#define CTORS_SECTION_ASM_OP "\n.section\t.ctor"
/* Output before read-only data.  */
#define DTORS_SECTION_ASM_OP "\n.section\t.dtor"

#define EXTRA_SECTIONS in_const, in_udata, in_ctors, in_dtors

#define EXTRA_SECTION_FUNCTIONS                                         \
void const_section (void)                                               \
{                                                                       \
  if (in_section != in_const) {                                         \
    asm_fprintf (asm_out_file, "%s\n", CONST_SECTION_ASM_OP);           \
    in_section = in_const;                                              \
  }                                                                     \
}                                                                       \
                                                                        \
```

```
void udata_section (void)                                               \
{                                                                       \
  if (in_section != in_udata) {                                         \
    asm_fprintf (asm_out_file, "%s\n", UDATA_SECTION_ASM_OP);           \
    in_section = in_udata;                                              \
  }                                                                     \
}                                                                       \
                                                                        \
void ctors_section (void)                                               \
{                                                                       \
  if (in_section != in_ctors) {                                         \
    asm_fprintf (asm_out_file, "%s\n", CTORS_SECTION_ASM_OP);           \
    in_section = in_ctors;                                              \
  }                                                                     \
}                                                                       \
                                                                        \
void dtors_section (void)                                               \
{                                                                       \
  if (in_section != in_dtors) {                                         \
    asm_fprintf (asm_out_file, "%s\n", DTORS_SECTION_ASM_OP);           \
    in_section = in_dtors;                                              \
  }                                                                     \
}

/* Define the name of our readonly data section.  */
#define READONLY_DATA_SECTION const_section

/* 6.15: Position Independent Code.  */

/* We don't need to care about position independent code, as
   no one can access our address space.  */

/* 6.16: Defining the Output Assembler Language.  */

/* Output at beginning of assembler file.  */
#define ASM_FILE_START(FILE)                                            \
  {                                                                     \
    output_file_directive (FILE, main_input_filename);                  \
    text_section ();                                                    \
  }

/* Output at end of assembler file.  */
#undef ASM_FILE_END(FILE)

/* It is useless to make the output of gcc identifiable on this machine.  */
#define ASM_IDENTIFY_GCC(FILE)

/* String containing the assembler's comment-starter.  */
#define ASM_COMMENT_START ";"

/* Output to assembler file text saying following lines
   may contain character constants, extra white space, comments, etc.  */
#define ASM_APP_ON ""

/* Output to assembler file text saying following lines
   no longer contain unusual constructs.  */
#define ASM_APP_OFF ""

/* We use this to output line number instructions to aid debugging.
   Line numbers are output only if BOTH -g (of course with a debugging level
   of at least 2) and -mlineno are given.  */
#define ASM_OUTPUT_SOURCE_LINE(F, LINE)                                 \
  do {                                                                  \
    if (TARGET_LINENO)                                                  \
      asm_fprintf (F, "\tsetl\t%I%u\n", LINE);                          \
  } while (0)

/* Switch to a generic section.  */
```

```
#define ASM_OUTPUT_SECTION_NAME(FILE, DECL, NAME, RELOC)                \
  asm_fprintf (FILE, ".section\t%s", NAME)

/* This is how to output an assembler line defining a `double' constant.  */
#define ASM_OUTPUT_DOUBLE(FILE, VALUE)                                  \
  do {                                                                  \
    unsigned long l;                                                    \
    REAL_VALUE_TO_TARGET_SINGLE (VALUE, l);                             \
    asm_fprintf (FILE, "\t%s\t$%lx\n", ASM_INT_OP, l);                  \
  } while (0)

/* This is how to output an assembler line defining a `float' constant.  */
#define ASM_OUTPUT_FLOAT(FILE, VALUE)                                   \
  do {                                                                  \
    unsigned long l;                                                    \
    REAL_VALUE_TO_TARGET_SINGLE (VALUE, l);                             \
    asm_fprintf (FILE, "\t%s\t$%lx\n", ASM_INT_OP, l);                  \
  } while (0)

/* This is how to output an assembler line defining an `int' constant.  */
#define ASM_OUTPUT_INT(FILE, VALUE)                                     \
  do {                                                                  \
    asm_fprintf (FILE, "\t%s\t", ASM_INT_OP);                           \
    output_addr_const (FILE, (VALUE));                                  \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* This is how to output an assembler line defining a `short' constant.  */
#define ASM_OUTPUT_SHORT(FILE, VALUE)                                   \
  do {                                                                  \
    asm_fprintf (FILE, "\t%s\t", ASM_BYTE_OP);                          \
    output_addr_const (FILE, (VALUE));                                  \
    asm_fprintf (FILE, "&255,(");                                       \
    output_addr_const (FILE, (VALUE));                                  \
    asm_fprintf (FILE, "@-8)&255\n");                                   \
  } while (0)

/* This is how to output an assembler line defining a `char' constant.  */
#define ASM_OUTPUT_CHAR(FILE, VALUE)                                    \
  do {                                                                  \
    asm_fprintf (FILE, "\t%s\t", ASM_BYTE_OP);                          \
    output_addr_const (FILE, (VALUE));                                  \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* This is how to output an assembler line for a numeric constant byte.  */
#define ASM_OUTPUT_BYTE(FILE, VALUE)                                    \
  asm_fprintf (FILE, "\t%s\t$%x\n", ASM_BYTE_OP, (VALUE))

/* Assembler pseudo to introduce `byte'/`char' constants.  */
#define ASM_BYTE_OP "byte"

/* Assembler pseudo to introduce `int'/`long' constants.  */
#define ASM_INT_OP "word"


/* This is how to output an assembler line defining a string constant.  */
#define ASM_OUTPUT_ASCII(FILE, PTR, LEN)                                \
  do {                                                                  \
    int i = 0;                                                          \
    int currlen = 0;                                                    \
    int quotes = 0;     /* are we writing ASCII? */                     \
    unsigned char c;                                                    \
    asm_fprintf (FILE, "\t%s\t", ASM_BYTE_OP);                          \
    while (i < (LEN)) {                                                 \
      c = (PTR)[i++] & 0377;                                            \
      if (c >= 32 && c <= 126) {                                        \
        if (!quotes) {                                                  \
```

```
          quotes = 1;                                                   \
          if (currlen != 0) {                                           \
            asm_fprintf (FILE, ",\"");                                  \
            currlen += 2;                                               \
          } else {                                                      \
            asm_fprintf (FILE, "\"");                                   \
            currlen++;                                                  \
          }                                                             \
        }                                                               \
        asm_fprintf (FILE, "%c", c);                                    \
        currlen++;                                                      \
        if (c == '"') {                                                 \
          asm_fprintf (FILE, "\"");  /* two double quotes give one */   \
          currlen++;                                                    \
        }                                                               \
        if (currlen > 60 && i < (LEN)) {                                \
          asm_fprintf (FILE, "\"\n\t%s\t", ASM_BYTE_OP);                \
          currlen = 0;                                                  \
          quotes = 0;                                                   \
        }                                                               \
      } else {                                                          \
        if (quotes) {                                                   \
          quotes = 0;                                                   \
          asm_fprintf (FILE, "\"");                                     \
          currlen++;                                                    \
        }                                                               \
        if (currlen != 0) {                                             \
          asm_fprintf (FILE, ",");                                      \
          currlen++;                                                    \
        }                                                               \
        asm_fprintf (FILE, "%i", c);                                    \
        if (c < 10)                                                     \
          currlen += 1;                                                 \
        else if (c < 100)                                               \
          currlen += 2;                                                 \
        else                                                            \
          currlen += 3;                                                 \
        if (currlen > 60 && i < (LEN)) {                                \
          asm_fprintf (FILE, "\n\t%s\t", ASM_BYTE_OP);                  \
          currlen = 0;                                                  \
        }                                                               \
      }                                                                 \
    }                                                                   \
    if (quotes) {                                                       \
      asm_fprintf (FILE, "\"");                                         \
    }                                                                   \
    if (currlen != 0) {                                                 \
      asm_fprintf(FILE, "\n");                                          \
    }                                                                   \
  } while (0)

/* The logical line separator for the assembler.  */
#define IS_ASM_LOGICAL_LINE_SEPARATOR(C) 0

/* Define the parentheses used to group arithmetic operations
   in assembler code.  */
#define ASM_OPEN_PAREN "("
#define ASM_CLOSE_PAREN ")"

/* This says how to output an assembler line
   to define a global common symbol.  */
#define ASM_OUTPUT_COMMON(FILE, NAME, SIZE, ROUNDED)                    \
  do {                                                                  \
    udata_section ();                                                   \
    assemble_name (FILE, (NAME));                                       \
    asm_fprintf (FILE, ":\t.common\t%u\n", (SIZE));                     \
  } while (0)
```

```
/* This says how to output an assembler line
   to define a local common symbol.  */
#define ASM_OUTPUT_LOCAL(FILE, NAME, SIZE, ROUNDED)                    \
  do {                                                                 \
    udata_section ();                                                  \
    assemble_name (FILE, (NAME));                                      \
    asm_fprintf (FILE, ":\t.reserv\t%u\n", (SIZE));                    \
  } while (0)

/* This is how to output the definition of a user-level label named NAME,
   such as the label on a static function or variable NAME.  */
#define ASM_OUTPUT_LABEL(FILE, NAME)                                   \
  do {                                                                 \
    assemble_name (FILE, (NAME));                                      \
    asm_fprintf (FILE, ":\n");                                         \
  } while (0)

/* This is how to output a command to make the user-level label named NAME
   defined for reference from other files.  */
#define ASM_GLOBALIZE_LABEL(FILE, NAME)                                \
  do {                                                                 \
    asm_fprintf (FILE, "\n\t.global\t");                               \
    assemble_name (FILE, NAME);                                        \
    asm_fprintf (FILE, "\n");                                          \
  } while (0)

/* A C statement to output to the stdio stream any text necessary
   for declaring the name of an external symbol named name which
   is referenced in this compilation but not defined.  */
#define ASM_OUTPUT_EXTERNAL(FILE, DECL, NAME)

/* A C statement to output on stream an assembler pseudo-op to
   declare a library function named external.  */
#define ASM_OUTPUT_EXTERNAL_LIBCALL(FILE, FUN)

/* This is how to output a reference to a user-level label named NAME.
   'assemble_name' uses this.  */
#define ASM_OUTPUT_LABELREF(FILE, NAME)                                \
  asm_fprintf (FILE, "%U%s", (NAME))

/* This is how to output an internal numbered label where
   PREFIX is the class of label and NUM is the number within the class.  */
#define ASM_OUTPUT_INTERNAL_LABEL(FILE, PREFIX, NUM)                   \
  asm_fprintf (FILE, "L%s%u:\n", PREFIX, NUM)

/* This is how to store into the string LABEL
   the symbol_ref name of an internal numbered label where
   PREFIX is the class of label and NUM is the number within the class.
   This is suitable for output with 'assemble_name'.  */
#define ASM_GENERATE_INTERNAL_LABEL(LABEL, PREFIX, NUM)                \
  sprintf (LABEL, "*L%s%u", PREFIX, NUM)

/* Store in OUTPUT a string (made with alloca) containing
   an assembler-name for a local static variable named NAME.
   LABELNO is an integer which is different for each call.  */
#define ASM_FORMAT_PRIVATE_NAME(OUTPUT, NAME, LABELNO)                 \
  do {                                                                 \
    (OUTPUT) = (char *) alloca (strlen ((NAME)) + 10);                 \
    sprintf ((OUTPUT), "*S%u%s%s", (LABELNO), USER_LABEL_PREFIX,       \
                       (NAME));                                        \
  } while (0)

#define SET_ASM_OP "="

/* This is how we tell the assembler to equate two values.  */
#define ASM_OUTPUT_DEF(FILE, LABEL1, LABEL2)                           \
  do {                                                                 \
    assemble_name (FILE, LABEL1);                                      \
```

```
    asm_fprintf (FILE, "\t%s\t", SET_ASM_OP);                           \
    assemble_name (FILE, LABEL2);                                       \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* The object format supports arbitrary sections.  */
#define HAS_INIT_SECTION

/* A C statement (sans semicolon) to output an element in the table of
   global constructors.  */
#define ASM_OUTPUT_CONSTRUCTOR(FILE, NAME)                              \
  do {                                                                  \
    ctors_section ();                                                   \
    asm_fprintf (FILE, "\t%s\t ", ASM_INT_OP);                          \
    assemble_name (FILE, NAME);                                         \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* A C statement (sans semicolon) to output an element in the table of
   global destructors.  */
#define ASM_OUTPUT_DESTRUCTOR(FILE, NAME)                               \
  do {                                                                  \
    dtors_section ();                                                   \
    asm_fprintf (FILE, "\t%s\t ", ASM_INT_OP);                          \
    assemble_name (FILE, NAME);                                         \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* How to refer to registers in assembler output.
   This sequence is indexed by compiler's hard-register-number (see above).  */
#define REGISTER_NAMES {"a", "ax", "i1", "i2", "i3", "i0"}

/* Print operand X (an rtx) in assembler syntax to file FILE.
   CODE is a letter or dot ('z' in '%z0') or 0 if no letter was specified.
   For '%' followed by punctuation, CODE is the punctuation and X is null.  */
#define PRINT_OPERAND(FILE, X, CODE)                                    \
  print_operand (FILE, X, CODE)

/* Define which CODE values are valid.  */
#define PRINT_OPERAND_PUNCT_VALID_P(CODE) 0

/* Print a memory address as an operand to reference that memory location.  */
#define PRINT_OPERAND_ADDRESS(FILE, ADDR)                               \
  print_operand_address (FILE, ADDR)

/* The prefix for register names.  Note that REGISTER_NAMES
   is supposed to include this prefix.  */
#define REGISTER_PREFIX ""

/* The prefix for local labels.  You should be able to define this as
   an empty string, or any arbitrary string (such as ".", ".L%", etc)
   without having to make any other changes to account for the specific
   definition.  Note it is a string literal, not interpreted by printf
   and friends.  */
#define LOCAL_LABEL_PREFIX "L"

/* The prefix to add to user-visible assembler symbols.  */
#define USER_LABEL_PREFIX "_"

/* The prefix for immediate operands.  */
#define IMMEDIATE_PREFIX "#"

/* This is how to output an insn to push a register on the stack.
   It need not be very fast code.  */
#define ASM_OUTPUT_REG_PUSH(FILE, REGNO)                                \
  do {                                                                  \
    asm_fprintf (FILE, "\taddi\t%s,%I%u\n",                             \
                 reg_names[STACK_POINTER_REGNUM], UNITS_PER_WORD);\
```

```
    if (REGNO == REG_A || REGNO == REG_AX)                              \
      asm_fprintf (FILE, "\tst%s\t(%Rcx)[%s]\n", reg_names[REGNO],      \
                         reg_names[STACK_POINTER_REGNUM]);              \
    else                                                                \
      asm_fprintf (FILE, "\tsti\t%s,(%Rcx)[%s]\n", reg_names[REGNO],    \
                         reg_names[STACK_POINTER_REGNUM]);              \
  } while (0);

/* This is how to output an insn to pop a register from the stack.
   It need not be very fast code.  */
#define ASM_OUTPUT_REG_POP(FILE, REGNO)                                 \
  do {                                                                  \
    if (REGNO == REG_A || REGNO == REG_AX)                              \
      asm_fprintf (FILE, "\tld%s\t(%Rcx)[%s]\n", reg_names[REGNO],      \
                         reg_names[STACK_POINTER_REGNUM]);              \
    else                                                                \
      asm_fprintf (FILE, "\tldi\t%s,(%Rcx)[%s]\n", reg_names[REGNO],    \
                         reg_names[STACK_POINTER_REGNUM]);              \
    asm_fprintf (FILE, "\tsubi\t%s,%I%u\n",                             \
                       reg_names[STACK_POINTER_REGNUM], UNITS_PER_WORD);\
  } while (0);

/* This is how to output an element of a case-vector that is absolute.  */
#define ASM_OUTPUT_ADDR_VEC_ELT(FILE, VALUE)                            \
  do {                                                                  \
    char label[30];                                                     \
    ASM_GENERATE_INTERNAL_LABEL (label, "L", VALUE);                    \
    asm_fprintf (FILE, "\t%s\t", ASM_INT_OP);                           \
    assemble_name (FILE, label);                                        \
    asm_fprintf (FILE, "\n");                                           \
  } while (0)

/* This is how to output an element of a case-vector that is relative.  */
#define ASM_OUTPUT_ADDR_DIFF_ELT(FILE, VALUE, REL) abort ()

/* Output a series of zeroes.  */
#define ASM_OUTPUT_SKIP(FILE, SIZE)                                     \
  do {                                                                  \
    int i, lines = (SIZE) / 24;                                         \
    int rem = (SIZE) % 24;                                              \
    for (i = 0; i < lines; i++) {                                       \
      asm_fprintf (FILE,                                                \
        "\t%s\t0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\n",      \
        ASM_BYTE_OP);                                                   \
    }                                                                   \
    if (rem > 0) {                                                      \
      asm_fprintf (FILE, "\t%s\t0", ASM_BYTE_OP);                       \
      for (i = 0; i < rem-1; i++) {                                     \
        asm_fprintf (FILE, ",0");                                       \
      }                                                                 \
      asm_fprintf (FILE, "\n");                                         \
    }                                                                   \
  } while (0)

/* This is how to output an assembler line
   that says to advance the location counter
   to a multiple of 2**LOG bytes.  */
#define ASM_OUTPUT_ALIGN(FILE, LOG)                                     \
  do {                                                                  \
    if ((LOG) != 0) {                                                   \
      if ((LOG) == 2)                                                   \
        asm_fprintf (FILE, "\talign\n");                                \
      else                                                              \
        fatal ("The assembler only supports alignment to 32 bit\n"      \
               "boundaries.  Fix the assembler and config/mpc/mpc.h\n"  \
               "in the GCC source tree.\n");                            \
    }                                                                   \
  } while (0)
```

```
/* Pretend to support debugging (to make gcc support -g).  */
#define SDB_DEBUGGING_INFO
#define DBX_REGISTER_NUMBER(NUMBER) (NUMBER)
#define PUT_SDB_SCL(a)
#define PUT_SDB_INT_VAL(a)
#define PUT_SDB_VAL(a)
#define PUT_SDB_DEF(a)
#define PUT_SDB_PLAIN_DEF(a)
#define PUT_SDB_ENDEF
#define PUT_SDB_TYPE(a)
#define PUT_SDB_SIZE(a)
#define PUT_SDB_START_DIM
#define PUT_SDB_NEXT_DIM(a)
#define PUT_SDB_LAST_DIM(a)
#define PUT_SDB_TAG(a)
#define PUT_SDB_BLOCK_START(a)
#define PUT_SDB_BLOCK_END(a)
#define PUT_SDB_FUNCTION_START(a)
#define PUT_SDB_FUNCTION_END(a)
#define PUT_SDB_EPILOGUE_END(a)

/* 6.18 Cross Compilation and Floating Point.  */

/* Define to enable software floating point emulation.  */
/* Currently disabled for native compilers, because the floating point
   emulation code doesn't work if sizeof(short) != 2.  This should not
   be a problem, as native compilers should know how to deal with the
   machine's own floating point format.  */
#ifdef CROSS_COMPILE
#define REAL_ARITHMETIC
#else
#define REAL_VALUE_ATOF(x, s) strtod((x), (char **)0)
#endif

/* 6.19 Miscellaneous Parameters.  */

/* Specify the machine mode that this machine uses
   for the index in the tablejump instruction.  */
#define CASE_VECTOR_MODE Pmode

/* Define this if the tablejump instruction expects the table
   to contain offsets from the address of the table.
   Do not define this if the table should contain absolute addresses.  */
#undef CASE_VECTOR_PC_RELATIVE

/* Define if operations between registers always perform the operation
   on the full register even if a narrower mode is specified.  */
#define WORD_REGISTER_OPERATIONS

/* Define if loading in MODE, an integral mode narrower than BITS_PER_WORD
   will either zero-extend or sign-extend.  The value of this macro should
   be the code that says which one of the two operations is implicitly
   done, NIL if none.  */
#define LOAD_EXTEND_OP(MODE) ZERO_EXTEND

/* FIXME: Specify the tree operation to be used to convert reals to integers.  */
#define IMPLICIT_FIX_EXPR FIX_ROUND_EXPR

/* This is the kind of divide that is easiest to do in the general case.  */
#define EASY_DIV_EXPR FLOOR_DIV_EXPR

/* Max number of bytes we can move to or from memory
   in one reasonably fast instruction.  */
#define MOVE_MAX 4

/* Define this to be nonzero if shift instructions ignore all but the low-order
   few bits.  */
```

```
#define SHIFT_COUNT_TRUNCATED 0

/* Value is 1 if truncating an integer of INPREC bits to OUTPREC bits
   is done just by pretending it is already truncated.  */
#define TRULY_NOOP_TRUNCATION(OUTPREC, INPREC) 1

/* We assume that the store-condition-codes instructions store 0 for false
   and some other value for true.  This is the value stored for true.  */
#define STORE_FLAG_VALUE 1

/* Specify the machine mode that pointers have.
   After generation of rtl, the compiler makes no further distinction
   between pointers and any other objects of this machine mode.  */
#define Pmode SImode

/* A function address in a call instruction is a byte address (for indexing
   purposes) so give the MEM rtx a byte's mode.  */
#define FUNCTION_MODE QImode

/* The system headers are C++-aware.  */
#define NO_IMPLICIT_EXTERN_C

/* We define this to 0 so that gcc will never accept a dollar sign in a
   variable name.  This is needed because the assembler will not accept
   dollar signs.  */
#define DOLLARS_IN_IDENTIFIERS 0

/* We define this to prevent the name mangler from putting dollar signs into
   function names.  */
#define NO_DOLLAR_IN_LABEL

/* We define this to prevent the name mangler from putting dots into
   function names.  */
#define NO_DOT_IN_LABEL

/* Use atexit for static constructors/destructors, instead of defining
   our own exit function.  */
#define HAVE_ATEXIT
```

D.1.3 C-Unterprogramme `mpc.c`

```
/* Subroutines for insn-output.c for MONADS-PC.
   Written by Klaus Espenlaub (kespenla@student.informatik.uni-ulm.de).

This file is is derived from software covered by the terms of the
GNU General Public License as published by the Free Software Foundation.
Consequently this file is subject to these terms.  */

#include "config.h"
#include <stdio.h>
#include "rtl.h"
#include "regs.h"
#include "hard-reg-set.h"
#include "real.h"
#include "insn-config.h"
#include "conditions.h"
#include "insn-flags.h"
#include "output.h"
#include "insn-attr.h"
#include "flags.h"
#include "recog.h"
#include "reload.h"
#include "expr.h"
#include "obstack.h"
#include "tree.h"
```

```
/* This function generates the assembly code for function entry.
   STREAM is a stdio stream to output the code to.
   SIZE is an int: how many units of temporary storage to allocate.
   Refer to the array 'regs_ever_live' to determine which registers
   to save; 'regs_ever_live[I]' is nonzero if register number I
   is ever used in the function.  This function is responsible for
   knowing which registers should not be saved even if used.  */

void
function_prologue (stream, size)
     FILE *stream;
     int size;
{
  int frame_size;

  frame_size = compute_frame_size (size);
  asm_fprintf (stream, "; prologue, locals: %i, outgoing: %i\n",
                       size, current_function_outgoing_args_size);
  /* save new frame pointer value (if needed).  */
  if (frame_pointer_needed) {
    asm_fprintf (stream, "\tldi\t%s,%s\n", reg_names[REG_I1],
                         reg_names[STACK_POINTER_REGNUM]);
  }
  asm_fprintf (stream, "\taddi\t%s,%I%u\n", reg_names[STACK_POINTER_REGNUM],
                       frame_size);
  /* save return address.  */
  /* FIXME: can be further optimised in the case of leaf funktions,
     especially when AX is never used.  */
  asm_fprintf (stream, "\tst%s\t%i(%Rcx)[%s]\n", reg_names[REG_AX],
                       -frame_size, reg_names[STACK_POINTER_REGNUM]);
  if (frame_pointer_needed || regs_ever_live[FRAME_POINTER_REGNUM]) {
    /* save frame pointer.  */
    asm_fprintf (stream, "\tsti\t%s,%i(%Rcx)[%s]\n",
                         reg_names[FRAME_POINTER_REGNUM],
                         -frame_size+UNITS_PER_WORD,
                         reg_names[STACK_POINTER_REGNUM]);
  }

  if (frame_pointer_needed) {
    asm_fprintf (stream, "\tldi\t%s,%s\n", reg_names[FRAME_POINTER_REGNUM],
                         reg_names[REG_I1]);
  }
  asm_fprintf (stream, "; end prologue\n");
}


/* This function generates the assembly code for function exit,
   on machines that need it.  Args are same as for FUNCTION_PROLOGUE.

   The function epilogue should not depend on the current stack pointer!
   It should use the frame pointer only, if there is a frame pointer.
   This is mandatory because of alloca; we also take advantage of it to
   omit stack adjustments before returning.  */

void
function_epilogue (stream, size)
     FILE *stream;
     int size;
{
  int frame_size;

  frame_size = compute_frame_size (size);
  asm_fprintf (stream, "; epilogue\n");
  if (frame_pointer_needed) {
    /* set the stack pointer to where it was after the prologue */
    asm_fprintf (stream, "\taddi\t%s,%I%u\n", reg_names[FRAME_POINTER_REGNUM],
                         frame_size);
    asm_fprintf (stream, "\tldi\t%s,%s\n", reg_names[STACK_POINTER_REGNUM],
```

```
                        reg_names[FRAME_POINTER_REGNUM]);
  }
  /* we now can use the stack pointer like the frame pointer, so it
     doesn't matter when the frame pointer gets restored.  */
  /* get return address.  */
  asm_fprintf (stream, "\tld%s\t%i(%Rcx)[%s]\n", reg_names[REG_A],
                       -frame_size, reg_names[STACK_POINTER_REGNUM]);
  if (frame_pointer_needed || regs_ever_live[FRAME_POINTER_REGNUM]) {
    /* restore the previous frame pointer.  */
    asm_fprintf (stream, "\tldi\t%s,%i(%Rcx)[%s]\n",
                         reg_names[FRAME_POINTER_REGNUM],
                         -frame_size+UNITS_PER_WORD,
                         reg_names[STACK_POINTER_REGNUM]);
  }
  asm_fprintf (stream, "\tsubi\t%s,%I%d\n", reg_names[STACK_POINTER_REGNUM],
                       frame_size);
  asm_fprintf (stream, "\tjump\t%s\n", reg_names[REG_A]);
  asm_fprintf (stream, "; end epilogue\n");
}

/* Compute the frame size required by the function.  This function is called
   during the reload pass and possibly in some other places, too.  */
int
compute_frame_size (size)
     int size;
{
  return size + current_function_outgoing_args_size + 2 * UNITS_PER_WORD;
}

/* FIXME: this is only partially implemented.  This causes suboptimal code
   to be emitted, as the compiler is forced to throw away the knowledge
   about the cc's too often.  */
void
notice_update_cc (exp)
     rtx exp;
{
  if (GET_CODE (exp) == SET) {
    /* Jumps do not alter the cc's.  */
    if (SET_DEST (exp) == pc_rtx)
      return;
    /* Function calls clobber the cc's.  */
    if (GET_CODE (SET_SRC (exp)) == CALL) {
      CC_STATUS_INIT;
      return;
    }
    /* Tests and compares set the cc's in predictable ways.  */
    if (SET_DEST (exp) == cc0_rtx) {
      cc_status.flags = 0;
      cc_status.value1 = SET_DEST (exp);
      cc_status.value2 = SET_SRC (exp);
      return;
    }
    /* all other SET are assumed to clobber the CC's.  This is not
       the whole truth, but keeps the compiler from emitting wrong code.  */
    CC_STATUS_INIT;
  } else {
    CC_STATUS_INIT;
  }
}

/* A C compound statement to output to stdio stream STREAM the
   assembler syntax for an instruction operand OP.  OP is an RTL
   expression.

   LETTER is a value that can be used to specify one of several ways
   of printing the operand.  It is used when identical operands
   must be printed differently depending on the context.  LETTER
   comes from the '%' specification that was used to request
```

```
   printing of the operand.  If the specification was just '%DIGIT'
   then LETTER is 0; if the specification was '%LTR DIGIT' then LETTER
   is the ASCII code for LTR.

   If OP is a register, this macro should print the register's name.
   The names can be found in an array 'reg_names' whose type is
   'char *[]'.  'reg_names' is initialized from 'REGISTER_NAMES'.

   When the machine description has a specification '%PUNCT' (a '%'
   followed by a punctuation character), this macro is called with
   a null pointer for OP and the punctuation character for LETTER.  */

void
print_operand (file, op, letter)
     FILE *file;                /* file to write to */
     rtx op;                    /* operand to print */
     int letter;                /* %<letter> or 0 */
{
  rtx memexpr;
  rtx offset, breg;
  REAL_VALUE_TYPE r;
  long l;

  switch (letter) {
  case 0:       /* no special letter */
    switch (GET_CODE (op)) {
    case REG:
      asm_fprintf (file, "%s", reg_names[REGNO (op)]);
      break;
    case MEM:
      if (CONSTANT_ADDRESS_P (XEXP (op, 0))) {
        output_addr_const (file, XEXP (op, 0));
        asm_fprintf (file, "(%Rcx)");
      } else {
        memexpr = XEXP (op, 0);
        switch (GET_CODE (memexpr)) {
        case REG:
          if (!REGNO_OK_FOR_BASE_P (REGNO (memexpr))) {
            abort ();
          }
          asm_fprintf (file, "(%Rcx)[%s]", reg_names[REGNO (memexpr)]);
          break;
        case PLUS:
          if (CONSTANT_ADDRESS_P (XEXP (memexpr, 0))) {
            offset = XEXP (memexpr, 0);
            breg = XEXP (memexpr, 1);
          } else if (CONSTANT_ADDRESS_P (XEXP (memexpr, 1))) {
            offset = XEXP (memexpr, 1);
            breg = XEXP (memexpr, 0);
          } else abort ();
          if (GET_CODE (breg) != REG || !REGNO_OK_FOR_BASE_P (REGNO (breg))) {
            abort ();
          }
          output_addr_const (file, offset);
          asm_fprintf (file, "(%Rcx)[%s]", reg_names[REGNO (breg)]);
          break;
        default:
          fprintf (stderr, "letter: %c\n", letter);
          print_rtl (stderr, op);
          fprintf (stderr, "\n");
          abort ();
        }
      }
      break;
    case CONST:
      memexpr = XEXP (op, 0);
      if (GET_CODE (memexpr) == PLUS) {
        asm_fprintf (file, "%I");
```

```
        output_addr_const (file, memexpr);
      } else {
        abort ();
      }
      break;
    case CONST_DOUBLE: /* fall through */
    case CONST_INT:
      asm_fprintf(file, "%I$%x", INTVAL(op));
      break;
    case CODE_LABEL: /* fall through */
    case LABEL_REF:  /* fall through */
    case SYMBOL_REF:
      asm_fprintf(file, "%I");
      output_addr_const (file, op);
      break;
    default:
      fprintf (stderr, "letter: %c\n", letter);
      print_rtl (stderr, op);
      fprintf (stderr, "\n");
      abort ();
    }
    break;
  default:
    fprintf (stderr, "letter: %c\n", letter);
    print_rtl (stderr, op);
    fprintf (stderr, "\n");
    abort ();
  }
}

/* A C compound statement to output to stdio stream STREAM the
   assembler syntax for an instruction operand that is a memory
   reference whose address is ADDR.  ADDR is an RTL expression.  */

void
print_operand_address (file, addr)
     FILE *file;
     rtx addr;
{
  rtx memexpr;
  rtx offset, breg;

  switch (GET_CODE (addr)) {
  case REG:
    asm_fprintf (file, "%s", reg_names[REGNO (addr)]);
    break;
  case MEM:
    if (CONSTANT_ADDRESS_P (XEXP (addr, 0))) {
      asm_fprintf (file, "%I");
      output_addr_const (file, XEXP (addr, 0));
    } else {
      memexpr = XEXP (addr, 0);
      switch (GET_CODE (memexpr)) {
      case REG:
        /* This case is somehow strange - from inspection it has to be
           implemented as it is, but why doesn't GCC use (REG...) then? */
        asm_fprintf (file, "%s", reg_names[REGNO (memexpr)]);
        break;
      case PLUS:
        if (CONSTANT_ADDRESS_P (XEXP (memexpr, 0))) {
          offset = XEXP (memexpr, 0);
          breg = XEXP (memexpr, 1);
        } else if (CONSTANT_ADDRESS_P (XEXP (memexpr, 1))) {
          offset = XEXP (memexpr, 1);
          breg = XEXP (memexpr, 0);
        } else abort ();
        if (GET_CODE (breg) != REG || !REGNO_OK_FOR_BASE_P (REGNO (breg))) {
          abort ();
```

```
        }
        output_addr_const (file, offset);
        asm_fprintf (file, "(%Rcx)[%s]", reg_names[REGNO (breg)]);
        break;
      default:
        print_rtl (stderr, addr);
        fprintf (stderr, "\n");
        abort ();
      }
    }
    break;
  case CODE_LABEL: /* fall through */
  case LABEL_REF:
    asm_fprintf(file, "%I");
    output_addr_const (file, addr);
    break;
  default:
    fprintf (stderr, "%s\n", GET_RTX_NAME (GET_CODE (addr)));
    print_rtl (stderr, addr);
    fprintf (stderr, "\n");
    abort ();
  }
}

/* Predicates for 'match_operand'.  */

/* s_register_operand is the same as register_operand, but it doesn't accept
   (SUBREG (MEM)...).  */
int
s_register_operand (op, mode)
     register rtx op;
     enum machine_mode mode;
{
  if (GET_MODE (op) != mode && mode != VOIDmode)
    return 0;

  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);

  /* We don't consider registers whose class is NO_REGS
     to be a register operand.  */
  return (GET_CODE (op) == REG
          && (REGNO (op) >= FIRST_PSEUDO_REGISTER
              || REGNO_REG_CLASS (REGNO (op)) != NO_REGS));
}

/* Return 1 if OP is an item in memory, given that we are in reload.  */
int
reload_memory_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  int regno = true_regnum (op);

  return (! CONSTANT_P (op)
          && (regno == -1
              || (GET_CODE (op) == REG
                  && REGNO (op) >= FIRST_PSEUDO_REGISTER)));
}

/* Return 1 if OP is a general operand, but no widened expression from
   a smaller mode  using subreg is allowed.  */
int
nowiden_general_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return general_operand (op, mode) &&
```

```
         ((GET_CODE (op) != SUBREG) ||
          (GET_MODE_SIZE (GET_MODE (SUBREG_REG (op))) >= GET_MODE_SIZE (mode)));
}

enum reg_class
mpc_secondary_reload_class (class, mode, x)
     enum reg_class class;
     enum machine_mode mode;
     rtx x;
{
  int regno = true_regnum (x);

  if (regno >= FIRST_PSEUDO_REGISTER) {
    regno = -1;
  }

  /* Moving stuff between registers is no problem.  */
  if (regno >= 0)
    return NO_REGS;

  /* 32 bit stuff and BLKmode works right away.  */
  if (mode == SImode || mode == SFmode || mode == BLKmode)
    return NO_REGS;

  /* QImode objects are possible for classes that don't contain AX.  */
  if (mode == QImode && ((class == AREG) || (class == INDEX_REGS) ||
                         (class == A_INDEX_REGS)))
    return NO_REGS;

  /* Otherwise we need a temporary that can be used for byte accesses.  */
  return A_INDEX_REGS;
}

rtx
mpc_get_address(ref, extra_offset)
     rtx ref;
     int extra_offset;
{
  rtx base;
  HOST_WIDE_INT offset = 0;

  if (GET_CODE (ref) == SUBREG) {
    offset = SUBREG_WORD (ref) * UNITS_PER_WORD;
    ref = SUBREG_REG (ref);
  }

  if (GET_CODE (ref) == REG) {
    ref = reg_equiv_mem[REGNO (ref)];
  }

  if (reload_in_progress) {
    base = find_replacement (&XEXP (ref, 0));
  } else {
    base = XEXP (ref, 0);
  }

  if (GET_CODE (base) == PLUS) {
    offset += INTVAL (XEXP (base, 1));
    base = XEXP (base, 0);
  }
  return plus_constant (base, offset + extra_offset);
}

/* Subroutines needed for moving/reloading of HImode entities.  */
void
mpc_reload_inhi (dest, src, temp)
     rtx dest;
     rtx src;
```

```
     rtx temp;
{
  /* load upper byte into temporary.  */
  emit_insn (gen_zero_extendqisi2 (temp,
                                   gen_rtx (MEM, QImode,
                                            mpc_get_address (src, 1))));
  /* shift upper byte into the right place.  */
  emit_insn (gen_rtx (SET, VOIDmode, temp,
                                     gen_rtx (ASHIFT, SImode, temp,
                                                      GEN_INT (8))));
  /* load lower byte into the result register.  */
  emit_insn (gen_zero_extendqisi2 (gen_rtx (SUBREG, SImode, dest, 0),
                                   gen_rtx (MEM, QImode,
                                            mpc_get_address (src, 0))));
  /* combine the temporary and the intermediate result to the final result.  */
  emit_insn (gen_rtx (SET, VOIDmode, gen_rtx (SUBREG, SImode, dest, 0),
                                     gen_rtx (IOR, SImode,
                                              gen_rtx (SUBREG, SImode, dest, 0),
                                              temp)));
}

void
mpc_reload_outhi (dest, src, temp)
     rtx dest;
     rtx src;
     rtx temp;
{
  /* move value to temporary.  */
  emit_insn (gen_rtx (SET, VOIDmode, temp,
                                     gen_rtx (SUBREG, SImode, src, 0)));
  /* store lower byte.  */
  emit_insn (gen_rtx (SET, VOIDmode, gen_rtx (MEM, QImode,
                                              mpc_get_address (dest, 0)),
                                     gen_rtx (SUBREG, QImode, temp, 0)));
  /* shift higher byte into lower byte in temporary.  Uses opposite shift
     direction, see define_expand.  */
  emit_insn (gen_rtx (SET, VOIDmode, temp,
                                     gen_rtx (LSHIFTRT, SImode, temp,
                                                        GEN_INT (-8))));
  /* store upper byte.  */
  emit_insn (gen_rtx (SET, VOIDmode, gen_rtx (MEM, QImode,
                                              mpc_get_address (dest, 1)),
                                     gen_rtx (SUBREG, QImode, temp, 0)));
}
```

D.1.4 Plattformbeschreibung `xm-mpc.h`

```
/* Configuration for the GNU C-compiler, for MONADS-PC.
   Written by Klaus Espenlaub (kespenla@student.informatik.uni-ulm.de).

This file is is derived from software covered by the terms of the
GNU General Public License as published by the Free Software Foundation.
Consequently this file is subject to these terms.  */

/* #defines that need visibility everywhere.  */
#define FALSE 0
#define TRUE 1

/* This describes the machine the compiler is hosted on.  */
#define HOST_BITS_PER_CHAR      8
#define HOST_BITS_PER_SHORT     32
#define HOST_BITS_PER_INT       32
#define HOST_BITS_PER_LONG      32
#define HOST_BITS_PER_LONGLONG  64

/* Arguments to use with 'exit'.  */
```

```
#define SUCCESS_EXIT_CODE 0
#define FATAL_EXIT_CODE 33

/* If compiled with GNU C, use the built-in alloca.  */
#ifdef __GNUC__
#define alloca __builtin_alloca
#else
#define USE_C_ALLOCA
#endif

/* If we do the ANSI C library, do it right.  */
#define HAVE_VPRINTF
#define MULTIBYTE_CHARS 1
#define HAVE_PUTENV

#define MD_CALL_PROTOTYPES

/* target machine dependencies.
   tm.h is a symbolic link to the actual target specific file.   */
#include "tm.h"
```

D.1.5 Makefile-Regeln `t-mpc`

```
# Makefile fragment for the MONADS-PC *target*.
# Written by Klaus Espenlaub (kespenla@student.informatik.uni-ulm.de).

# No low-level emulation functions are needed.
LIBGCC1=
CROSS_LIBGCC1=
LIBGCC1_TEST=

# Don't install "assert.h" in gcc. We use the one in glibc.
INSTALL_ASSERT_H=

# Don't run fixproto
STMP_FIXPROTO=

# we don't have DF mode, make it equal to SF mode.
TARGET_LIBGCC2_CFLAGS=-DDF=SF

# suppress the standard rules for crtbegin.o and crtend.o
T = t

# Rules for our assembly language versions of crtbegin and crtend
crtbegin.o:     $(srcdir)/config/mpc/crtbegin.s $(GCC_PASSES)
        $(GCC_FOR_TARGET) $(GCC_CFLAGS) $(INCLUDES) -o $@ -c $<

crtend.o:       $(srcdir)/config/mpc/crtend.s $(GCC_PASSES)
        $(GCC_FOR_TARGET) $(GCC_CFLAGS) $(INCLUDES) -o $@ -c $<
```

D.1.6 Makefile-Regeln `x-mpc`

Diese Datei wird hier nicht angegeben, denn sie enthält ausschließlich einen kurzen Kommentar. Es wird erwartet, daß keine speziellen Makefile-Regeln für die Übersetzung des GNU C-Compilers auf dem MONADS-PC erforderlich sind. Dies kann jedoch erst definitiv festgelegt werden, wenn der Compiler auf dem MONADS-PC selbst lauffähig ist.

D.2 Entwicklungstools für den MONADS-PC

Die folgenden Listings stellen den Code des Assemblers, Linkers, der Bibliotheksverwaltungsprogramme und den Stub-Generator vor.

D.2.1 Assembler `as`

D.2.1.1 Assembler-Hauptprogramm `as.c`

```
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>

#include "utils.h"
#include "getopt.h"
#include "as-sym.h"
#include "parser.h"


/* Version number */
#define MAJOR 1
#define MINOR 0


char *pname;            /* the name of this program */
bool nowarn = false;    /* 1=suppress warnings */
bool genoutname = true; /* 1=we need to come up with an output filename */
bool printver = false;  /* 1=print version number */
char *outname;          /* the object file we are supposed to make */
char *inname;           /* the assembler source we are processing */
char *sourceName;       /* the source language file, NULL if unknown */
symentry *symtab;       /* the table of all symbols */
sectentry *secttab;     /* list of all sections */
FILE *junkfp;           /* stuff with unknown section goes here */
sectentry *csect;       /* the current section */


static bool Assemble(void)
{
  FILE *ip;
  FILE *op;
  symentry *sym, *new;
  sectentry *sect, *nsect;
  unsigned i;

  sourceName = NULL;
  if (inname[0] == '\0') {
    ip = stdin;
  } else {
    if ((ip = fopen(inname, "r")) == NULL) {
      Error("cannot open input file `%s': %s", inname, strerror(errno));
      return false;
    }
  }
  lineNo = 1;
  lineBuffer = NULL;
  lineBufferAlloc = 0;
  yyin = ip;      /* set the scanner input stream */
  if (yyparse() == 1) {
```

```
  /* error occurred during parsing, remove output file */
  Error("syntax error in '%s'", inname);
  unlink(outname);
  fclose(ip);
  xfree(sourceName);
  xfree(lineBuffer);
  lineBuffer = NULL;
  lineBufferAlloc = 0;
  return false;
} else {
  if ((op = fopen(outname, "w")) == NULL) {
    Error("cannot open output file '%s': %s", outname, strerror(errno));
    xfree(sourceName);
    xfree(lineBuffer);
    lineBuffer = NULL;
    lineBufferAlloc = 0;
    return false;
  }

  fputs(":MPC object file 1.0\n", op);
  if (sourceName != NULL) {
    fprintf(op, ":source %s\n", sourceName);
  }
  fputs(":begin symbol\n", op);
  sym = symtab;
  while (sym != NULL) {
    if (sym->group != 'J') {
      /* place common symbols in object */
      if (sym->iscommon) {
        if (sym->type == local) {
          fprintf(sym->commSect->fp, "\talign\n");
          fprintf(sym->commSect->fp, "%s:\n", sym->name);
          fprintf(sym->commSect->fp, "\tblockb\t%u\n", sym->size);
          sym->commSect->lines += 3;
          sym->iscommon = false; /* symbol is no longer common */
        } else {
          /* leave global common symbol in the symbol table, the linker
             takes care if an actual object needs to be generated.  */
        }
      }
      fprintf(op, "%s\t%c", sym->name,
                  (sym->type == local ? tolower(sym->group) : sym->group));
      if (sym->iscommon) {
        fputc('*', op);
      }
      fprintf(op, "\t%i\n", sym->size);
    }
    new = sym->next;
    free(sym->name);
    free(sym);
    sym = new;
  }
  symtab = NULL;
  fputs(":end symbol\n", op);

  fputs(":begin section\n", op);
  sect = secttab;
  while (sect != NULL) {
    if ((sect->lines != 0) && (strcmp(sect->name, "junk") != 0)) {
      fprintf(op, "%s\t%i\n", sect->name, sect->lines);
    }
    sect = sect->next;
  }
  fputs(":end section\n", op);

  fputs(":end header\n", op);

  sect = secttab;
```

```
    while (sect != NULL) {
      if (strcmp(sect->name, "junk") != 0) {
        rewind(sect->fp);
        for (i = 0; i < sect->lines; i++) {
          ReadLine(sect->fp);
          fputs(lineBuffer, op);
        }
      }
      fclose(sect->fp);
      free(sect->name);
      nsect = sect->next;
      free(sect);
      sect = nsect;
    }

    if (fclose(op) != 0) {
      Error("closing output file '%s': %s", outname, strerror(errno));
      xfree(sourceName);
      xfree(lineBuffer);
      lineBuffer = NULL;
      lineBufferAlloc = 0;
      return false;
    }

    if (fclose(ip) != 0) {
      Error("closing input file '%s': %s", inname, strerror(errno));
      xfree(sourceName);
      xfree(lineBuffer);
      lineBuffer = NULL;
      lineBufferAlloc = 0;
      return false;
    }
    xfree(sourceName);
    xfree(lineBuffer);
    lineBuffer = NULL;
    lineBufferAlloc = 0;
    return true;
  }
}


int main(int argc, char *argv[])
{
  int c;                    /* current option */
  int errflg = 0;           /* > 0 means there are errors in the cmdline */
  int inlen;                /* length of the input filename */

  pname = argv[0];

  while ((c = getopt(argc, argv, "dno:V")) != EOF) {
    switch (c) {
      case 'd':
        yydebug = 1;
        break;
      case 'n':
        nowarn = true;
        break;
      case 'o':
        outname = optarg;
        genoutname = false;
        break;
      case 'V':
        printver = true;
        break;
      case '?':
      default:
        errflg++;
    }
```

```
  }
  /* if -o is specified, allow 0 (i.e. read from stdin) or 1 arguments,
     if it is not, allow >= 1 arguments.  */
  if ((!genoutname) && (optind != argc) && (optind != argc - 1)) ||
      (genoutname && (optind == argc)))
    errflg++;

  if (errflg || printver) {
    fprintf(stderr, "MONADS-PC pseudo-assembler version %i.%i\n", MAJOR, MINOR);
    if (errflg) {
      fprintf(stderr, "usage: %s [-nV] [-o <filename>] files...\n", pname);
      exit(2);
    }
  }

  symtab = NULL;
  secttab = NULL;
  csect = NULL;

  do {
    if (argc == optind) {
      inname = "";     /* read from stdin */
    } else {
      inname = argv[optind];
      if (genoutname) {
        inlen = strlen(inname);
        if ((inlen>2) && (inname[inlen-2] == '.') && (inname[inlen-1] == 's')) {
          outname = StrDup(inname);
          outname[inlen-1] = 'o';
        } else {
          outname = (char *) malloc(inlen+3);
          strcpy(outname, inname);
          strcat(outname, ".o");
        }
      }
    }
    junkfp = tmpfile();
    if (junkfp == NULL) {
      Error("cannot create temporary junk file: %s", strerror(errno));
      exit(2);
    }
    if (!Assemble()) {
      exit(1);
    }
    fclose(junkfp);
    junkfp = NULL;
    if (genoutname)
      free(outname);
    optind++;
  } while (optind < argc);
  return 0;
}
```

D.2.1.2 Scanner `as-lex.l`

```
/* definitions */

%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include "utils.h"
#include "as-parse.h"
#include "as-sym.h"
```

```
static unsigned StrNum(const char *);
static int ProcessReserved(const char *);
%}

%option 8bit
%option case-sensitive
%option never-interactive
%option nounput
%option noyywrap

WS      [ \t]+
LID     [\.A-Za-z_][\.0-9A-Za-z_]*
DIG     [0-9]
DNUM    {DIG}+
HEXNUM  "$"[0-9A-Fa-f]+
STR     (\"[^"\n]*\")*


%%

"+"|"-"|"*"|"/"|"%"|"\\"|"\|"|"&"|"@"   { AppendLine(yytext); return OP; }
{WS}                    { AppendLine(yytext); /* ignore whitespace */ }
{DNUM}|{HEXNUM}         { AppendLine(yytext); yylval.value = StrNum(yytext);
                          return NUM; }
{STR}                   { AppendLine(yytext); yylval.text = StrDup(yytext);
                          return STRING; }
{LID}                   { AppendLine(yytext); return ProcessReserved(yytext); }
";"[^\n]*               { AppendLine(yytext); /* ignore comments */ }
"\n"                    { AppendLine(yytext); return yytext[0]; }
<<EOF>>                 { return yytext[0]; }
.                       { AppendLine(yytext); return yytext[0]; }

%%

/* user subroutines */

static unsigned StrNum(const char *src)
{
  unsigned dest;

  if (src[0] == '$') {
    dest = strtol(&(src[1]), NULL, 16);
  } else {
    dest = strtol(src, NULL, 10);
  }
  return dest;
}

static int ProcessReserved(const char *id)
{
  if (id[0] == '.') {
    if (strcmp(id, ".global") == 0) {
      return RGLOBAL;
    } else if (strcmp(id, ".common") == 0) {
      return RCOMMON;
    } else if (strcmp(id, ".reserv") == 0) {
      return RRESERVE;
    } else if (strcmp(id, ".section") == 0) {
      return RSECTION;
    } else if (strcmp(id, ".file") == 0) {
      return RFILE;
    } else if (strcmp(id, ".text") == 0) { /* for GNU as compatibility */
      return RTEXT;
    } else {
      yylval.text = StrDup(&(yytext[1]));
      return DID;
    }
```

```
  } else if ((id[0] == 'a') && (id[1] == '\0' ||
                                (id[1] == 'x' && id[2] == '\0'))) {
    return REG;
  } else if ((id[0] == 'i') && (id[1] >= '0') && (id[1] <= '3') &&
             (id[2] == '\0')) {
    return REG;
  } else if ((id[0] == 'c') && (id[1] == 'x') && (id[2] == '\0')) {
    return REG;
  } else if ((id[0] == 'c') && isdigit((int) id[1]) &&
             ((id[2] == '\0') || (isdigit((int) id[2]) && (id[3] == '\0')))) {
    return REG;
  } else if (strcmp(id, "dummy_segment") == 0) {
    return RASMDEC;
  } else if (strcmp(id, "data_segment") == 0) {
    return RASMDEC2;
  } else if (strcmp(id, "code_segment") == 0) {
    return RASMDEC;
  } else if (strcmp(id, "interface_procedure") == 0) {
    return RASMDEC;
  } else if (strcmp(id, "data_size_of") == 0) {
    return FUNCT;
  } else if (strcmp(id, "ptr_size_of") == 0) {
    return FUNCT;
  } else {
    yylval.text = StrDup(yytext);
    return ID;
  }
}
```

D.2.1.3 Parser `as-parse.y`

```
/* MONADS-PC pseudo-assembler parser. */

%{
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#include "as-sym.h"

#define YYDEBUG 1

extern int yylex(void);
static void CopyLine(void);

void yyerror(const char *);
extern unsigned lineNo;
%}

%union {
char *text;
unsigned value;
}

%token RGLOBAL
%token RCOMMON
%token RRESERVE
%token RSECTION
%token RTEXT
%token RFILE
%token RASMDEC
%token RASMDEC2
%token <text> ID
%token <text> DID
%token REG
%token FUNCT
```

```
%token <value> NUM
%token <text> STRING
%token OP

%left OP
%right UNARY

%%

program:  input

input:    /* empty */
        | input line '\n'               { lineNo++; CopyLine(); }
;

line:     /* empty */
        | ID ':'                { if (LabelDef($1)) YYERROR; }
        | ID '=' expr           { if (LabelDef($1)) YYERROR; }
        | ID ':' RCOMMON NUM    { if (LabelDefCommon($1, $4)) YYERROR; }
        | ID ':' RRESERVE NUM   { if (LabelDefLSpace($1, $4)) YYERROR; }
        | ID opers              { /* no action, not even $$ = $1 */ }
        | RGLOBAL ID            { LabelDeclGlobal($2); }
        | RSECTION ID           { SetSection($2); }
        | RSECTION DID          { SetSection($2); }
        | RSECTION RTEXT        { SetSection("text"); /* resolve ambiguity */ }
        | RTEXT                 { SetSection("text"); /* as compatibility */ }
        | RFILE STRING          { if (SetSource($2)) YYERROR; }
        | RASMDEC ID            { if (LabelDef($2)) YYERROR; }
        | RASMDEC ID ',' opers2 { if (LabelDef($2)) YYERROR; }
        | RASMDEC2 ID ',' ID    { if (LabelDef($4)) YYERROR; }
;

opers:    /* empty */
        | oper
        | opers ',' oper
;
opers2:   oper
        | opers2 ',' oper

oper:     expr
        | '#' expr
        | '(' REG ')' '[' REG ']'       { /* expr can be empty if idm/nidm */ }
        | '(' REG ')'                   { /* expr can be empty if idm/nidm */ }
        | expr '(' REG ')' '[' REG ']'
        | expr '(' REG ')'
;

expr:     ID                            { LabelUsed($1); }
        | FUNCT '(' expr2 ')'
        | FUNCT '(' ')'
        | NUM                           { /* no action, not even $$ = $1 */ }
        | STRING                        { free($1); }
        | REG
        | expr OP expr
        | OP expr       %prec UNARY
        | '(' expr2 ')'
;

expr2:    ID                            { LabelUsed($1); }
        | FUNCT '(' expr2 ')'
        | FUNCT '(' ')'
        | NUM                           { /* no action, not even $$ = $1 */ }
        | STRING                        { free($1); }
        | expr OP expr
        | OP expr       %prec UNARY
        | '(' expr2 ')'
;
```

```
%%

void yyerror(const char *msg)
{
  Error("%s in line %u", msg, lineNo);
}

static void CopyLine(void)
{
  extern sectentry *csect;
  extern FILE *junkfp;
  int i;

  if ((lineBuffer != NULL) && (lineBuffer[0] != '\0')) {
    if ((csect == NULL) || (lineBuffer[0] == '\n')) {
      /* there is no section active */
      fprintf(junkfp, "%s\n", lineBuffer);
    } else {
      i = 0;
      while (lineBuffer[i] == ' ' || lineBuffer[i] == '\t') {
        i++;
      }
      /* don't write lines containing only whitespace  */
      if (lineBuffer[i] != '\n') {
        if (!(csect->dontCopy)) {
          fprintf(csect->fp, "%s", lineBuffer);
          (csect->lines)++;
        } else {
          csect->dontCopy = false;
        }
      }
    }
    lineBuffer[0] = '\0';
  }
}
```

D.2.1.4 Symbolverwaltung `as-sym.h`

```
/* assembler symbol table declarations */

#ifndef _AS_SYM_H
#define _AS_SYM_H

#include <sys/types.h>
#include "utils.h"

typedef enum {local, global, unknown} symtype;

typedef struct _sectentry {
  FILE *fp;
  char *name;
  unsigned lines;
  bool dontCopy;
  struct _sectentry *next;
} sectentry;

typedef struct _symentry {
  symtype type;
  char group; /* B uninit. data, D data, T text, U undef., locals lowercase */
  unsigned size;   /* size of a common declaration */
  bool iscommon;
  sectentry *commSect;
  char *name;
  struct _symentry *next;
} symentry;
```

```
extern symentry *symtab;
extern sectentry *secttab;
extern char *pname;
extern char *sourceName;
extern sectentry *csect;

bool LabelDef(char *);
bool LabelDefCommon(char *, unsigned);
bool LabelDefLSpace(char *, unsigned);
void LabelDeclGlobal(char *);
void LabelUsed(char *);
bool SetSource(char *);
void SetSection(char *);

#endif /* _AS_SYM_H */
```

D.2.1.5 Symbolverwaltung `as-sym.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include "as-sym.h"

extern void yyerror(const char *);

static char EncodeSection(void)
{
  if (csect == NULL) {
    Error("label defined before first section started");
    exit(2);
  }
  if (strcmp(csect->name, "text") == 0) {
    return 'T';
  } else if (strcmp(csect->name, "itext") == 0) {
    return 'T';
  } else if (strcmp(csect->name, "init") == 0) {
    return 'T';
  } else if (strcmp(csect->name, "fini") == 0) {
    return 'T';
  } else if (strncmp(csect->name, "data", 5) == 0) {
    return 'D';
  } else if (strncmp(csect->name, "const", 5) == 0) {
    return 'C';
  } else if (strncmp(csect->name, "udata", 5) == 0) {
    return 'B';
  } else if (strcmp(csect->name, "ctor") == 0) {
    return 'C';
  } else if (strcmp(csect->name, "dtor") == 0) {
    return 'C';
  } else if (strcmp(csect->name, "share") == 0) {
    return 'D';
  } else if (strcmp(csect->name, "junk") == 0) {
    return 'J';
  } else {
    return 'X';
  }
}


bool LabelDef(char *label)
{
  char errmsg[1000];
  symentry *sym;
```

```
  sym = symtab;
  while ((sym != NULL) && (strcmp(label, sym->name) != 0)) {
    sym = sym->next;
  }
  if (sym == NULL) {
    /* new label defined, make it local */
    sym = (symentry *) malloc(sizeof(symentry));
    sym->name = label;
    sym->type = local;
    sym->group = EncodeSection();
    sym->iscommon = false;
    sym->commSect = NULL;
    sym->size = 0;
    sym->next = symtab;
    symtab = sym;
    return false;
  } else {
    if ((sym->group != 'U') && !sym->iscommon) {
      sprintf(errmsg, "label '%s' defined twice", label);
      yyerror(errmsg);
      free(label);
      return true;
    }
    if (sym->type == unknown) {
      sym->type = local;
    }
    sym->group = EncodeSection();
    sym->iscommon = false;
    sym->commSect = NULL;
    free(label);
    return false;
  }
}

void LabelUsed(char *label)
{
  symentry *sym;

  sym = symtab;
  while ((sym != NULL) && (strcmp(label, sym->name) != 0)) {
    sym = sym->next;
  }
  if (sym == NULL) {
    /* new label used, make it undefined */
    sym = (symentry *) malloc(sizeof(symentry));

    sym->name = label;
    sym->type = unknown;
    sym->group = 'U';
    sym->iscommon = false;
    sym->commSect = NULL;
    sym->size = 0;
    sym->next = symtab;
    symtab = sym;
  } else {
    free(label);
  }
}

void LabelDeclGlobal(char *label)
{
  symentry *sym;

  if (csect != NULL) {
    csect->dontCopy = true;
  }
  sym = symtab;
  while ((sym != NULL) && (strcmp(label, sym->name) != 0)) {
```

```
    sym = sym->next;
  }
  if (sym == NULL) {
    /* new label declared global */
    sym = (symentry *) malloc(sizeof(symentry));
    sym->name = label;
    sym->type = global;
    sym->group = 'U';
    sym->iscommon = false;
    sym->commSect = NULL;
    sym->size = 0;
    sym->next = symtab;
    symtab = sym;
  } else {
    sym->type = global;
    sym->iscommon = false;
    sym->commSect = NULL;
    free(label);
  }
}

bool LabelDefCommon(char *label, unsigned size)
{
  char errmsg[1000];
  symentry *sym;

  csect->dontCopy = true;
  sym = symtab;
  while ((sym != NULL) && (strcmp(label, sym->name) != 0)) {
    sym = sym->next;
  }
  if (sym == NULL) {
    /* new label declared global common */
    sym = (symentry *) malloc(sizeof(symentry));
    sym->name = label;
    sym->type = global;
    sym->group = EncodeSection();
    sym->iscommon = true;
    sym->commSect = csect;
    sym->size = size;
    sym->next = symtab;
    symtab = sym;
    return false;
  } else {
    if (sym->type == local) {
      sprintf(errmsg, "label `%s' defined twice", label);
      yyerror(errmsg);
      free(label);
      return true;
    }
    if (sym->type == unknown) {
      sym->type = global;
      sym->iscommon = true;
      sym->commSect = csect;
    }
    sym->group = EncodeSection();
    sym->size = size;
    free(label);
    return false;
  }
}


bool LabelDefLSpace(char *label, unsigned size)
{
  char errmsg[1000];
  symentry *sym;
```

```
  csect->dontCopy = true;
  sym = symtab;
  while ((sym != NULL) && (strcmp(label, sym->name) != 0)) {
    sym = sym->next;
  }
  if (sym == NULL) {
    /* new label declared local common */
    sym = (symentry *) malloc(sizeof(symentry));
    sym->name = label;
    sym->type = local;
    sym->group = EncodeSection();
    sym->iscommon = true;
    sym->commSect = csect;
    sym->size = size;
    sym->next = symtab;
    symtab = sym;
    return false;
  } else {
    if (sym->type == global) {
      sprintf(errmsg, "label `%s' defined twice", label);
      yyerror(errmsg);
      free(label);
      return true;
    }
    if (sym->type == unknown) {
      sym->type = local;
      sym->iscommon = true;
      sym->commSect = csect;
    }
    sym->group = EncodeSection();
    sym->size = size;
    free(label);
    return false;
  }
}


bool SetSource(char *name)
{
  if (sourceName == NULL) {
    sourceName = name;
    return false;
  } else {
    yyerror("duplicate .file");
    free(name);
    return true;
  }
}

void SetSection(char *name)
{
  sectentry *sect;

  sect = secttab;
  while ((sect != NULL) && (strcmp(name, sect->name) != 0)) {
    sect = sect->next;
  }
  if (sect == NULL) {
    /* new section */
    sect = (sectentry *) malloc(sizeof(sectentry));
    sect->name = name;
    sect->fp = tmpfile();
    if (sect->fp == NULL) {
      Error("cannot create temporary file, exiting");
      exit(2);
    }
    sect->lines = 0;
    sect->next = secttab;
```

```
    secttab = sect;
  } else {
    free(name);
  }
  csect = sect;
  csect->dontCopy = true;     /* suppress copying the .section line */
}
```

D.2.2 Linker `ld`

D.2.2.1 Hauptprogramm `ld.c`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <alloca.h>
#include <assert.h>

#include "utils.h"
#include "ld-sym.h"
#include "getopt.h"


/* Version number */
#define MAJOR 1
#define MINOR 0


#define MPCASM_OPT "+g" PREFIX "/micro/micro.ep -c +p -x"


extern int yydebug;


typedef struct _pathEntry {      /* search path element */
  char *name;
  struct _pathEntry *next;
} pathEntry;

typedef struct _fileEntry {      /* file list element */
  char *name;
  bool isarchive;
  struct _fileEntry *next;
} fileEntry;


char *pname;                    /* the name of this program */
bool noWarnSize = false;        /* don't warn if a symbol has different sizes */
bool keepRelocatable = false;   /* output can be linked again */
bool showfiles = false;         /* show files as they are linked */

pathEntry *searchHead, *searchTail;
fileEntry *filesHead, *filesTail;

objectHeader linkObj;


static void AddToLibPath(const char *dirname)
{
```

```
  pathEntry *newdir;
  struct stat finfo;

  if ((stat(dirname, &finfo) != 0) || ((finfo.st_mode & S_IFMT) != S_IFDIR)) {
    Warning("directory `%s' does not exist", dirname);
    return;
  }
  newdir = (pathEntry *) malloc(sizeof(pathEntry));
  newdir->next = NULL;
  newdir->name = StrDup(dirname);
  if (searchTail != NULL) {
    searchTail->next = newdir;
  }
  searchTail = newdir;
  if (searchHead == NULL) {
    searchHead = newdir;
  }
}


static void ClearLibPath(void)
{
  pathEntry *curr, *deldir;

  curr = searchHead;
  while (curr != NULL) {
    deldir = curr;
    free(deldir->name);
    curr = curr->next;
    free(deldir);
  }
  searchHead = searchTail = NULL;
}


static char *SearchLibPath(const char *libname)
{
  pathEntry *dir;
  char name[FILENAME_MAX];
  bool notfound;

  dir = searchHead;
  notfound = true;
  while ((dir != NULL) && notfound) {
    strcpy(name, dir->name);
    strcat(name, "/");
    strcat(name, libname);
    if (access(name, R_OK) == 0) {
      notfound = false;
    } else {
      dir = dir->next;
    }
  }

  if (dir == NULL) {
    return NULL;
  }
  return StrDup(name);
}


static void AddLibraryToList(const char *lname)
{
  char *libname;
  fileEntry *newfile;
  size_t len;

  newfile = (fileEntry *) malloc(sizeof(fileEntry));
```

```
  newfile->next = NULL;
  len = strlen(lname);
  libname = (char *) alloca(len + 6);
  libname[0] = 'l';
  libname[1] = 'i';
  libname[2] = 'b';
  strcpy(&(libname[3]), lname);
  libname[len+3] = '.';
  libname[len+4] = 'a';
  libname[len+5] = '\0';
  newfile->name = SearchLibPath(libname);
  if (newfile->name == NULL) {
    Warning("no library file '%s' in search path", libname);
    free(newfile);
    return;
  }
  newfile->isarchive = true;
  if (filesTail != NULL) {
    filesTail->next = newfile;
  }
  filesTail = newfile;
  if (filesHead == NULL) {
    filesHead = newfile;
  }
}


static void AddFileToList(const char *fname)
{
  fileEntry *newfile;
  size_t len;

  newfile = (fileEntry *) malloc(sizeof(fileEntry));
  newfile->next = NULL;
  newfile->name = StrDup(fname);
  len = strlen(fname);
  newfile->isarchive = ((fname[len-2] == '.') && (fname[len-1] == 'a'));
  if (filesTail != NULL) {
    filesTail->next = newfile;
  }
  filesTail = newfile;
  if (filesHead == NULL) {
    filesHead = newfile;
  }
}


static void ClearFileList(void)
{
  fileEntry *curr, *del;

  curr = filesHead;
  while (curr != NULL) {
    del = curr;
    free(del->name);
    curr = curr->next;
    free(del);
  }
  filesHead = filesTail = NULL;
}


static unsigned Link(void)
{
  fileEntry *curr;
  symbol *currSym;
  FILE *ifd;
  objectHeader *objInfo;
```

```
  libraryHeader *libInfo;
  unsigned libSym;
  bool completed;
#ifndef HASHING
  char *name;
#endif
  section *udata;
  int linklevel, lastlevel;
  symbol *lastSymbol;

  linklevel = 0;
  lastlevel = 0;
  lastSymbol = NULL;
  curr = filesHead;
  while (curr != NULL) {
    if (!(curr->isarchive)) {
      /* object files are linked unconditionally, even if unreferenced.  */
      if ((ifd = OpenObject(curr->name)) == NULL) {
        Error("cannot open input file `%s': %s", curr->name, strerror(errno));
        return 2;
      }
      objInfo = ReadObjectHeader(ifd, curr->name);
      if (showfiles) {
        fprintf(stderr, "linking with %s\n", curr->name);
      }
      if (objInfo == NULL) return 2;
      if (!CopyObject(&linkObj, objInfo, ifd)) {
        fclose(ifd);
        ClearObject(objInfo);
        free(objInfo);
        return 1;
      }
      fclose(ifd);
      ClearObject(objInfo);
      free(objInfo);
    } else {
      /* members of libraries are only linked if a reference is resolved.  */
      if ((ifd = OpenLibrary(curr->name)) == NULL) {
        Error("cannot open library file `%s': %s", curr->name, strerror(errno));
        return 2;
      }
      libInfo = ReadLibraryHeader(ifd, curr->name);
      if (libInfo == NULL)
        return 2;
      if (libInfo->count == 0) {
        /* empty library, all done */
        fclose(ifd);
        ClearLibrary(libInfo);
        free(libInfo);
      } else {
        if (showfiles) {
          fprintf(stderr, "linking with %s\n", curr->name);
          linklevel = 0;
          lastlevel = 0;
          lastSymbol = linkObj.symTail;
        }
        currSym = linkObj.symHead;
        while (currSym != NULL) {
          if (currSym->group == 'U') {
            libSym = 0;
            while ((libSym < libInfo->count) &&
                   (strcmp(currSym->name, libInfo->tab[libSym].sym) != 0)) {
              libSym++;
            }
            if (libSym < libInfo->count) {
              if (showfiles) {
                if (lastlevel == linklevel) {
                  fprintf(stderr, "level %i (%s)\t", linklevel, currSym->name);
```

```
              } else {
                fprintf(stderr, "level %i (%s)\t", linklevel-1,currSym->name);
              }
              lastlevel = linklevel;
            }

            /* this symbol is defined by the current archive member.  */
            /* new symbol entries MUST be appended at the end of the
               current symbols, otherwise this whole construction fails.  */
            if (!CopyLibraryMember(&linkObj, &(libInfo->tab[libSym]), ifd,
                                   curr->name)) {
              fclose(ifd);
              ClearLibrary(libInfo);
              free(libInfo);
              return 1;
            }
          }
        }
        currSym = currSym->next;
        if (showfiles) {
          if (currSym == lastSymbol) {
            linklevel++;
            lastSymbol = linkObj.symTail;
          }
        }
      }
      fclose(ifd);
      ClearLibrary(libInfo);
      free(libInfo);
    }
  }
  curr = curr->next;
}
if (keepRelocatable) {
  return 0;
}
completed = true;
udata = NULL;
currSym = linkObj.symHead;
while (currSym != NULL) {
  if (currSym->group == 'U') {
    Error("undefined symbol `%s'", currSym->name);
    completed = false;
  }
  if (currSym->iscommon && currSym->group != 'D') {
    udata = linkObj.sects;
    while ((udata != NULL) && (strcmp(udata->name, "udata") != 0)) {
      udata = udata->next;
    }
    if (udata == NULL) {
      AddUniqueSection(&linkObj, "udata", 0);
      udata = linkObj.sects;
      while ((udata != NULL) && (strcmp(udata->name, "udata") != 0)) {
        udata = udata->next;
      }
      assert(udata != NULL);
      udata->tmpfile = tmpfile();
    }
#ifdef HASHING
    if (fprintf(udata->tmpfile,
                "\talign\n%s\t= offset()+XSdataoffset\n\tblockb\t%u\n",
                currSym->newname, currSym->size) < 0) {
#else
    EncodeSymbol(currSym->name, currSym->number, &name);
    if (fprintf(udata->tmpfile,
                "\talign\n%s\t= offset()+XSdataoffset\n\tblockb\t%u\n",
                name, currSym->size) < 0) {
#endif
```

```
        Error("cannot write to temp file: %s", strerror(errno));
        return 2;
      }
      udata->lines += 3;
#ifndef HASHING
      free(name);
#endif
    }
    currSym = currSym->next;
  }
  if (completed) {
    return 0;
  } else {
    return 1;
  }
}


static unsigned OutputResultFinal(const char *outname, bool outputAsm, bool map)
{
  char *cmdline;
  FILE *tmpfd;
  char *tmpfname;
  char *listname;
  unsigned retval;
  int st;
  int syserrno;
  struct stat sb;
  mode_t permission, mask;

  listname = NULL;
  if (outputAsm) {
    tmpfname = (char *) outname;
  } else {
    if (map) {
      listname = (char *) alloca(strlen(outname) + 3);
      strcpy(listname, outname);
      strcat(listname, ".l");
    }
    tmpfname = choose_temp();
    if (*tmpfname == '\0') {
      Error("cannot construct main temp file name");
      return 2;
    }
  }
  if ((tmpfd = fopen(tmpfname, "w+")) == NULL) {
    Error("cannot create main temp file: %s", strerror(errno));
    return 2;
  }

  if ((retval = DumpSection(&linkObj, "const_start_glue", tmpfd)) > 0) {
    fclose(tmpfd);
    remove(tmpfname);
    return retval;
  }
  if ((retval = DumpSection(&linkObj, "const_start", tmpfd)) > 0) {
    fclose(tmpfd);
    remove(tmpfname);
    return retval;
  }
  if ((retval = DumpSection(&linkObj, "const", tmpfd)) > 0) {
    fclose(tmpfd);
    remove(tmpfname);
    return retval;
  }
  if ((retval = DumpSection(&linkObj, "ctor", tmpfd)) > 0) {
    fclose(tmpfd);
    remove(tmpfname);
```

```
  return retval;
}
if ((retval = DumpSection(&linkObj, "dtor", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "const_end", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "const_end_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}

if ((retval = DumpSection(&linkObj, "data_start_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "data_start", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "data", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "data_end", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "data_end_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}

if ((retval = DumpSection(&linkObj, "udata_start_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "udata_start", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "udata", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "udata_end", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "udata_end_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
```

```
}

if ((retval = DumpSection(&linkObj, "text_start_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "text_start", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "itext", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "init", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "fini", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "text", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "text_end", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}
if ((retval = DumpSection(&linkObj, "text_end_glue", tmpfd)) > 0) {
  fclose(tmpfd);
  remove(tmpfname);
  return retval;
}

if (fclose(tmpfd) < 0) {
  Error("cannot close temp file: %s", strerror(errno));
  return 2;
}

if (outputAsm) {
  return 0;
} else {
  /* start the real assembler to produce executable */
  cmdline = (char *) alloca (FILENAME_MAX);
  strcpy(cmdline, MPCASM_NAME);
  strcat(cmdline, " ");
  strcat(cmdline, MPCASM_OPT);
  strcat(cmdline, " +a");
  strcat(cmdline, outname);
  if (listname == NULL) {
    strcat(cmdline, " -l ");
  } else {
    strcat(cmdline, " +l");
    strcat(cmdline, listname);
    strcat(cmdline, " ");
  }
  strcat(cmdline, tmpfname);
  errno = 0;
  st = system(cmdline);
```

```
    syserrno = errno;
    remove(tmpfname);
    if (st == 0) {
      if (stat(outname, &sb) != -1) {
        permission = sb.st_mode;
        mask = umask(0);        /* get umask */
        umask(mask);
        permission = (permission | S_IXUSR | S_IXGRP | S_IXOTH) & (~mask);
        chmod(outname, permission);
      }
      return 0;
    } else {
      remove(outname);
      if (syserrno > 0) {
        /* /bin/sh was not found */
        Error("system() failed: %s", strerror(syserrno));
        return 2;
      }
    }
    return 1;  /* cmd was unsuccessful */
  }
}


static unsigned OutputResultObj(const char *outname)
{
  FILE *fd;
  symbol *sym;
  section *sect;
  unsigned i;

  if ((fd = fopen(outname, "w+")) == NULL) {
    Error("cannot create output file `%s': %s", outname, strerror(errno));
    return 2;
  }
  if (fputs(":MPC object file 1.0\n", fd) < 0) {
    Error("cannot write to output file `%s': %s", outname, strerror(errno));
    return 2;
  }
  /* FIXME: currently no source file name written.  It is not quite clear
     what should be written here - one of the source files, a list of
     all the source files, or something else.  */
  if (fputs(":begin symbol\n", fd) < 0) {
    Error("cannot write to output file `%s': %s", outname, strerror(errno));
    return 2;
  }
  sym = linkObj.symHead;
  while (sym != NULL) {
    if (fprintf(fd, "%s\t%c", sym->name, sym->group) < 0) {
      Error("cannot write to output file `%s': %s", outname, strerror(errno));
      return 2;
    }
    if (sym->iscommon) {
      if (fputc('*', fd) < 0) {
        Error("cannot write to output file `%s': %s", outname, strerror(errno));
        return 2;
      }
    }
    if (fprintf(fd, "\t%i\n", sym->size) < 0) {
      Error("cannot write to output file `%s': %s", outname, strerror(errno));
      return 2;
    }
    sym = sym->next;
  }
  if (fputs(":end symbol\n", fd) < 0) {
    Error("cannot write to output file `%s': %s", outname, strerror(errno));
    return 2;
  }
```

```
  if (fputs(":begin section\n", fd) < 0) {
    Error("cannot write to output file ‘%s’: %s", outname, strerror(errno));
    return 2;
  }
  sect = linkObj.sects;
  while (sect != NULL) {
    if ((sect->lines != 0) && (strcmp(sect->name, "junk") != 0)) {
      if (fprintf(fd, "%s\t%i\n", sect->name, sect->lines) < 0) {
        Error("cannot write to output file ‘%s’: %s", outname, strerror(errno));
        return 2;
      }
    }
    sect = sect->next;
  }
  if (fputs(":end section\n", fd) < 0) {
    Error("cannot write to output file ‘%s’: %s", outname, strerror(errno));
    return 2;
  }

  if (fputs(":end header\n", fd) < 0) {
    Error("cannot write to output file ‘%s’: %s", outname, strerror(errno));
    return 2;
  }

  sect = linkObj.sects;
  while (sect != NULL) {
    rewind(sect->tmpfile);
    for (i = 0; i < sect->lines; i++) {
      ReadLine(sect->tmpfile);
      if (fputs(lineBuffer, fd) < 0) {
        Error("cannot write to output file ‘%s’: %s", outname, strerror(errno));
        return 2;
      }
    }
    sect = sect->next;
  }

  if (fclose(fd) < 0) {
    Error("error closing output file %s’: %s", outname, strerror(errno));
    return 2;
  }
  return 0;
}


int main(int argc, char *argv[])
{
  char *outname = "a.out";       /* output file name */
  bool printVer = false;         /* print version number */
  bool makeAsmSource = false;    /* output can be sent through real assembler */
  bool printMap = false;         /* print a listing of the sections */
  int c;                         /* current option */
  int errflg = 0;                /* > 0 means there are errors in the cmdline */
  unsigned status;

  pname = argv[0];
  status = 0;

  searchHead = searchTail = NULL;
  filesHead = filesTail = NULL;

  AddToLibPath(STDLIBPATH);

  /* return arguments in order, as link order makes a difference, and even
     -l options must be processed in the relative position to normal files
     and other -l options. */
  while ((c = getopt(argc, argv, "-drmstVXZo:L:l:")) != EOF) {
```

```
    switch (c) {
      case 1:
        AddFileToList(optarg);
        break;
      case 'd':
        yydebug = 1;
        break;
      case 'r':
        keepRelocatable = true;
        break;
      case 'm':
        printMap = true;
        break;
      case 's':
        break;       /* there is nothing to strip, ignore option */
      case 't':
        noWarnSize = true;
        break;
      case 'L':
        AddToLibPath(optarg);
        break;
      case 'l':
        AddLibraryToList(optarg);
        break;
      case 'o':
        outname = optarg;
        break;
      case 'V':
        printVer = true;
        break;
      case 'X':
        makeAsmSource = true;
        break;
      case 'Z':
        showfiles = true;
        break;
      case '?':
      default:
        errflg++;
    }
  }
  while (optind < argc) {
    AddFileToList(argv[optind]);
    optind++;
  }
  if (filesHead == NULL) {            /* no files specified */
    errflg++;
  }

  if (errflg || printVer) {
    fprintf(stderr, "MONADS-PC pseudo-linker version %i.%i\n", MAJOR, MINOR);
  }
  if (errflg) {
    fprintf(stderr, "usage: %s [-rmstVXZ] [-L <dir>] [-lo <file>] files...\n",
                    pname);
    status = 2;
  }

  lineBuffer = NULL;
  lineBufferAlloc = 0;

  linkObj.source = NULL;
  linkObj.symHead = NULL;
  linkObj.symTail = NULL;
  linkObj.sects = NULL;
#ifdef HASHING
  AllocateHashTable(18); /* make a 262144 element hash table */
#endif
```

```
  if (!keepRelocatable) {
    AddFileToList(ASMGLUEOBJ);
  }

  if (status == 0) {
    status = Link();
    if (status == 0) {
      if (keepRelocatable) {
        status = OutputResultObj(outname);
      } else {
        status = OutputResultFinal(outname, makeAsmSource, printMap);
      }
    }
  }

  xfree(lineBuffer);
  lineBuffer = NULL;

  ClearObject(&linkObj);
#ifdef HASHING
  FreeHashTable();
#endif

  ClearLibPath();
  ClearFileList();

  return status;
}
```

D.2.2.2 Scanner `ld-lex.l`

```
/* definitions */

%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include "utils.h"
#include "ld-sym.h"
#include "ld-parse.h"

static int ProcessReserved(const char *);

extern char *pname;
extern bool readArchive;
extern unsigned readLimit;

#define YY_INPUT(buf, result, max_size)                                 \
  do {                                                                  \
    if (readArchive) {                                                  \
      if (readLimit < (unsigned) max_size)                              \
        result = fread(buf, 1, readLimit, yyin);                        \
      else                                                              \
        result = fread(buf, 1, max_size, yyin);                         \
      if (result > 0)                                                   \
        readLimit -= result;                                            \
    } else {                                                            \
      result = fread(buf, 1, max_size, yyin);                           \
    }                                                                   \
    if (result == 0 && ferror(yyin))                                    \
      YY_FATAL_ERROR("input in flex scanner failed");                   \
  } while (0);
%}
```

```
%option 8bit
%option case-sensitive
%option never-interactive
%option nounput
%option noyywrap

WS      [ \t]+
LID     [A-Za-z_][0-9A-Za-z_]*
DIG     [0-9]
DNUM    {DIG}+
HEXNUM  "$"[0-9A-Fa-f]+
STR     ("\""[^"\n]*"\"")*


%%

"+"|"-"|"*"|"/"|"%"|"\\"|"\|"|"&"|"@"   { AppendLine(yytext); return OP; }
{WS}                    { AppendLine(yytext); /* ignore whitespace */ }
{DNUM}|{HEXNUM}         { AppendLine(yytext); return NUM; }
{STR}                   { AppendLine(yytext); return STRING; }
{LID}                   { yylval.tinfo.tpos = strlen(lineBuffer);
                          yylval.tinfo.len = strlen(yytext);
                          AppendLine(yytext);
                          return ProcessReserved(yytext); }
";"[^\n]*               { AppendLine(yytext); /* ignore comments */ }
"\n"                    { AppendLine(yytext); return yytext[0]; }
<<EOF>>                 { return yytext[0]; }
.                       { AppendLine(yytext); return yytext[0]; }

%%

/* user subroutines */

static int ProcessReserved(const char *id)
{
  if ((id[0] == 'a') && (id[1] == '\0' || (id[1] == 'x' && id[2] == '\0'))) {
    return REG;
  } else if ((id[0] == 'i') && (id[1] >= '0') && (id[1] <= '3') &&
             (id[2] == '\0')) {
    return REG;
  } else if ((id[0] == 'c') && (id[1] == 'x') && (id[2] == '\0')) {
    return REG;
  } else if ((id[0] == 'c') && isdigit((int) id[1]) &&
             ((id[2] == '\0') || (isdigit((int) id[2]) && (id[3] == '\0')))) {
    return REG;
  } else if (strcmp(id, "data_segment") == 0) {
    return RASMDEC2;
  } else if (strcmp(id, "data_size_of") == 0) {
    return FUNCT;
  } else if (strcmp(id, "ptr_size_of") == 0) {
    return FUNCT;
  }
  return ID;
}
```

D.2.2.3 Parser `ld-parse.y`

```
/* MONADS-PC pseudo-linker parser. */

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
```

```
#include <errno.h>
#include <assert.h>

#include "ld-sym.h"

#define YYDEBUG 1

extern int yylex(void);
void yyerror(const char *);
extern void AppendLine(const char *);
static void SelectSection(void);
static void CopyLine(void);
static bool TranslateID(const idInfo, bool);

extern unsigned lineNo;
extern char *pname;
extern objectHeader linkObj;
extern objectHeader *currObj;
extern bool firstSection;
extern bool keepRelocatable;
static unsigned currSectLines;
static section *currSect;
static section *currLink;
%}

%union {
  idInfo tinfo;
}

%token RASMDEC2
%token <tinfo> ID
%token REG
%token FUNCT
%token NUM
%token STRING
%token OP

%left OP
%right UNARY

%%

program:  input

input:    /* empty */
        | input line '\n'               { lineNo++; CopyLine(); }
;

line:     /* empty */
        | ID ':'                { if (TranslateID($1, true)) YYABORT; }
        | ID '=' expr           { if (TranslateID($1, false)) YYABORT; }
        | ID opers              { /* do nothing */ }
        | RASMDEC2 ID ',' ID    { if (TranslateID($4, false)) YYABORT; }
;

opers:    /* empty */
        | oper
        | opers ',' oper
;

oper:     expr
        | '#' expr
        | '(' REG ')' '[' REG ']'       { /* expr can be empty if idm/nidm */ }
        | '(' REG ')'                   { /* expr can be empty if idm/nidm */ }
        | expr '(' REG ')' '[' REG ']'
        | expr '(' REG ')'
;
```

```
expr:     ID                            { if (TranslateID($1, false)) YYABORT; }
        | FUNCT '(' expr2 ')'
        | FUNCT '(' ')'
        | NUM
        | STRING
        | REG
        | expr OP expr
        | OP expr       %prec UNARY
        | '(' expr2 ')'
;

expr2:    ID                            { if (TranslateID($1, false)) YYABORT; }
        | FUNCT '(' expr2 ')'
        | FUNCT '(' ')'
        | NUM
        | STRING
        | expr OP expr
        | OP expr       %prec UNARY
        | '(' expr2 ')'
;

%%

void yyerror(const char *msg)
{
  Error("%s in line %u", msg, lineNo);
}

static void SelectSection(void)
{
  section *newsect;

  if (firstSection) {
    firstSection = false;
    currSect = currObj->sects;
  } else {
    currSect = currSect->next;
  }
  if (currSect != NULL) {
    currSectLines = currSect->lines;
    currLink = linkObj.sects;
    while ((currLink != NULL) &&
           (strcmp(currLink->name, currSect->name) != 0)) {
      currLink = currLink->next;
    }
    if (currLink == NULL) {
      newsect = (section *) malloc(sizeof(section));
      newsect->name = StrDup(currSect->name);
      newsect->lines = 0;
      newsect->tmpfile = tmpfile();
      newsect->next = linkObj.sects;
      linkObj.sects = newsect;
      currLink = newsect;
    }
  }
}


static void CopyLine(void)
{
  if ((currSectLines == 0) || firstSection) {
    SelectSection();
  }
  if (fprintf(currLink->tmpfile, "%s", lineBuffer) < 0) {
    Error("cannot write to temp file: %s", strerror(errno));
    exit(2);
  }
  lineBuffer[0] = '\0';
```

```
  currLink->lines++;
  currSectLines--;
}

static bool TranslateID(const idInfo id, bool conv)
{
  char *oldid, *newid, *rest;
  symbol *oldsym;
  char group;

  oldid = (char *) alloca(id.len+1);
  strncpy(oldid, &(lineBuffer[id.tpos]), id.len);
  oldid[id.len] = '\0';
  LookupTranslation(oldid, &newid, &oldsym);
  if (newid == NULL)
    return true;
  rest = (char *) alloca(strlen(&(lineBuffer[id.tpos+id.len]))+1);
  strcpy(rest, &(lineBuffer[id.tpos+id.len]));
  lineBuffer[id.tpos] = '\0';
  AppendLine(newid);
  group = toupper(oldsym->group);
  if (!keepRelocatable && conv &&
      (group == 'B' || group == 'C' || group == 'D' || group == 'T')) {
    if (group == 'T') {
      AppendLine("\t= offset()/4");
    } else {
      AppendLine("\t= offset()+XS");
      if (group == 'C') {
        AppendLine("const");
      } else {
        AppendLine("data");
      }
      AppendLine("offset");
    }
  } else {
    AppendLine(rest);
  }
  return false;
}
```

D.2.2.4 Symbolverwaltung `ld-sym.h`

```
#ifndef _LD_SYM_H
#define _LD_SYM_H

#include <sys/types.h>
#include "utils.h"


#define HASHING

typedef struct {
  unsigned tpos;
  unsigned len;
} idInfo;

typedef struct _symbol {
  char *name;         /* symbol the compiler used in the assembler output */
  unsigned char group;  /* B udata, D data, T text, U undef., locals lcase */
  bool iscommon;
  unsigned size;
  char *oldname;      /* name as read from the object file */
  unsigned oldnumber; /* number extracted from the object file */
  char *newname;      /* new name assigned to avoid clashes */
  unsigned newnumber; /* number assigned to the symbol */
  struct _symbol *nexthash; /* chain of symbols with same key */
```

```
  struct _symbol *next;
} symbol;

typedef struct _section {
  char *name;
  unsigned lines;
  FILE *tmpfile;
  struct _section *next;
} section;

typedef struct {
  char *source;
  symbol *symHead, *symTail;
  section *sects;
} objectHeader;

typedef struct _libraryIndex {
  char *sym;
  off_t offset;
  objectHeader *obj;
} libraryIndex;

typedef struct _libraryHeader {
  unsigned count;
  libraryIndex *tab; /* pointer to array, length is count */
  char *strtab; /* all symbol names, 0 delimited, one after another */
} libraryHeader;


extern bool firstSection;
extern bool noWarnSize;
extern bool showfiles;

extern bool readArchive; /* interface to scanner */
extern unsigned readLimit;

void AllocateHashTable(unsigned);
void FreeHashTable(void);

void EncodeSymbol(char *, unsigned, char **);
void LookupTranslation(char *, char**, symbol **);
unsigned DumpSection(objectHeader *, const char *, FILE *);
void AddUniqueSection(objectHeader *, const char *, const unsigned);

FILE *OpenObject(const char *);
objectHeader *ReadObjectHeader(FILE *, const char *);
void ClearObject(objectHeader *);
bool CopyObject(objectHeader *, objectHeader *, FILE *);

FILE *OpenLibrary(const char *);
libraryHeader *ReadLibraryHeader(FILE *, const char *);
void ClearLibrary(libraryHeader *);
bool CopyLibraryMember(objectHeader *, libraryIndex *, FILE *, const char *);

#endif /* _LD_SYM_H */
```

D.2.2.5 Symbolverwaltung `ld-sym.c`

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <assert.h>
```

```
#include <limits.h>

#include "utils.h"
#include "ld-sym.h"
#include "parser.h"
#include "ar.h"


bool firstSection;
objectHeader *currObj;

bool readArchive;
unsigned readLimit;


static unsigned hashbits;
static unsigned hashmask;
static symbol **hashtable;


static unsigned NumLen(unsigned num)
{
  unsigned numlen;

  if (num == 0) {
    return 1;
  } else {
    numlen = 0;
    do {
      num /= 10;
      numlen++;
    } while (num != 0);
    return numlen;
  }
}


void EncodeSymbol(char *name, unsigned num, char **newname)
{
  if (num == 0) {
    *newname = StrDup(name);
  } else {
    *newname = (char *) malloc(strlen(name) + NumLen(num) + 2);
    sprintf(*newname, "N%u%s", num, name);
  }
}


static void DecodeSymbol(char *oldname, char **name, unsigned *num)
{
  char *startn;

  if (oldname[0] == 'N') {
    *num = strtoul(&(oldname[1]), &startn, 10);
    *name = StrDup(startn);
  } else {
    *name = StrDup(oldname);
    *num = 0;
  }
}


static symbol *AddUniqueSymbol(objectHeader *info, const char *name,
                               const char group, const bool iscommon,
                               const unsigned size,
                               const char *oldname, const unsigned oldnum)
{
  symbol *newsym;
```

```
  newsym = (symbol *) malloc(sizeof(symbol));

  newsym->name = StrDup(name);
  newsym->group = group;
  newsym->iscommon = iscommon;
  newsym->size = size;
  newsym->oldname = StrDup(oldname);
  newsym->oldnumber = oldnum;
  newsym->newname = NULL;
  newsym->newnumber = 0;
  newsym->nexthash = NULL;
  newsym->next = NULL;
  if (info->symHead == NULL) {
    info->symHead = info->symTail = newsym;
  } else {
    info->symTail->next = newsym;
    info->symTail = newsym;
  }
  return newsym;
}


void AddUniqueSection(objectHeader *info, const char *name,
                      const unsigned lines)
{
  section *curr, *prev;
  section *newsect;

  prev = NULL;
  curr = info->sects;
  while ((curr != NULL) && (strcmp(curr->name, name) != 0)) {
    prev = curr;
    curr = curr->next;
  }
  assert(curr == NULL);
  newsect = (section *) malloc(sizeof(section));
  newsect->name = StrDup(name);
  newsect->lines = lines;
  newsect->tmpfile = NULL;
  if (prev == NULL) {
    newsect->next = NULL;
    info->sects = newsect;
  } else {
    newsect->next = prev->next;
    prev->next = newsect;
  }
}


FILE *OpenObject(const char *fname)
{
  FILE *ifd;

  ifd = fopen(fname, "r");
  if (ifd == NULL)
    return NULL;
  lineNo = 1;
  ReadLine(ifd);
  if (strcmp(lineBuffer, ":MPC object file 1.0\n") != 0) {
    Error("object file format error in `%s': magic", fname);
    return NULL;
  }
  return ifd;
}


objectHeader *ReadObjectHeader(FILE *fd, const char *fname)
```

```
{
  char *symname;
  char *symgroup;
  bool symcommon;
  unsigned symsize;
  char *sectname;
  unsigned symnum;
  char *realname;
  unsigned sectlines;
  objectHeader *info;
  size_t len;

  info = (objectHeader *) malloc(sizeof(objectHeader));
  info->source = NULL;
  info->symHead = info->symTail = NULL;
  info->sects = NULL;

  ReadLine(fd);
  if (strncmp(lineBuffer, ":source", 7) == 0) {
    len = strlen(lineBuffer); /* including space and \n */
    lineBuffer[len - 1] = '\0'; /* chop off \n */
    info->source = StrDup(&(lineBuffer[8]));
    ReadLine(fd);
  }
  if (strcmp(lineBuffer, ":begin symbol\n") != 0) {
    Error("object format error in `%s': bsym", fname);
    ClearObject(info);
    free(info);
    return NULL;
  }
  ReadLine(fd);
  while ((strcmp(lineBuffer, ":end symbol\n") != 0) && (!feof(fd))) {
    symname = strtok(lineBuffer, "\t");
    symgroup = strtok(NULL, "\t");
    symcommon = ((strlen(symgroup) == 2) && (symgroup[1] == '*'));
    symsize = atoi(strtok(NULL, "\n"));
    DecodeSymbol(symname, &realname, &symnum);
    AddUniqueSymbol(info, realname, symgroup[0], symcommon, symsize,
                    symname, symnum);
    free(realname);
    ReadLine(fd);
  }
  if (feof(fd)) {
    Error("object format error in `%s': no esym", fname);
    ClearObject(info);
    free(info);
    return NULL;
  }
  ReadLine(fd);
  if (strcmp(lineBuffer, ":begin section\n") != 0) {
    Error("object format error in `%s': bsect", fname);
    ClearObject(info);
    free(info);
    return NULL;
  }
  ReadLine(fd);
  while ((strcmp(lineBuffer, ":end section\n") != 0) && (!feof(fd))) {
    sectname = strtok(lineBuffer, "\t");
    sectlines = atoi(strtok(NULL, "\n"));
    AddUniqueSection(info, sectname, sectlines);
    ReadLine(fd);
  }
  if (feof(fd)) {
    Error("object format error in `%s': no esect", fname);
    ClearObject(info);
    free(info);
    return NULL;
  }
```

```
  ReadLine(fd);
  if (strcmp(lineBuffer, ":end header\n") != 0) {
    Error("object format error in '%s': no ehdr", fname);
    ClearObject(info);
    free(info);
    return NULL;
  }
  return info;
}


void ClearObject(objectHeader *obj)
{
  section *currSect, *delSect;
  symbol *currSym, *delSym;

  xfree(obj->source);
  obj->source = NULL;

  currSym = obj->symHead;
  while (currSym != NULL) {
    delSym = currSym;
    free(delSym->name);
    free(delSym->oldname);
    xfree(delSym->newname);
    currSym = currSym->next;
    free(delSym);
  }
  obj->symHead = obj->symTail = NULL;

  currSect = obj->sects;
  while (currSect != NULL) {
    delSect = currSect;
    free(delSect->name);
    if (delSect->tmpfile != NULL) {
      fclose(delSect->tmpfile);
    }
    currSect = currSect->next;
    free(delSect);
  }
  obj->sects = NULL;
}


static unsigned Hash(const char *key)
{
  unsigned retval = 0xbeefcafe & hashmask;
  int i;

#if (UINT_MAX != 4294967295U)
#error This requires at least a 32 bit integer
#endif
  for (i = 0; key[i] != '\0'; i++) {
    retval = (retval << 7) | (retval >> (hashbits - 7));
    retval ^= (unsigned) key[i];
    retval &= hashmask;
  }
  return retval;
}


static void DumpHash(void) __attribute__((unused));

static void DumpHash(void)
{
  unsigned i;
  symbol *curr;
```

```
  for (i = 0; i <= hashmask; i++) {
    if (hashtable[i] != NULL) {
      fprintf(stderr, "%08x: `%s' `%s'->`%s'\n", i, hashtable[i]->name,
                      hashtable[i]->oldname, hashtable[i]->newname);
      curr = hashtable[i]->nexthash;
      while (curr != NULL) {
        fprintf(stderr, "        : `%s' `%s'->`%s'\n", curr->name,
                        curr->oldname, curr->newname);
        curr = curr->nexthash;
      }
    }
  }
}


void AllocateHashTable(unsigned bits)
{
  hashbits = bits;
  hashmask = (1 << bits) - 1;
  hashtable = (symbol **) calloc(hashmask + 1, sizeof(symbol *));
}


static __inline__ void InsertHashString(const char *key, symbol *elem)
{
  unsigned hash = Hash(key);

  elem->nexthash = hashtable[hash];
  hashtable[hash] = elem;
}


static symbol *LookupHashString(const char *key)
{
  symbol *curr;
  unsigned hash = Hash(key);

  curr = hashtable[hash];
  while (curr != NULL && strcmp(curr->name, key) != 0) {
    curr = curr->nexthash;
  }
  return curr;
}


/* Reorder hash chain to make sure that global symbols will now be at
   the front of the chain.  This is used to implement the visibility rules
   for local symbols: they are inserted in the front and moved to the back
   after the file they are defined in is finished.  They cannot be deleted
   from the chain, because the numbers stored in them are needed to find
   the next available number.  */

static void ReorderHashChain(void)
{
  unsigned i;
  bool locals;
  symbol *curr, *prev, *next;

  for (i = 0; i <= hashmask; i++) {
    curr = hashtable[i];
    prev = NULL;
    /* no locals seen yet, no need to reorder entries.  */
    locals = false;
    while (curr != NULL) {
      next = curr->nexthash;
      if (!islower(curr->group)) {
        if (locals) {
          /* locals is always false in the first round, so we can assume
```

```
            that prev is not NULL any more.  */
          curr->nexthash = hashtable[i];
          hashtable[i] = curr;
          prev->nexthash = next;
        } else {
          prev = curr;
        }
      } else {
        locals = true;
        prev = curr;
      }
      /* prev is not changed in general, because if an entry is moved, the
         prev element of the next element to be looked at is still the same
         element as before.  */
      curr = next;
    }
  }
}


void FreeHashTable(void)
{
#ifdef HASH_STAT
  unsigned hashused;
  unsigned hashlength;
  symbol *curr;
  int i;

  for (i = 9; i <= hashmask; i++) {
    if (hashtable[i] != NULL) {
      hashused++;
      hashlength++;
      curr = hashtable[i]->nexthash;
      while (curr != NULL) {
        hashlength++;
        curr = curr->nexthash;
      }
    }
  }
  /* Note that relatively long chains are normal, because local symbols
     are very similar and many occur in each file linked.  We hash with
     the original symbol, so these "duplicates" cannot be avoided. It
     doesn't affect performance too much, because they are always in
     the back of the list, so usually they aren't even looked at.  */
  fprintf(stderr, "%u buckets used (out of %u)\n", hashused, hashmask+1);
  fprintf(stderr, "average chain length of used buckets %f\n",
                  hashlength * 1.0 / hashused);
  fprintf(stderr, "average chain length of all buckets %f\n",
                  hashlength * 1.0 / (hashmask + 1));
#endif
  hashbits = 0;
  hashmask = 0;
  free(hashtable);
  hashtable = NULL;
}


static unsigned FindNewNumber(const char *name, bool global)
{
  unsigned num;
  symbol *curr;

  curr = LookupHashString(name);
  if (global) {
    return 0;
  }
  num = 1;
  while (curr != NULL) {
```

```
    if ((strcmp(curr->name, name) == 0) && (curr->newnumber >= num)) {
      num = curr->newnumber + 1;
    }
    curr = curr->nexthash;
  }
  return num;
}


static symbol *FindGlobalSymbol(const char *name)
{
  symbol *curr;

  curr = LookupHashString(name);
  while ((curr != NULL) &&
         ((strcmp(name, curr->name) != 0) || islower(curr->group))) {
    curr = curr->nexthash;
  }
  return curr;
}


static void AddTranslation(symbol *new, const char *newname,
                           const unsigned newnum)
{
  EncodeSymbol(new->name, newnum, &(new->newname));
  new->newnumber = newnum;
}


void LookupTranslation(char *oldname, char **newname, symbol **sym)
{
  symbol *curr;
  char *name;
  unsigned oldnum;

  DecodeSymbol(oldname, &name, &oldnum);
  curr = LookupHashString(name);
  while ((curr != NULL) && (strcmp(oldname, curr->oldname) != 0)) {
    curr = curr->nexthash;
  }
  if (curr == NULL) {
    Error("internal: %s not found", oldname);
    *newname = NULL;
    return;
  }
  assert(curr != NULL);
  *newname = curr->newname;
  *sym = curr;
}


bool LinkSymbols(objectHeader *linkObj, objectHeader *obj)
{
  symbol *currSym, *linkLoc, *exSym;
  unsigned newnum;

  /* do the linking of symbols */
  currSym = obj->symHead;
  while (currSym != NULL) {
    if (islower(currSym->group)) {
      /* symbol to be added is a local symbol, always make new mapping */
      assert(currSym->iscommon == false);
      newnum = FindNewNumber(currSym->name, false);
      linkLoc = AddUniqueSymbol(linkObj, currSym->name, currSym->group,
                                currSym->iscommon, currSym->size,
                                currSym->oldname, currSym->oldnumber);
      AddTranslation(linkLoc, currSym->name, newnum);
```

```
      InsertHashString(linkLoc->name, linkLoc);
    } else {
      /* global symbol */
      exSym = FindGlobalSymbol(currSym->name);
      if (exSym == NULL) {
        /* new global symbol, avoid clashes with local symbols */
        newnum = FindNewNumber(currSym->name, true);
        linkLoc = AddUniqueSymbol(linkObj, currSym->name, currSym->group,
                                  currSym->iscommon, currSym->size,
                                  currSym->oldname, currSym->oldnumber);
        AddTranslation(linkLoc, currSym->name, newnum);
        InsertHashString(linkLoc->name, linkLoc);
      } else {
        /* global symbol with the same name already exists, maybe undefined */
        if ((currSym->iscommon) || (exSym->iscommon)) {
          /* symbol to be combined */
          exSym->iscommon |= currSym->iscommon;
          if (exSym->group != currSym->group) {
            if (exSym->group == 'U') {
              exSym->group = currSym->group;
            } else if (currSym->group != 'U') {
              if (exSym->group == 'B' &&
                  (currSym->group == 'B' || currSym->group == 'D')) {
                exSym->group = currSym->group;
              } else if (exSym->group == 'D' && currSym->group != 'B') {
                Error("symbol '%s' defined in two groups: '%c' '%c'",
                      exSym->name, currSym->group, exSym->group);
                return false;
              }
            }
          }
          if (exSym->size == 0) {
            exSym->size = currSym->size;
          }
          if ((exSym->size != currSym->size) && (currSym->size != 0) &&
              (!noWarnSize)) {
            Warning("common symbol '%s' with different size: %u %u",
                    currSym->name, exSym->size, currSym->size);
          }
          /* Global symbols always get the same number in all linked object
             files (there are no numbers in assembled object files), so only
             only one translation is needed for them.  This speeds up the
             link by having less translation entries (i.e. the same number
             as symbol entries, not more as it would be otherwise) and allows
             using the symbol table as the translation table.  */
        } else {
          /* non-common global symbol */
          if ((exSym->group == 'U') || (currSym->group == 'U')) {
            /* one symbol is undefined */
            if (exSym->group == 'U') {
              exSym->group = currSym->group;
              exSym->iscommon = false;
            }
            if (exSym->size == 0) {
              exSym->size = currSym->size;
            }
            /* The long comment in the previous case applies here as well.  */
          } else {
            Error("duplicate symbol: '%s'", currSym->name);
            return false;
          }
        }
      }
    }
    currSym = currSym->next;
  }
  return true;
}
```

```
bool CopyObject(objectHeader *linkObj, objectHeader *obj, FILE *ifd)
{
  if (!LinkSymbols(linkObj, obj))
    return false;

  firstSection = true;
  currObj = obj;
  lineBuffer[0] = '\0';
  if (currObj->sects != NULL) {
    readArchive = false;
    yyin = ifd;
    if (yyparse() == 1) {
      /* error occurred */
      return false;
    }
  }
  ReorderHashChain();
  return true;
}


FILE *OpenLibrary(const char *fname)
{
  FILE *ifd;

  ifd = fopen(fname, "r");
  if (ifd == NULL)
    return NULL;
  lineNo = 1;
  ReadLine(ifd);
  if (strcmp(lineBuffer, AR_MAGIC) != 0) {
    Error("library file format error in '%s': magic", fname);
    return NULL;
  }
  return ifd;
}

libraryHeader *ReadLibraryHeader(FILE *ifd, const char *fname)
{
  struct arHeader hdr;
  unsigned char int4[4];
  libraryHeader *libh;
  struct stat st;
  unsigned mapDate, mapSize;
  unsigned date;
  unsigned i;
  unsigned symCnt;
  char *strp;

  libh = (libraryHeader *) malloc(sizeof(libraryHeader));
  libh->count = 0;
  libh->tab = NULL;
  libh->strtab = NULL;

  /* for some reason feof(ifd) doesn't work here.  */
  fseek(ifd, 0, SEEK_END);
  if (ftell(ifd) == AR_MAGIC_LEN) {
    return libh;
  }
  fseek(ifd, AR_MAGIC_LEN, SEEK_SET);

  if (fread(&hdr, 1, sizeof(struct arHeader), ifd) != sizeof(struct arHeader)) {
    Error("cannot read member header in '%s': %s", fname, strerror(errno));
    exit(2);
  }
  if (strncmp(hdr.magic, AR_MAGIC_ENTRY, 2) != 0) {
```

```
    Error("archive format error in ‘%s’: no magic", fname);
    exit(2);
  }
  if (strncmp((char *)&(hdr.name), AR_SYMTAB_MEMBER, AR_SYMTAB_MEMBER_LEN) == 0 &&
      hdr.name[AR_SYMTAB_MEMBER_LEN] == ’ ’) {
    mapDate = strtoul((char *) &(hdr.date), NULL, 10);
    stat(fname, &st);
    date = st.st_mtime;
    if (date > mapDate) {
      Warning("symbol table older than library - hope that’s OK");
    }
    mapSize = strtoul((char *) &(hdr.size), NULL, 10);

    if (fread(int4, 1, 4, ifd) != 4) {
      Error("cannot read symbol table length in ‘%s’: %s", fname,
            strerror(errno));
      exit(2);
    }
    symCnt = (int4[0] << 24) + (int4[1] << 16) + (int4[2] << 8) + int4[3];
    libh->tab = (libraryIndex *) malloc(sizeof(libraryIndex) * symCnt);
    for (i = 0; i < symCnt; i++) {
      if (fread(int4, 1, 4, ifd) != 4) {
        Error("cannot read symbol table entry in ‘%s’: %s", fname,
              strerror(errno));
        exit(2);
      }
      libh->tab[i].offset = (int4[0] << 24) + (int4[1] << 16) + (int4[2] << 8) +
                            int4[3];
      libh->tab[i].obj = NULL;
    }
    libh->strtab = malloc(mapSize - 4 * (symCnt + 1));
    if (fread(libh->strtab, 1, mapSize - 4 * (symCnt + 1), ifd) !=
        mapSize - 4 * (symCnt + 1)) {
      Error("cannot read string table in ‘%s’: %s", fname, strerror(errno));
      exit(2);
    }
    strp = libh->strtab;
    for (i = 0; i < symCnt; i++) {
      libh->tab[i].sym = strp;
      strp += strlen(strp)+1;
    }
    libh->count = symCnt;
    return libh;
  } else {
    Warning("no symbol table in ‘%s’", fname);
    free(libh);
    return NULL;
  }
}

void ClearLibrary(libraryHeader *lib)
{
  unsigned i;

  for (i = 0; i < lib->count; i++) {
    if (lib->tab[i].obj != NULL)
    {
      ClearObject(lib->tab[i].obj);
    }
  }
  lib->count = 0;
  xfree(lib->tab);
  lib->tab = NULL;
  xfree(lib->strtab);
  lib->strtab = NULL;
}
```

```
bool CopyLibraryMember(objectHeader *linkObj, libraryIndex *lidx, FILE *ifd,
                       const char *fname)
{
  struct arHeader hdr;
  char *name, *nameEnd;
  char *symname;
  unsigned symnum;
  char *realname;
  char *symgroup;
  bool symcommon;
  unsigned symsize;
  char *sectname;
  unsigned sectlines;
  objectHeader *obj;
  size_t len, objSize;

  name = NULL;
  fseek(ifd, lidx->offset, SEEK_SET);
  if (fread(&hdr, 1, sizeof(struct arHeader), ifd) != sizeof(struct arHeader)) {
    Error("cannot read member header in '%s': %s", fname, strerror(errno));
    exit(2);
  }
  if (strncmp(hdr.magic, AR_MAGIC_ENTRY, 2) != 0) {
    Error("archive format error in '%s': no magic", fname);
    exit(2);
  }
  objSize = strtoul((char *) &(hdr.size), NULL, 10);
  if (hdr.name[0] == '#' && hdr.name[1] == '1' && hdr.name[2] == '/' &&
      isdigit((unsigned char) hdr.name[3])) {
    len = strtoul((char *) &(hdr.name[3]), NULL, 10);
    name = (char *) malloc(len+1);
    fread(name, 1, len, ifd);
    name[len] = '\0';
    objSize -= len;
  } else {
    nameEnd = strchr(hdr.name, ' ');
    *nameEnd = '\0';
    name = StrDup(hdr.name);
  }

  if (showfiles) {
    fprintf(stderr, " -- linking %s, size: %i\n", name, objSize);
  }

  lineNo = 1;
  obj = (objectHeader *) malloc(sizeof(objectHeader));
  obj->source = NULL;
  obj->symHead = obj->symTail = NULL;
  obj->sects = NULL;

  ReadLine(ifd);
  objSize -= strlen(lineBuffer);
  if (strcmp(lineBuffer, ":MPC object file 1.0\n") != 0) {
    Error("member '%s' of archive file '%s' is no object file: no magic",
          name, fname);
    exit(2);
  }
  ReadLine(ifd);
  objSize -= strlen(lineBuffer);
  if (strncmp(lineBuffer, ":source", 7) == 0) {
    len = strlen(lineBuffer); /* including space and \n */
    lineBuffer[len - 1] = '\0'; /* chop off \n */
    obj->source = StrDup(&(lineBuffer[8]));
    ReadLine(ifd);
    objSize -= strlen(lineBuffer);
  }
  if (strcmp(lineBuffer, ":begin symbol\n") != 0) {
    Error("object format error in '%s': bsym", fname);
```

```
  ClearObject(obj);
  free(obj);
  return false;
}
ReadLine(ifd);
objSize -= strlen(lineBuffer);
while ((strcmp(lineBuffer, ":end symbol\n") != 0) && (!feof(ifd))) {
  symname = strtok(lineBuffer, "\t");
  symgroup = strtok(NULL, "\t");
  symcommon = ((strlen(symgroup) == 2) && (symgroup[1] == '*'));
  symsize = atoi(strtok(NULL, "\n"));
  DecodeSymbol(symname, &realname, &symnum);
  AddUniqueSymbol(obj, realname, symgroup[0], symcommon, symsize,
                  symname, symnum);
  free(realname);
  ReadLine(ifd);
  objSize -= strlen(lineBuffer);
}
if (feof(ifd)) {
  Error("object format error in '%s': no esym", fname);
  ClearObject(obj);
  free(obj);
  return false;
}
ReadLine(ifd);
objSize -= strlen(lineBuffer);
if (strcmp(lineBuffer, ":begin section\n") != 0) {
  Error("object format error in '%s': bsect", fname);
  ClearObject(obj);
  free(obj);
  return false;
}
ReadLine(ifd);
objSize -= strlen(lineBuffer);
while ((strcmp(lineBuffer, ":end section\n") != 0) && (!feof(ifd))) {
  sectname = strtok(lineBuffer, "\t");
  sectlines = atoi(strtok(NULL, "\n"));
  AddUniqueSection(obj, sectname, sectlines);
  ReadLine(ifd);
  objSize -= strlen(lineBuffer);
}
if (feof(ifd)) {
  Error("object format error in '%s': no esect", fname);
  ClearObject(obj);
  free(obj);
  return false;
}
ReadLine(ifd);
objSize -= strlen(lineBuffer);
if (strcmp(lineBuffer, ":end header\n") != 0) {
  Error("object format error in '%s': no ehdr", fname);
  ClearObject(obj);
  free(obj);
  return false;
}

if (!LinkSymbols(linkObj, obj))
  return false;

firstSection = true;
currObj = obj;
lidx->obj = obj;
lineBuffer[0] = '\0';
if (currObj->secs != NULL) {
  readArchive = true;
  readLimit = objSize;
  yyin = ifd;
  if (yyparse() == 1) {
```

```
      /* error occurred */
      return false;
    }
  }
  ReorderHashChain();
  return true;
}


unsigned DumpSection(objectHeader *obj, const char *sname, FILE *fd)
{
  section *curr, *prev;
  unsigned i;

  prev = NULL;
  curr = obj->sects;
  while ((curr != NULL) && (strcmp(curr->name, sname) != 0)) {
    prev = curr;
    curr = curr->next;
  }
  if (curr == NULL)
    return 0;          /* nothing to write */
  if (prev == NULL) {   /* unchain section descriptor */
    obj->sects = curr->next;
  } else {
    prev->next = curr->next;
  }
  rewind(curr->tmpfile);
  for (i = 0; i < curr->lines; i++) {
    ReadLine(curr->tmpfile);
    if (fputs(lineBuffer, fd) < 0) {
      Error("cannot write to temp file: %s", strerror(errno));
      return 2;
    }
  }
  fclose(curr->tmpfile);
  free(curr->name);
  free(curr);
  return 0;
}
```

D.2.2.6 Anpassung an den MONADS-Assembler `asmglue.s`

```
        .file   "asmglue.s"
;
; This file provides the native assembler (mpcasm) glue.
;
.section        .const_start_glue
        .global XSconst
data_segment ro,XSconst
;
        .global XSconstoffset
XSconstoffset   = $10000000
;
.section        .const_end_glue
end_segment
;
.section        .data_start_glue
        .global XSdata
data_segment ro,XSdata
;
        .global XSdataoffset
XSdataoffset    = $20000000
;
.section        .data_end_glue
end_segment
```

```
;
.section        .udata_start_glue
        .global XSudata
dummy_segment XSudata
;
.section        .udata_end_glue
end_segment
;
.section        .text_start_glue
        .global XScode
code_segment XScode
;
        .global XSstackoffset
XSstackoffset   = $30000000
;
.section        .text_end_glue
end_segment
;
```

D.2.3 Bibliotheksverwaltung `ar` und `ranlib`

D.2.3.1 `ar`-Hauptprogramm `ar.c`

```
/* This implements a BSD 4.4-like ar to be used for the MONADS-PC.
   BEWARE: as of binutils-2.8 GNU tools can only read this archive format,
   but not write it. */

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <utime.h>
#include <unistd.h>
#include <alloca.h>

#include "ar.h"
#include "ar-idx.h"
#include "utils.h"

/* Version number */
#define MAJOR 1
#define MINOR 0

#define BUFSIZE 4096   /* MUST be even */


typedef void (*arOperation)(const char *, const char *,
                            unsigned, const char *[]);
typedef enum {append, before, after} arPlacement;


char *pname;
static arOperation arOp;
arPlacement arPlace;
bool nowarnCreate;
bool preserveExtract;
bool updateExisting;
bool updateMap;
bool verbose;
bool printVersion;
```

```
char buffer[BUFSIZE];
bool archiveChanged;


static void Usage(void)
{
  fprintf(stderr,
    "usage: %s [-]{dmpqrtx}[abciosuvV] [member-name] archive-file file...\n",
    pname);
}


static void SetOperation(arOperation op)
{
  if (arOp != NULL) {
    Error("two operations specified");
    Usage();
    exit(2);
  }
  arOp = op;
}


static bool MatchMember(const char *cname, unsigned num, const char *vname[])
{
  unsigned i;

  if (vname[0] == NULL) {
    return true;
  }

  i = 0;
  while ((i < num) && (strcmp(cname, vname[i]) != 0)) {
    i++;
  }
  return (i < num);
}


static void WriteHeader(FILE *fd, arTableEntry *entry)
{
  char nbuff[16];
  char *wname;
  struct arHeader h;

  if ((strlen(entry->name) > 15) || (strchr(entry->name, ' ') != NULL)) {
    sprintf(nbuff, "#1/%u", strlen(entry->name));
    wname = nbuff;
    entry->addLen = strlen(entry->name);
  } else {
    wname = entry->name;
    entry->addLen = 0;
  }
  sprintf((char *)(&h), "%-16s%-12lu%-6lu%-6lu%-8lo%-10lu", wname, entry->date,
                        entry->uid, entry->gid, entry->mode,
                        entry->size + entry->addLen);
  memcpy(h.magic, AR_MAGIC_ENTRY, 2);
  fwrite((char *)(&h), 1, sizeof(struct arHeader), fd);
  if (wname == nbuff) {
    fwrite(entry->name, 1, strlen(entry->name), fd);
  }
}


static void AppendNew(FILE *fd, const char *name)
{
  FILE *ifd;
  arTableEntry entry;
```

```
  struct stat buf;
  size_t bread;

  if ((ifd = fopen(name, "r")) == NULL) {
    Error("cannot open ‘%s’: %s", name, strerror(errno));
    exit(2);
  }
  fstat(fileno(ifd), &buf);
  entry.name = strrchr(name, ’/’);
  if (entry.name == NULL) {
    entry.name = (char *)name;
  } else {
    entry.name++;
  }
  entry.date = buf.st_mtime;
  entry.uid = buf.st_uid;
  entry.gid = buf.st_gid;
  entry.mode = buf.st_mode;
  entry.size = buf.st_size;

  WriteHeader(fd, &entry);
  do {
    if ((bread = fread(buffer, 1, BUFSIZE, ifd)) == 0 && !feof(ifd)) {
      Error("cannot read ‘%s’: %s", name, strerror(errno));
      exit(2);
    }
    if (fwrite(buffer, 1, bread, fd) != bread) {
      Error("cannot write archive");
      exit(2);
    }
  } while (bread == BUFSIZE);
  if (((bread + entry.addLen) & 1) > 0) {
    fputc(’\012’, fd);
  }
  fclose(ifd);
}


static void CopyMember(FILE *ifd, off_t offset, size_t size, FILE *ofd)
{
  size_t blocks;
  size_t bread;

  fseek(ifd, offset, SEEK_SET);
  while (size > 0 || feof(ifd)) {
    if (size < BUFSIZE) {
      blocks = size;
    } else {
      blocks = BUFSIZE;
    }
    bread = fread(buffer, 1, blocks, ifd);
    if (bread < 0) {
      Error("cannot read member");
      exit(2);
    }
    if (fwrite(buffer, 1, bread, ofd) != bread) {
      Error("cannot write member");
      exit(2);
    }
    size -= bread;
  }
  if (size > 0) {
    Error("unexpected end of file");
    exit(2);
  }
}
```

```
static void ArDelete(const char *dummy, const char *archive, unsigned filec,
                     const char *files[])
{
  FILE *fd, *tmpfd;
  arTableEntry *curr;
  unsigned i;
  size_t size;
  bool *delflag = (bool *) alloca(filec);

  memset(delflag, 0, filec);
  fd = OpenUpdate(archive, false, nowarnCreate);
  tmpfd = tmpfile();
  ReadTable(fd);
  curr = tableHead;
  CopyMember(fd, 0, AR_MAGIC_LEN, tmpfd);
  if (hasMap) {
    size = (curr->size + sizeof(struct arHeader) + curr->addLen + 1) &
           0xfffffffe;
    CopyMember(fd, curr->headerStart, size, tmpfd);
    curr = curr->next;
  }
  while (curr != NULL) {
    i = 0;
    while ((i < filec) && strcmp(files[i], curr->name) != 0) {
      i++;
    }
    if (i >= filec) {
      size = (curr->size + sizeof(struct arHeader) + curr->addLen + 1) &
             0xfffffffe;
      CopyMember(fd, curr->headerStart, size, tmpfd);
    } else {
      delflag[i] = true;
      archiveChanged = true;
    }
    curr = curr->next;
  }
  if (verbose) {
    for (i = 0; i < filec; i++) {
      if (delflag[i]) {
        printf("d - %s\n", files[i]);
      } else {
        printf("no member named ‘%s’\n", files[i]);
      }
    }
  }
  if (archiveChanged) {
    ftruncate(fileno(fd), 0);
    rewind(fd);
    CopyMember(tmpfd, 0, ftell(tmpfd), fd);
  }
  ClearArTable();
  fclose(tmpfd);
  fclose(fd);
}


static void ArMove(const char *membername, const char *archive, unsigned filec,
                   const char *files[])
{
  Error("m operation not implemented yet");
  exit(2);
}


static void ArPrint(const char *dummy, const char *archive, unsigned filec,
                    const char *files[])
{
  FILE *fd;
```

```
  arTableEntry *curr;

  fd = OpenRead(archive);
  ReadTable(fd);
  curr = tableHead;
  if (hasMap) {
    curr = curr->next;
  }
  while (curr != NULL) {
    if (MatchMember(curr->name, filec, files)) {
      if (verbose) {
        printf("\n<member %s>\n\n", curr->name);
      }
      CopyMember(fd, curr->dataStart, curr->size, stdout);
    }
    curr = curr->next;
  }
  fclose(fd);
  ClearArTable();
}


static void ArQuickappend(const char *dummy, const char *archive,
                          unsigned filec, const char *files[])
{
  FILE *fd;
  unsigned i;

  fd = OpenUpdate(archive, false, nowarnCreate);
  fseek(fd, 0, SEEK_END);
  for (i = 0; i < filec; i++) {
    if (verbose) {
      printf("q - %s\n", files[i]);
    }
    AppendNew(fd, files[i]);
    archiveChanged = true; /* rebuild map even for quickappend */
  }
  fclose(fd);
}


static void ArReplace(const char *membername, const char *archive,
                      unsigned filec, const char *files[])
{
  FILE *fd, *tmpfd;
  arTableEntry *curr;
  unsigned i;
  size_t size;

  fd = OpenUpdate(archive, false, nowarnCreate);
  tmpfd = tmpfile();
  ReadTable(fd);
  curr = tableHead;
  CopyMember(fd, 0, AR_MAGIC_LEN, tmpfd);
  if (hasMap) {
    size = (curr->size + sizeof(struct arHeader) + curr->addLen + 1) &
           0xfffffffe;
    CopyMember(fd, curr->headerStart, size, tmpfd);
    curr = curr->next;
  }
  while (curr != NULL) {
    if (arPlace == before && strcmp(curr->name, membername) == 0) {
      for (i = 0; i < filec; i++) {
        if (verbose) {
          printf("r - %s\n", files[i]);
        }
        AppendNew(tmpfd, files[i]);
        archiveChanged = true;
```

```
      }
    }
    i = 0;
    while ((i < filec) && strcmp(files[i], curr->name) != 0) {
      i++;
    }
    if (i >= filec) {
      size = (curr->size + sizeof(struct arHeader) + curr->addLen + 1) &
             0xfffffffe;
      CopyMember(fd, curr->headerStart, size, tmpfd);
    }
    if (arPlace == after && strcmp(curr->name, membername) == 0) {
      for (i = 0; i < filec; i++) {
        if (verbose) {
          printf("r - %s\n", files[i]);
        }
        AppendNew(tmpfd, files[i]);
        archiveChanged = true;
      }
    }
    curr = curr->next;
  }
  if (arPlace == append) {
    for (i = 0; i < filec; i++) {
      if (verbose) {
        printf("r - %s\n", files[i]);
      }
      AppendNew(tmpfd, files[i]);
      archiveChanged = true;
    }
  }
  if (archiveChanged) {
    ftruncate(fileno(fd), 0);
    rewind(fd);
    CopyMember(tmpfd, 0, ftell(tmpfd), fd);
  }
  ClearArTable();
  fclose(tmpfd);
  fclose(fd);
}


static void ArTable(const char *dummy, const char *archive, unsigned filec,
                    const char *files[])
{
  FILE *fd;
  arTableEntry *curr;
  char perm[9];
  char *timebuf;

  fd = OpenRead(archive);
  ReadTable(fd);
  fclose(fd);
  curr = tableHead;
  if (hasMap) {
    curr = curr->next;
  }
  while (curr != NULL) {
    if (MatchMember(curr->name, filec, files)) {
      if (verbose) {
        timebuf = ctime(&(curr->date));
        /* only rwx bits, no file type */
        perm[0] = (curr->mode & S_IRUSR) != 0 ? 'r' : '-';
        perm[1] = (curr->mode & S_IWUSR) != 0 ? 'w' : '-';
        perm[2] = (curr->mode & S_IXUSR) != 0 ? 'x' : '-';
        perm[3] = (curr->mode & S_IRGRP) != 0 ? 'r' : '-';
        perm[4] = (curr->mode & S_IWGRP) != 0 ? 'w' : '-';
        perm[5] = (curr->mode & S_IXGRP) != 0 ? 'x' : '-';
```

```
        perm[6] = (curr->mode & S_IROTH) != 0 ? 'r' : '-';
        perm[7] = (curr->mode & S_IWOTH) != 0 ? 'w' : '-';
        perm[8] = (curr->mode & S_IXOTH) != 0 ? 'x' : '-';
        perm[9] = '\0';
        printf("%s %lu/%lu %6lu %.12s %.4s ", perm, curr->uid, curr->gid,
               curr->size, timebuf+4, timebuf+20);
      }
      printf("%s\n", curr->name);
    }
    curr = curr->next;
  }
  ClearArTable();
}


static void ArExtract(const char *dummy, const char *archive, unsigned filec,
                      const char *files[])
{
  FILE *fd, *ofd;
  arTableEntry *curr;
  struct utimbuf tb;

  fd = OpenRead(archive);
  ReadTable(fd);
  curr = tableHead;
  if (hasMap) {
    curr = curr->next;
  }
  while (curr != NULL) {
    if (MatchMember(curr->name, filec, files)) {
      if (verbose) {
        printf("x - %s\n", curr->name);
      }
      if ((ofd = fopen(curr->name, "w")) == NULL) {
        Error("cannot create output file");
        exit(2);
      }
      CopyMember(fd, curr->dataStart, curr->size, ofd);
      fclose(ofd);
      chmod(curr->name, curr->mode);
      if (preserveExtract) {
        tb.actime = curr->date;
        tb.modtime = curr->date;
        utime(curr->name, &tb);
      }
    }
    curr = curr->next;
  }
  fclose(fd);
  ClearArTable();
}


static void ArNop(const char *dummy, const char *archive, unsigned filec,
                  const char *files[])
{
}


int main(int argc, char *argv[])
{
  char *operation;
  char *membername;
  char *archive;
  char **files;
  unsigned filec;

  pname = argv[0];
```

```
if ((argc < 3) || (argv[1][0] == '\0') ||
    ((argv[1][0] == '-') && (argv[1][1] == '\0'))) {
  Usage();
  exit(2);
}
operation = argv[1];
if (operation[0] == '-') {
  operation++;
}

arOp = NULL;
arPlace = append;
nowarnCreate = false;
preserveExtract = false;
updateExisting = false;
verbose = false;
printVersion = false;
updateMap = false;

while (operation[0] != '\0') {
  switch (operation[0]) {
    /* operations */
    case 'd':
      SetOperation(ArDelete);
      break;
    case 'm':
      SetOperation(ArMove);
      break;
    case 'p':
      SetOperation(ArPrint);
      break;
    case 'q':
      SetOperation(ArQuickappend);
      break;
    case 'r':
      SetOperation(ArReplace);
      break;
    case 't':
      SetOperation(ArTable);
      break;
    case 'x':
      SetOperation(ArExtract);
      break;
    /* placement */
    case 'a':
      arPlace = after;
      break;
    case 'b': /* fall through */
    case 'i':
      arPlace = before;
      break;
    /* misc stuff */
    case 'c':
      nowarnCreate = true;
      break;
    case 'o':
      preserveExtract = true;
      break;
    case 's':
      updateMap = true;
      break;
    case 'u':
      updateExisting = true;
      break;
    case 'v':
      verbose = true;
      break;
    case 'V':
```

```
        printVersion = true;
        break;
      default:
        Error("unknown option '%c'", operation[0]);
        exit(2);
    }
    operation++;
  }

  if (printVersion) {
    fprintf(stderr, "MONADS-PC ar, version %i.%i\n", MAJOR, MINOR);
  }

  if (arOp == NULL && updateMap) {
    arOp = ArNop;
  }
  /* operation is known now, let's do something */
  if (arOp == NULL) {
    if (printVersion) {
      exit(0);
    } else {
      Error("no operation specified");
      Usage();
      exit(2);
    }
  }
  if (((arOp == ArMove) || (arOp == ArReplace)) && (arPlace != append)) {
    membername = argv[2];
    archive = argv[3];
    files = &(argv[4]);
    filec = argc - 4;
  } else {
    membername = NULL;
    archive = argv[2];
    files = &(argv[3]);
    filec = argc - 3;
  }

  tableHead = tableTail = NULL;
  archiveChanged = false;

  arOp(membername, archive, filec, (const char **) files);

  if (updateMap || (hasMap && archiveChanged)) {
    Ranlib(archive);
  }

  return 0;
}
```

D.2.3.2 Definition des Bibliotheksformats `ar.h`

```
/* ar archive file definitions.  */

#ifndef _AR_H
#define _AR_H

#define AR_MAGIC "!<arch>\012"
#define AR_MAGIC_LEN 8

#define AR_MAGIC_ENTRY "'\012"

#define AR_SYMTAB_MEMBER "__.SYMDEF"
#define AR_SYMTAB_MEMBER_LEN 9
```

```
struct arHeader {
  char name[16];    /* member name */
  char date[12];    /* member mtime */
  char uid[6];      /* member uid */
  char gid[6];      /* member gid */
  char mode[8];     /* member mode, octal */
  char size[10];    /* member size */
  char magic[2];    /* entry magic */
};

#endif /* _AR_H */
```

D.2.3.3 ar-Indexverwaltung `ar-idx.h`

```
#ifndef _RANLIB_IDX_H
#define _RANLIB_IDX_H


#include <sys/types.h>
#include "utils.h"


typedef struct _arTableEntry {
  char *name;
  size_t addLen; /* addition size required for name */
  time_t date;
  uid_t uid;
  gid_t gid;
  mode_t mode;
  off_t size;
  off_t headerStart;
  off_t dataStart; /* file position where data starts */
  struct _arTableEntry *next;
} arTableEntry;


extern arTableEntry *tableHead, *tableTail;
extern bool hasMap;


FILE *OpenRead(const char *);
FILE *OpenUpdate(const char *, bool, bool);
void ReadTable(FILE *);
void ClearArTable(void);
void Ranlib(const char *);

#endif /* _RANLIB_IDX_H */
```

D.2.3.4 ar-Indexverwaltung `ar-idx.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "ar.h"
#include "ar-idx.h"
#include "utils.h"
```

```
#define BUFSIZE 4096


bool hasMap;
arTableEntry *tableHead, *tableTail;
char buffer[BUFSIZE];


FILE *OpenRead(const char *archive)
{
  FILE *fd;
  char arMbuf[AR_MAGIC_LEN+1];

  fd = fopen(archive, "r");
  if (fd == NULL) {
    Error("cannot open archive file: %s", strerror(errno));
    exit(2);
  }
  fgets(arMbuf, sizeof(arMbuf),  fd);
  if (strncmp(arMbuf, AR_MAGIC, AR_MAGIC_LEN) != 0) {
    Error("this is not an archive file");
    exit(2);
  }
  return fd;
}


FILE *OpenUpdate(const char *archive, bool nocreate, bool nowarnCreate)
{
  FILE *fd;
  char arMbuf[AR_MAGIC_LEN+1];

  fd = fopen(archive, "r+");
  if (fd == NULL) {
    if (nocreate) {
      Error("cannot open archive: %s", strerror(errno));
      exit(2);
    }
    if (errno == ENOENT) {
      if (!nowarnCreate) {
        Warning("creating archive file");
      }
      if ((fd = fopen(archive, "w+")) == NULL) {
        Error("cannot create archive file: %s", strerror(errno));
        exit(2);
      }
      fputs(AR_MAGIC, fd);
      fflush(fd);
      fseek(fd, 0, SEEK_SET);
    } else {
      Error("cannot open archive file: %s", strerror(errno));
      exit(2);
    }
  } else {
    fgets(arMbuf, sizeof(arMbuf),  fd);
    if (strncmp(arMbuf, AR_MAGIC, AR_MAGIC_LEN) != 0) {
      Error("this is not an archive file");
      exit(2);
    }
  }
  return fd;
}


void ReadTable(FILE *fd)
{
  struct arHeader entry;
```

```
  arTableEntry *nentr;
  int filepos;
  size_t bread;
  size_t namelen;
  char *nameEnd;

  hasMap = false;
  fseek(fd, AR_MAGIC_LEN, SEEK_SET);
  filepos = AR_MAGIC_LEN;
  while (!feof(fd)) {
    if ((bread = fread(&entry, 1, sizeof(entry), fd)) != sizeof(entry)) {
      if (bread == 0)   /* at eof, fseek did not set eof indicator */
        break;
      Error("format or read error");
      exit(2);
    }
    if (strncmp(entry.magic, AR_MAGIC_ENTRY, 2) != 0) {
      Error("no magic after entry header, format error");
      exit(2);
    }
    nentr = (arTableEntry *) malloc(sizeof(arTableEntry));
    nentr->date = strtol(entry.date, NULL, 10);
    nentr->uid = strtol(entry.uid, NULL, 10);
    nentr->gid = strtol(entry.gid, NULL, 10);
    nentr->mode = strtol(entry.mode, NULL, 8);
    nentr->size = strtol(entry.size, NULL, 0);
    nentr->headerStart = filepos;

    filepos += sizeof(entry);

    if (entry.name[0] == '#' && entry.name[1] == '1' && entry.name[2] == '/'
        && isdigit((unsigned char) entry.name[3])) {
      /* long name or name containing spaces */
      namelen = atoi(&(entry.name[3]));
      nentr->addLen = namelen;
      nentr->size -= namelen;
      nentr->name = (char *) malloc(namelen + 1);
      if (fread(nentr->name, 1, namelen, fd) != namelen) {
        Error("unexpected error while reading name");
        exit(2);
      }
      nentr->name[namelen] = '\0';
    } else {
      nentr->addLen = 0;
      nameEnd = strchr(entry.name, ' ');
      *nameEnd = '\0';
      nentr->name = StrDup(entry.name);
    }
    nentr->dataStart = filepos + nentr->addLen;
    filepos += (nentr->size + nentr->addLen + 1) & 0xfffffffe;
    fseek(fd, filepos, SEEK_SET);
    if (strcmp(entry.name, AR_SYMTAB_MEMBER) == 0)
      hasMap = true;
    nentr->next = NULL;
    if (tableHead == NULL) {
      tableHead = nentr;
      tableTail = nentr;
    } else {
      tableTail->next = nentr;
      tableTail = nentr;
    }
  }
}


void ClearArTable(void)
{
  arTableEntry *curr, *del;
```

```
  curr = tableHead;
  while (curr != NULL) {
    del = curr;
    free(del->name);
    curr = del->next;
    free(del);
  }
  tableHead = tableTail = NULL;
}


void Ranlib(const char *name)
{
  FILE *fd, *tmpfd;
  unsigned firstStart;
  unsigned symCnt;
  unsigned symAlloc;
  unsigned int *symPtr;
  char *symname;
  char *symgroup;
  unsigned strAlloc;
  char *strPtr;
  unsigned strIdx;
  unsigned size, bread;
  int namelen;
  arTableEntry *curr;
  struct arHeader hdr;
  time_t symtabdate;
  unsigned i;
  unsigned char int4[4];
  struct stat st;

  fd = OpenUpdate(name, true, false);
  ReadTable(fd);

  symCnt = 0;
  symAlloc = 0;
  symPtr = NULL;
  strAlloc = 0;
  strPtr = NULL;
  strIdx = 0;

  /* copy all members to temporary file while calculating the index */
  tmpfd = tmpfile();

  curr = tableHead;
  if (hasMap)
    curr = curr->next;
  if (curr != NULL) {
    firstStart = curr->headerStart;
  } else {
    firstStart = 0;
  }
  while (curr != NULL) {
    fseek(fd, curr->headerStart, SEEK_SET);
    if (fread(buffer, 1, curr->dataStart-curr->headerStart, fd) !=
        (unsigned) curr->dataStart-curr->headerStart) {
      Error("cannot read member header: %s", strerror(errno));
      exit(2);
    }
    if (fwrite(buffer, 1, curr->dataStart-curr->headerStart, tmpfd) !=
        (unsigned) curr->dataStart-curr->headerStart) {
      Error("cannot write to temp file: %s", strerror(errno));
      exit(2);
    }
    if (curr->size > 21 && fread(buffer, 1, 21, fd) == 21 &&
        strncmp(buffer, ":MPC object file 1.0\n", 21) == 0) {
```

```
    /* object file, read header */
    AppendLine(""); /* initialise lineBuffer */
    ReadLine(fd);
    if (strncmp(lineBuffer, ":source", 7) == 0) {
      ReadLine(fd);
    }
    if (strcmp(lineBuffer, ":begin symbol\n") != 0) {
      Error("object format error: bsym");
      exit(2);
    }
    ReadLine(fd);
    while ((strcmp(lineBuffer, ":end symbol\n") != 0) && (!feof(fd))) {
      symname = strtok(lineBuffer, "\t");
      symgroup = strtok(NULL, "\t");
      if (isupper((unsigned) symgroup[0]) && (symgroup[0] != 'U')) {
        if (symCnt >= symAlloc) {
          symAlloc += 1000;
          symPtr = (unsigned *) realloc(symPtr, symAlloc*4);
        }
        symPtr[symCnt] = curr->headerStart - firstStart;
        symCnt++;
        namelen = strlen(symname)+1;
        if (strIdx + namelen >= strAlloc) {
          if (namelen > 1000)
            strAlloc += namelen;
          else
            strAlloc += 1000;
          strPtr = (char *) realloc(strPtr, strAlloc);
        }
        memcpy(&(strPtr[strIdx]), symname, namelen);
        strIdx += namelen;
      }
      ReadLine(fd);
    }
  }
  fseek(fd, curr->dataStart, SEEK_SET);
  size = curr->size;
  do {
    if (size > BUFSIZE)
      bread = BUFSIZE;
    else
      bread = size;
    if ((bread = fread(buffer, 1, bread, fd)) < 0) {
      Error("cannot read archive member: %s", strerror(errno));
      exit(2);
    }
    if (fwrite(buffer, 1, bread, tmpfd) != bread) {
      Error("cannot write to temp file: %s", strerror(errno));
      exit(2);
    }
    size -= bread;
  } while (size != 0);
  if (((curr->size + curr->addLen) & 1) != 0) {
    fputc('\012', tmpfd);
  }
  curr = curr->next;
}

if ((hasMap && symCnt == 0) || (symCnt > 0)) {
  /* remove map or insert new map */
  fseek(fd, AR_MAGIC_LEN, SEEK_SET);
  if (symCnt > 0) {
    firstStart = AR_MAGIC_LEN + sizeof(struct arHeader) + 4 * (symCnt + 1) +
                 strIdx;
    firstStart += firstStart & 1; /* round up to next even boundary */
    symtabdate = 0;
    sprintf((char *) &hdr, "%-16s%-12lu%-6lu%-6lu%-8lo%-10lu",
                           AR_SYMTAB_MEMBER, symtabdate, 0L, 0L, 0L,
```

```
                          4L * (symCnt + 1) + strIdx);
      memcpy(hdr.magic, AR_MAGIC_ENTRY, 2);
      fwrite((char *) &hdr, 1, sizeof(struct arHeader), fd);
      int4[0] = (symCnt >> 24) & 0xff;
      int4[1] = (symCnt >> 16) & 0xff;
      int4[2] = (symCnt >> 8) & 0xff;
      int4[3] = symCnt & 0xff;
      fwrite(int4, 1, 4, fd);
      for (i = 0; i < symCnt; i++) {
        symPtr[i] += firstStart;
        int4[0] = (symPtr[i] >> 24) & 0xff;
        int4[1] = (symPtr[i] >> 16) & 0xff;
        int4[2] = (symPtr[i] >> 8) & 0xff;
        int4[3] = symPtr[i] & 0xff;
        fwrite(int4, 1, 4, fd);
      }
      fwrite(strPtr, 1, strIdx, fd);
      if ((strIdx & 1) != 0)
        fputc('\012', fd);

    }

    /* copy all archive members from temp file to archive */
    rewind(tmpfd);
    do {
      if ((bread = fread(buffer, 1, BUFSIZE, tmpfd)) < 0) {
        Error("cannot read temp file: %s", strerror(errno));
        exit(2);
      }
      if (fwrite(buffer, 1, bread, fd) != bread) {
        Error("cannot write to archive: %s", strerror(errno));
        exit(2);
      }
    } while (bread == BUFSIZE);

    if (symCnt > 0) {
      /* rewrite symbol table header to make sure that it is newer than
         the archive time stamp.  */
      fflush(fd);
      fseek(fd, AR_MAGIC_LEN, SEEK_SET);
      fstat(fileno(fd), &st);
      symtabdate = st.st_mtime + 3; /* increase this if necessary */
      sprintf((char *) &hdr, "%-16s%-12lu%-6lu%-6lu%-8lo%-10lu",
                             AR_SYMTAB_MEMBER, symtabdate, 0L, 0L, 0L,
                             4L * (symCnt + 1) + strIdx);
      memcpy(hdr.magic, AR_MAGIC_ENTRY, 2);
      fwrite((char *) &hdr, 1, sizeof(struct arHeader), fd);
    }
  }
  fclose(fd);
  fclose(tmpfd);
  ClearArTable();
  xfree(symPtr);
  xfree(strPtr);
}
```

D.2.3.5 `ranlib`-Hauptprogramm `ranlib.c`

```
#include <stdio.h>

#include "utils.h"
#include "ar-idx.h"
#include "getopt.h"


/* Version number */
```

```
#define MAJOR 1
#define MINOR 0


char *pname;


int main(int argc, char *argv[])
{
  bool printVer = false;         /* print version number */
  int c;                         /* current option */
  int errflg = 0;                /* > 0 means there are errors in the cmdline */

  pname = argv[0];

  while ((c = getopt(argc, argv, "V")) != EOF) {
    switch (c) {
      case 'V':
        printVer = true;
        break;
      default:
        errflg++;
    }
  }
  if (optind != argc-1) {
    errflg++;
  }

  if (errflg || printVer) {
    fprintf(stderr, "MONADS-PC pseudo-archive indexer version %i.%i\n", MAJOR, MINOR);
  }
  if (errflg) {
    fprintf(stderr, "usage: %s [-V] file\n", pname);
    exit(2);
  }

  Ranlib(argv[argc-1]);

  return 0;
}
```

D.2.4 Stub-Generator `stubgen`

D.2.4.1 Hauptprogramm `stubgen.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#include "utils.h"
#include "parser.h"
#include "getopt.h"


/* Version number */
#define MAJOR 1
#define MINOR 0


char *pname;            /* the name of this program */
char *outname;          /* the stub file we are supposed to make */
char *inname;           /* the class definition we are processing */
```

```
bool printver;


bool Stubgen(void)
{
  FILE *ifd;

  if ((ifd = fopen(inname, "r")) == NULL) {
    Error("cannot open input file '%s': %s", inname, strerror(errno));
    return false;
  }
  lineNo = 1;
  yyin = ifd;
  if (yyparse() == 1) {
    Error("syntax error in %s", inname);
    fclose(ifd);
    return false;
  }
  fclose(ifd);
  return true;
}


int main(int argc, char *argv[])
{
  extern int yydebug;
  int c;                    /* current option */
  int errflg = 0;           /* > 0 means there are errors in the cmdline */

  pname = argv[0];

  while ((c = getopt(argc, argv, "dV")) != EOF) {
    switch (c) {
      case 'd':
        yydebug = 1;
        break;
      case 'V':
        printver = true;
        break;
      case '?':
      default:
        errflg++;
    }
  }

  /* Require exactly two non-option arguments, one for the name of the C
     header file to be converted and a second one for the basename of
     the generated stub files.  */
  if (optind != argc - 2)
    errflg++;

  if (errflg || printver) {
    fprintf(stderr, "MONADS-PC C stub generator version %i.%i\n", MAJOR, MINOR);
    if (errflg) {
      fprintf(stderr, "usage: %s [-V] infile outfile-prefix\n", pname);
      exit(2);
    }
  }

  inname = argv[optind];
  outname = argv[optind + 1];
  if (!Stubgen()) {
    exit(1);
  }
  return 0;
}
```

D.2.4.2 Scanner `sg-lex.l`

```
/* definitions */

%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include "utils.h"
#include "sg-parse.h"

static int ProcessReserved(const char *);

extern char *pname;
%}

%option 8bit
%option case-sensitive
%option never-interactive
%option nounput
%option noyywrap

WS      [ \t]+
LID     [A-Za-z_][0-9A-Za-z_]*
ONUM    "0"[1-9][0-9]*
DNUM    [1-9][0-9]*
HEXNUM  "0x"[0-9A-Fa-f]+


%%

{WS}                    { /* ignore whitespace */ }
{ONUM}|{DNUM}|{HEXNUM}  { yylval.value = strtol(yytext, NULL, 0); return NUM; }
{LID}                   { return ProcessReserved(yytext); }
"#ifndef"               { return IFNDEF; }
"#endif"                { return ENDIF; }
"#include"              { return INC; }
"#define"               { return DEF; }
"mpc/mpc\.h"            { return MPCH; }
"*"+                    { yylval.value = strlen(yytext); return STARS; }
"/*"[.\n]*"*/"          { /* ignore comments */ }
"\n"                    { return yytext[0]; }
.                       { return yytext[0]; }
<<EOF>>                 { return yytext[0]; }

%%

/* user subroutines */

static int ProcessReserved(const char *id)
{
  if (strcmp(id, "CLASS") == 0) {
    return CNAME;
  } else if (strcmp(id, "METHOD") == 0) {
    return METHOD;
  } else if (strcmp(id, "OPEN_METHOD") == 0) {
    return METHOD;
  } else if (strcmp(id, "CLOSE_METHOD") == 0) {
    return METHOD;
  }
  yylval.text = StrDup(id);
  return ID;
}
```

D.2.4.3 Parser `sg-parse.y`

```
/* MONADS-PC stub generator parser.  */

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <assert.h>
#include <errno.h>

#include "utils.h"

#define YYDEBUG 1

typedef enum { VOID_TYPE, INT_TYPE, FLOAT_TYPE, STRING_TYPE, MODCAP_TYPE }
  type_enum;

/* Constructed as a doubly linked circular list, then changed into a dll.  */
struct paramList {
  type_enum type;
  bool byreference;
  struct paramList *next;
  struct paramList *prev;
};

extern int yylex(void);

const char *typename[MODCAP_TYPE+1][2] =
  { { "void ", NULL },
    { "int ", "int *" },
    { "float ", "float *" },
    { "char *", "char **" },
    { "modcap ", "modcap *" } };

extern unsigned lineNo;
extern char *outname;
char *className;
static void yyerror(const char *);
static bool OutputMethod(int, char *, struct paramList *);
static struct paramList *Parm(char *, int, struct paramList *);
%}

%union {
char *text;
unsigned value;
struct paramList *list;
}

%token <text> ID
%token <value> NUM
%token IFNDEF
%token ENDIF
%token INC
%token MPCH
%token DEF
%token CNAME
%token METHOD
%token <value> STARS

%type <list> idl

%%

header:   input

input:    ws guard ws include ws classn class endgrd ws
```

```
;

ws:       /* empty */
        | ws '\n'                               { lineNo++; }
;

guard:    IFNDEF ID '\n' DEF ID '\n'            { lineNo += 2; }
;

include:  INC '<' MPCH '>' '\n'                 { lineNo++; }
;

classn:   DEF CNAME ID '\n'                     { lineNo++; className = $3; }
;

class:    method
        | '\n'                                  { lineNo++; }
        | class method
        | class '\n'                            { lineNo++; }
;

method:  METHOD '(' NUM ',' ID ',' idl ')' '\n' { if (OutputMethod($3,$5,$7))
                                                    YYABORT;
                                                  lineNo++; }
;

idl:      ID                                    { if (!($$=Parm($1, 0, NULL)))
                                                    YYABORT; }
        | ID STARS                              { if (!($$=Parm($1, $2, NULL)))
                                                    YYABORT; }
        | idl ',' ID                            { if (!($$=Parm($3, 0, $1)))
                                                    YYABORT; }
        | idl ',' ID STARS                      { if (!($$=Parm($3, $4, $1)))
                                                    YYABORT; }
;

endgrd:   ENDIF '\n'                            { lineNo++; }
;

%%

static void yyerror(const char *msg)
{
  Error("%s in line %u", msg, lineNo);
}

/* Generate the stub code file, one for each interface function.  The code
   can only handle 4 types: int, float, char * and modcap, though it should
   be easy to add things like semaphores or something simple structured.
   What it can do:
    - pass int/float by value and reference
    - pass char * by value and reference
    - pass modcap by value and reference
    - return int/float, char * and modcap.  */
static bool OutputMethod(int num, char *id, struct paramList *params)
{
  struct paramList *param;
  int paramnum;
  bool modcaprecycle = false;
  bool stringrecycle = false;
  bool firststring = true;
  char *accessval;
  int laststrref = -1;
  char oname[FILENAME_MAX];
  FILE *of;

  sprintf(oname, "%s.%u.c", outname, num);
  if ((of = fopen(oname, "w")) == NULL) {
```

```
  Error("cannot create output file %s: %s", oname, strerror(errno));
  return true;
}
if (className == NULL) {
  Warning("no class name specified");
} else if (strncmp(className, id, strlen(className)) != 0) {
  Warning("class %s contains a method with different name prefix",
          className);
}
assert(params != NULL);

/* break circular list into a normal doubly linked list (the tail pointer
   is not needed, so it isn't saved.  */
params->prev->next = NULL;
params->prev = NULL;

/* generate the required #includes */
fprintf(of, "#include <stdlib.h>\n"
            "#include <string.h>\n"
            "#define _MPC_STUB\n"
            "#include <mpc/mpc.h>\n");

/* print the function declaration line */
fprintf(of, "%s%s(", typename[params->type][params->byreference], id);
if (num == 0) {
  /* create function */
  fprintf(of, "modcap *mcinst, modcap mcmgr");
} else if (num == 1) {
  /* open function */
  fprintf(of, "mcallseg *mcs, modcap mc");
} else {
  fprintf(of, "mcallseg mcs");
}
param = params->next;
paramnum = 0;
while (param != NULL) {
  if (param->type == VOID_TYPE) {
    yyerror("'void' not allowed inside parameter list");
    return true;
  }
  fprintf(of, ", %sp%u", typename[param->type][param->byreference],paramnum);
  param = param->next;
  paramnum++;
}
fprintf(of, ")\n");

/* begin of the stub function */
fprintf(of, "{\n");

/* declare some variables we are going to use */
if (num == 0 || num == 1) {
  fprintf(of, "  register int new_mcs;\n");
}
param = params->next;
while (param != NULL) {
  if (param->type == STRING_TYPE) {
    fprintf(of, "  register size_t len;\n");
    param = NULL;
  } else {
    param = param->next;
  }
}
if (params->type != VOID_TYPE) {
  fprintf(of, "  register %sretval;\n",
              typename[params->type][params->byreference]);
}

fprintf(of, "  asm(\"loadcb c11,#16\");\n");
```

```
/* print the function specific setup code */
switch (num) {
case 0: /* create */
  /* Find a module call segment we can use.  If there is a mcs available
     for reuse, take that, otherwise allocate a new one.  The mcs is left
     in the freelist, as it is only used in the create call, which
     automatically closes it.  */
  fprintf(of, "  asm(\"loadc c14,XPmcs(c11)\");\n"
              "  if (__mcs_free < 0) {\n"
              "    new_mcs = __mcs_alloc++;\n"
              "    asm(\"crshseg c11,#$8000,#1,#0,c13\\n\\t\"\n"
              "        \"storec c13,(c14)[%%0]\" :: \"D\" (new_mcs));\n"
              "    asm(\"sti %%0,(c14)[%%1]\""
                " :: \"D\" (__mcs_free), \"D\" (new_mcs*4));\n"
              "    __mcs_free = new_mcs;\n"
              "  } else {\n"
              "    new_mcs = __mcs_free;\n"
              "    asm(\"loadc c13,(c14)[%%0]\" :: \"D\" (new_mcs));\n"
              "  }\n"
              "  asm(\"loadc c14,XPmodcap(c11)\\n\\t\"\n"
              "      \"loadc c14,(c14)[%%0]\\n\\t\"\n"
              "      \"prccall (c13),(c14),(c15)[%%1],c12\""
                " :: \"D\" ((unsigned)mcmgr&0x0fffffff),\"D\" "
                "((unsigned)mcinst&0x0fffffff));\n");
  break;
case 1: /* open */
  /* Find a module call segment we can use.  If there is a mcs available
     for reuse, take that, otherwise allocate a new one.  */
  fprintf(of, "  asm(\"loadc c14,XPmcs(c11)\");\n"
              "  if (__mcs_free < 0) {\n"
              "    new_mcs = __mcs_alloc++;\n"
              "    asm(\"crshseg c11,#$8000,#1,#0,c13\\n\\t\"\n"
              "        \"storec c13,(c14)[%%0]\" :: \"D\" (new_mcs));\n"
              "  } else {\n"
              "    new_mcs = __mcs_free;\n"
              "    asm(\"ldi %%0,(c14)[%%1]\""
                " : \"=D\" (__mcs_free) : \"D\" (__mcs_free*4));\n"
              "    asm(\"loadc c13,(c14)[%%0]\" :: \"D\" (new_mcs));\n"
              "  }\n"
              "  *mcs = (mcallseg) (new_mcs | 0xf0000000);\n"
              "  asm(\"loadc c14,XPmodcap(c11)\\n\\t\"\n"
              "      \"loadc c14,(c14)[%%0]\\n\\t\"\n"
              "      \"procall (c13),(c14),c12\""
                " :: \"D\" ((unsigned)mc&0x0fffffff));\n");
  break;
case 2: /* close */
  /* Mark mcs as free.  The mcs handle should never be used afterwards,
     because it may use an invalidated entry or some other module that
     has been opened afterwards - so be careful.  */
  fprintf(of, "  asm(\"loadc c14,XPmcs(c11)\\n\\t\"\n"
              "      \"loadc c13,(c14)[%%0]\" :: \"D\" "
                "((unsigned)mcs&0x0fffffff));\n"
              "  asm(\"sti %%0,(c14)[%%1]\""
                " :: \"D\" (__mcs_free), \"D\" "
                " (((unsigned)mcs&0x0fffffff)*4));\n"
              "  __mcs_free = (unsigned)mcs&0x0fffffff;\n"
              "  asm(\"precall (c13),#%u,c12\");\n", num);
  break;
default: /* delete or some random function */
  fprintf(of, "  asm(\"loadc c13,XPmcs(c11)\\n\\t\"\n"
              "      \"loadc c13,(c13)[%%0]\\n\\t\"\n"
              "      \"precall (c13),#%u,c12\" :: \"D\" "
                "((unsigned)mcs&0x0fffffff));\n",
              num);
}
param = params->next;
paramnum = 0;
while (param != NULL) {
```

```
switch (param->type) {
case INT_TYPE: /* fall through */
case FLOAT_TYPE:
  if (!param->byreference) {
    /* int / float */
    fprintf(of, "  asm(\"pval #%u\" :: \"a\" (p%u));\n",
                paramnum, paramnum);
  } else {
    /* int* / float* */
    fprintf(of, "  switch ((unsigned)p%u&0xf0000000) {\n"
                "  case 0x10000000:\n"
                "    asm(\"movc c0,c13\");\n"
                "    break;\n"
                "  case 0x20000000:\n"
                "    asm(\"movc c1,c13\");\n"
                "    break;\n"
                "  case 0x30000000:\n"
                "    asm(\"movc c2,c13\");\n"
                "    break;\n"
                "  default:\n"
                "    asm(\"invc c13\");\n"
                "  }\n"
                "  asm(\"prefr #%u,c13,%%0,#4\""
                  " :: \"a\" ((unsigned)p%u&0x0fffffff));\n",
                paramnum, paramnum, paramnum);
    /* another option would be to store a pointer in each segment that
       points to the segments itself and use loadc c13,(cx) - but that
       doesn't work for the constant segment due to kernel limitations
       when this module is downloaded to the MONADS.  */
  }
  break;
case STRING_TYPE:
  /* char* / char** */
  if (param->byreference) {
    accessval = "*";
    laststrref = paramnum;
  } else {
    accessval = "";
  }
  /* string allocation strategy: If there is no string in the freelist
     then create a new one (exact length) - else if the first string
     in the list fits then use it.  If it doesn't fit then put it
     back *at the end* of the freelist and allocate a new string.  */
  stringrecycle = true;
  fprintf(of, "  len = strlen(%sp%u);\n"
              "  asm volatile(\"loadc c13,XPstrpool(c11)\\n\\t\"\n"
              "               \"cmpci c13\\n\\t\"\n"
              "               \"beq n%u*4\\n\\t\"\n"
              "               \"loadc c14,1(c13)\\n\\t\"\n"
              "               \"storec c14,XPstrpool(c11)\\n\\t\"\n"
              "               \"invp 1(c13)\\n\\t\"\n"
              "               \"cmpci c14\\n\\t\"\n"
              "               \"bne o%u*4\\n\\t\"\n"
              "               \"invp XPstrpoollast(c11)\\n\"\n"
              "               \"o%u:\\n\\t\"\n"
              "               \"loadc c14,(c13)\\n\\t\"\n"
              "               \"ldaln (c14)\\n\\t\"\n"
              "               \"cmpa %%0\\n\\t\"\n"
              "               \"bhs p%u*4\\n\\t\"\n"
              "               \"loadc c14,XPstrpoollast(c11)\\n\\t\"\n"
              "               \"cmpci c14\\n\\t\"\n"
              "               \"bne q%u*4\\n\\t\"\n"
              "               \"storec c13,XPstrpool(c11)\\n\\t\"\n"
              "               \"bunc r%u*4\\n\"\n"
              "               \"q%u:\\n\\t\"\n"
              "               \"storec c13,1(c14)\\n\"\n"
              "               \"r%u:\\n\\t\"\n"
              "               \"storec c13,XPstrpoollast(c11)\\n\"\n"
```

```
                "               \"n%u:\\n\\t\"\n"
                "               \"crshseg c11,#0,%%0,#2,c13\\n\\t\"\n"
                "               \"movc c13,c14\\n\\t\"\n"
                "               \"storec c13,(c14)\\n\"\n"
                "               \"p%u:\" :: \"D\" (len+4) : \"a\");\n"
                "  asm volatile(\"sta 0(c14)\\n\\t\"\n"
                "               \"bmovb (cx)[%%0],4(c14)\\n\\t\"\n"
                "               \"pref #%u,c13\\n\\t\"\n"
                "               \"loadc c14,XPstrtmplast(c11)\\n\\t\"\n"
                "               \"cmpci c14\\n\\t\"\n"
                "               \"bne s%u*4\\n\\t\"\n"
                "               \"storec c14,XPstrtmpalloc(c11)\\n\\t\"\n"
                "               \"bunc t%u*4\\n\"\n"
                "               \"s%u:\\n\\t\"\n"
                "               \"storec c13,1(c14)\\n\"\n"
                "               \"t%u:\\n\\t\"\n"
                "               \"storec c13,XPstrtmplast(c11)\\n\\t\""
                  " :: \"D\" (%sp%u), \"a\" (len) : \"a\");\n",
                accessval, paramnum, paramnum, paramnum,
                paramnum, paramnum, paramnum, paramnum, paramnum, paramnum,
                paramnum, paramnum, paramnum, paramnum, paramnum, paramnum,
                accessval, paramnum);
    break;
  case MODCAP_TYPE:
    if (!param->byreference) {
      /* modcap */
      modcaprecycle = true;
      fprintf(of, "  asm(\"loadc c13,XPmcpool(c11)\\n\\t\"\n"
                  "      \"cmpci c13\\n\\t\"\n"
                  "      \"beq n%u*4\\n\\t\"\n"
                  "      \"invcap (c13)\\n\\t\"\n"
                  "      \"loadc c14,0(c13)\\n\\t\"\n"
                  "      \"storec c14,XPmcpool(c11)\\n\\t\"\n"
                  "      \"bunc o%u*4\\n\"\n"
                  "      \"n%u:\\n\\t\"\n"
                  "      \"crshseg c11,#$4000,#1,#1,c13\\n\"\n"
                  "      \"o%u:\\n\\t\"\n"
                  "      \"loadc c14,XPmctmplast(c11)\\n\\t\"\n"
                  "      \"cmpci c14\\n\\t\"\n"
                  "      \"bne p%u*4\\n\\t\"\n"
                  "      \"storec c14,XPmctmpalloc(c11)\\n\"\n"
                  "      \"p%u:\\n\\t\"\n"
                  "      \"storec c13,(c14)\\n\\t\"\n"
                  "      \"storec c13,XPmctmplast(c11)\\n\\t\"\n"
                  "      \"loadc c14,XPmodcap(c11)[%%0]\\n\\t\"\n"
                  "      \"movcap (c14),(c13)\\n\\t\"\n"
                  "      \"pref #%u,c13\" :: \"D\" "
                    "((unsigned)p%u&0x0fffffff));\n",
                  paramnum, paramnum, paramnum, paramnum, paramnum,
                  paramnum, paramnum, paramnum);
    } else {
      /* modcap* */
      fprintf(of, "  asm(\"loadc c13,XPmodcap(c11)[%%0]\\n\\t\"\n"
                  "  asm(\"pref #%u,c13\" :: \"D\" "
                    "((unsigned)(*p%u)&0x0fffffff));\n",
                  paramnum, paramnum);
    }
    break;
  default:
    yyerror("internal: I don't know how to handle this type");
    return true;
  }
  param = param->next;
  paramnum++;
}
if (params->type == INT_TYPE || params->type == FLOAT_TYPE) {
  if (params->byreference) {
    Error("'%s' not supported as return type in line %u",
```

```
          typename[params->type][params->byreference], lineNo);
    return true;
  }
  /* i3 has to be saved, too, as the compiler might need the frame
     pointer, but has no clue how to save it around the call.  */
  fprintf(of, "  asm volatile(\"sti i0,___i0save(cx)\\n\\t\"\n"
              "               \"sti i3,___i3save(cx)\\n\\t\"\n"
              "               \"call\\n\\t\"\n"
              "               \"ldi i3,___i3save(cx)\\n\\t\"\n"
              "               \"ldi i0,___i0save(cx)\""
                " : \"=a\" (retval) :: \"ax\",\"i1\",\"i2\");\n");
} else {
  if (params->type == MODCAP_TYPE) {
    /* handle modcap return values - passed as last argument */
     fprintf(of, "  retval = modcap_alloc();\n"
                 "  asm(\"loadc c13,XPmodcap(c11)[%%0]\\n\\t\"\n"
                 "      \"pref %u,c13\" :: \"D\" "
                   "((unsigned)retval&0x0fffffff));\n",
                 paramnum);
  } else if (params->type == STRING_TYPE) {
    /* handle string return values - pass an empty string as last argument */
    stringrecycle = true;
    fprintf(of, "  asm(\"loadc c13,XPstrpool(c11)\\n\\t\"\n"
                "      \"cmpci c13\\n\\t\"\n"
                "      \"beq n%u*4\\n\\t\"\n"
                "      \"loadc c14,1(c13)\\n\\t\"\n"
                "      \"storec c14,XPstrpool(c11)\\n\\t\"\n"
                "      \"cmpci c14\\n\\t\"\n"
                "      \"bne o%u*4\\n\\t\"\n"
                "      \"invp XPstrpoollast(c11)\\n\\t\"\n"
                "      \"bunc o%u*4\\n\"\n"
                "      \"n%u:\\n\\t\"\n"
                "      \"crshseg c11,#0,%%0,#2,c13\\n\\t\"\n"
                "      \"movc c13,c14\\n\\n\"\n"
                "      \"storec c13,(c14)\\n\"\n"
                "      \"o%u:\\n\\t\"\n"
                "      \"clr (c14)\\n\\t\"\n"
                "      \"pref #%u,c13\""
                "      \"loadc c14,XPstrtmplast(c11)\\n\\t\"\n"
                "      \"cmpci c14\\n\\t\"\n"
                "      \"bne p%u*4\\n\\t\"\n"
                "      \"storec c14,XPstrtmpalloc(c11)\\n\\t\"\n"
                "      \"bunc q%u*4\\n\"\n"
                "      \"p%u:\\n\\t\"\n"
                "      \"storec c13,1(c14)\\n\"\n"
                "      \"q%u:\\n\\t\"\n"
                "      \"storec c13,XPstrtmplast(c11)\\n\\t\"\n"
                  " :: \"D\" (len+4));\n",
               paramnum, paramnum, paramnum, paramnum, paramnum, paramnum,
               paramnum, paramnum, paramnum, paramnum);
  }
  /* i3 has to be saved, too, as the compiler might need the frame
     pointer, but has no clue how to save it around the call.  */
  fprintf(of, "  asm volatile(\"sti i0,___i0save(cx)\\n\\t\"\n"
              "               \"sti i3,___i3save(cx)\\n\\t\"\n"
              "               \"call\\n\\t\"\n"
              "               \"ldi i3,___i3save(cx)\\n\\t\"\n"
              "               \"ldi i0,___i0save(cx)\""
                " ::: \"a\", \"ax\", \"i1\", \"i2\");\n");
}

/* copy the strings passed by reference */
param = params->next;
paramnum = 0;
while (stringrecycle && param != NULL) {
  if (param->type == STRING_TYPE) {
    if (param->byreference) {
      if (firststring) {
```

```
        fprintf(of, "  asm(\"loadc c13,XPstrtmpalloc(c11)\\n\\t\"\n"
                    "      \"loadc c14,(c13)\\n\\t\"\n"
                    "      \"lda (c14)\" : \"=a\" (len) : );\n"
                    "  *p%u = (char *) malloc(len+1);\n"
                    "  asm volatile(\"bmovb 4(c14),(cx)[%%0]\""
                      " :: \"D\" (*p%u), \"a\" (len) : \"a\");\n"
                    "  *p%u[len] = '\\0';\n",
                    paramnum, paramnum, paramnum);
        firststring = false;
      } else {
        fprintf(of, "  asm(\"loadc c13,1(c13)\\n\\t\"\n"
                    "      \"loadc c14,(c13)\\n\\t\"\n"
                    "      \"lda (c14)\" : \"=a\" (len) : );\n"
                    "  *p%u = (char *) malloc(len+1);\n"
                    "  asm volatile(\"bmovb 4(c14),(cx)[%%0]\""
                      " :: \"D\" (*p%u), \"a\" (len) : \"a\");\n"
                    "  *p%u[len] = '\\0';\n",
                    paramnum, paramnum, paramnum);
      }
    } else if (paramnum <= laststrref) {
      /* Strings passed by value - just skip over parameters.  Omit
         the skips after the last string by reference parameter.  */
      /* This could be optimised by having another chain of parameters
         that really need to be copied out.  But this costs space and
         time, so it is not clear if it is worth the effort.  */
      if (firststring) {
        fprintf(of, "  asm(\"loadc c13,XPstrtmpalloc(c11)\");\n");
      } else {
        fprintf(of, "  asm(\"loadc c13,1(c13)\");\n");
      }
    }
  }
  param = param->next;
  paramnum++;
}
if (params->type == STRING_TYPE) {
  if (firststring) {
    fprintf(of, "  asm(\"loadc c13,XPstrtmpalloc(c11)\\n\\t\"\n"
                "      \"loadc c14,(c13)\\n\\t\"\n"
                "      \"lda (c14)\" : \"=a\" (len) : );\n"
                "  retval = (char *) malloc(len+1);\n"
                "  asm volatile(\"bmovb 4(c14),(cx)[%%0]\""
                  " :: \"D\" (retval), \"a\" (len) : \"a\");\n"
                "  retval[len] = '\\0';\n");
    firststring = false;
  } else {
    fprintf(of, "  asm(\"loadc c13,1(c13)\\n\\t\"\n"
                "      \"loadc c14,(c13)\\n\\t\"\n"
                "      \"lda (c14)\" : \"=a\" (len) : );\n"
                "  retval = (char *) malloc(len+1);\n"
                "  asm volatile(\"bmovb 4(c14),(cx)[%%0]\""
                  " :: \"D\" (retval), \"a\" (len) : \"a\");\n"
                "  retval[len] = '\\0';\n");
  }
}
if (modcaprecycle) {
  /* recycle the parameter modcaps (including return value).  */
  fprintf(of, "  asm(\"loadc c14,XPmctmplast(c11)\\n\\t\"\n"
              "      \"loadc c13,XPmcpool(c11)\\n\\t\"\n"
              "      \"storec c13,(c14)\\n\\t\"\n"
              "      \"loadc c14,XPmctmpalloc(c11)\\n\\t\"\n"
              "      \"storec c14,XPmcpool(c11)\\n\\t\"\n"
              "      \"invp XPmctmpalloc(c11)\\n\\t\"\n"
              "      \"invp XPmctmplast(c11)\");\n");
}
if (stringrecycle) {
  /* recycle parameter strings (including the return value).  */
  fprintf(of, "  asm(\"loadc c14,XPstrtmpalloc(c11)\\n\\t\"\n"
```

```
                "      \"loadc c13,XPstrpool(c11)\\n\\t\"\n"
                "      \"storec c14,XPstrpool(c11)\\n\\t\"\n"
                "      \"loadc c14,XPstrtmplast(c11)\\n\\t\"\n"
                "      \"storec c13,1(c14)\\n\\t\"\n"
                "      \"cmpci c13\\n\\t\"\n"
                "      \"bne z0*4\\n\\t\"\n"
                "      \"storec c14,XPstrpoollast(c11)\\n\"\n"
                "      \"z0:\\n\\t\"\n"
                "      \"invp XPstrtmpalloc(c11)\\n\\t\"\n"
                "      \"invp XPstrtmplast(c11)\");\n");
  }
  if (params->type != VOID_TYPE) {
    fprintf(of, "  return retval;\n");
  }
  fprintf(of, "}\n");
  fclose(of);
  return false;
}

/* Store the type in the list of function parameters (the return value
   counts as a parameter here, it is handled as a special case in the
   code above.  */
static struct paramList *Parm(char *id, int stars, struct paramList *rest)
{
  struct paramList *newparam;
  type_enum type;

  if (strcmp(id, "int") == 0) {
    type = INT_TYPE;
  } else if (strcmp(id, "float") == 0) {
    type = FLOAT_TYPE;
  } else if (strcmp(id, "char") == 0) {
    type = STRING_TYPE;
    stars--;
  } else if (strcmp(id, "modcap") == 0) {
    type = MODCAP_TYPE;
  } else if (strcmp(id, "void") == 0) {
    type = VOID_TYPE;
  } else {
    Error("don't know how to handle type '%s' in line %u", id, lineNo);
    return NULL;
  }

  if (stars < 0 || stars > 1) {
    yyerror("can not handle this combination of type and stars");
    return NULL;
  }
  newparam = (struct paramList *) malloc(sizeof(struct paramList));
  if (rest == NULL) {
    newparam->next = newparam;
    newparam->prev = newparam;
  } else {
    newparam->prev = rest->prev;
    rest->prev->next = newparam;
    newparam->next = rest;
    rest->prev = newparam;
  }
  newparam->type = type;
  newparam->byreference = (stars == 1);
  return newparam->next;
}
```

D.2.5 Gemeinsame Teile

D.2.5.1 Hilfsroutinen `utils.h`

```
/* type declarations that are used everywhere */

#ifndef _UTILS_H
#define _UTILS_H

#include <stdio.h>


typedef enum {false, true} bool;

extern unsigned lineNo;
extern char *lineBuffer;
extern unsigned lineBufferAlloc;

extern char *pname;

char *StrDup(const char *);

char *choose_temp(void);
FILE *tmpfile(void);

void xfree(void *);
void ReadLine(FILE *);
void AppendLine(const char *);

void Warning(const char *, ...);
void Error(const char *, ...);


#endif /* _UTILS_H */
```

D.2.5.2 Hilfsroutinen `utils.c`

```
/* misc functions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <stdarg.h>
#include <unistd.h>

#include "utils.h"


unsigned lineNo;
char *lineBuffer;
unsigned lineBufferAlloc;

#define TEMP_TEMPLATE "keXXXXXX"

/* This implementation of tmpfile() is not optimal, but that doesn't matter,
   as tmpfile() can only called 26 times at most, after which it fails, and
   we don't even need that many runs.  It was only reimplemented, because
   Sun's implementation of tmpfile() really sucks: it only tries to use
   /var/tmp, does not care what's in the environment variable TMPDIR,
   and /var/tmp is usually too small to do anything useful with it.  */

char *choose_temp(void)
{
```

```
  char *dir = getenv ("TMPDIR");

  if (dir == NULL || access(dir, R_OK | W_OK | X_OK)) {
    dir = "/tmp";
    if (access(dir, R_OK | W_OK | X_OK)) {
      return NULL;
    }
  }
  return tempnam(dir, "ke");
}


FILE *tmpfile(void)
{
  char *template = choose_temp();
  FILE *f;

  if (strlen(template) == 0) {
    return NULL;
  }
  f = fopen(template, "w+b");
  if (f != NULL) {
    remove (template);
  }
  free(template);
  return f;
}


char *StrDup(const char *s)
{
  size_t len;
  char *dest;

  len = strlen(s) + 1;
  dest = (char *) malloc(len);
  memcpy(dest, s, len);
  return dest;
}


void xfree(void *ptr)
{
  if (ptr != NULL)
    free(ptr);
}


void ReadLine(FILE *ifd)
{
  if (lineBuffer == NULL) {
    lineBufferAlloc = 1000;
    lineBuffer = (char *) malloc(lineBufferAlloc);
  }
  fgets(lineBuffer, lineBufferAlloc, ifd);
  while ((strlen(lineBuffer) == lineBufferAlloc-1) &&
         (lineBuffer[lineBufferAlloc-2] != '\n')) {
    lineBufferAlloc += 1000;
    lineBuffer = realloc(lineBuffer, lineBufferAlloc);
    fgets(&(lineBuffer[lineBufferAlloc-1-1000]), 1001, ifd);
  }
  lineNo++;
}


void AppendLine(const char *text)
{
  size_t newlen, len;
```

```
  newlen = strlen(text);
  if (lineBuffer == NULL) {
    if (newlen + 1 > 500) {
      lineBufferAlloc = newlen + 1;
    } else {
      lineBufferAlloc = 500;
    }
    lineBuffer = (char *) malloc(lineBufferAlloc);
    lineBuffer[0] = '\0';
  } else {
    len = strlen(lineBuffer);
    if (len + newlen + 1 > lineBufferAlloc) {
      if (newlen > 500) {
        lineBufferAlloc += newlen;
      } else {
        lineBufferAlloc += 500;
      }
      lineBuffer = (char *) realloc(lineBuffer, lineBufferAlloc);
    }
  }
  if (lineBuffer == NULL) {
    fprintf(stderr, "%s: fatal: out of memory, line too long\n", pname);
    exit(2);
  }
  strcat(lineBuffer, text);
}


static bool headline = false;


void Warning(const char *text, ...)
{
  char errmsg[1000];
  va_list ap;

  if (!headline) {
    fprintf(stderr, "%s:\n", pname);
    headline = true;
  }
  va_start(ap, text);
  vsprintf(errmsg, text, ap);
  va_end(ap);
  fprintf(stderr, "  warning: %s\n", errmsg);
}


void Error(const char *text, ...)
{
  char errmsg[1000];
  va_list ap;

  if (!headline) {
    fprintf(stderr, "%s:\n", pname);
    headline = true;
  }
  va_start(ap, text);
  vsprintf(errmsg, text, ap);
  va_end(ap);
  fprintf(stderr, "  error: %s\n", errmsg);
}
```

D.2.5.3 Parser- und Scannerdeklarationen `parser.h`

```
/* Definitions for bison/yacc parsers */

#ifndef _PARSER_H
#define _PARSER_H

#include <stdio.h>

extern FILE *yyin;
extern int yydebug;
extern int yyparse(void);

#endif /* _PARSER_H */
```

D.3 C-Bibliothek

Die Portierung der C-Bibliothek ist bisher sehr unvollständig, deshalb werden hier nur ausgewählte Teile abgedruckt. Die übrigen Teile der Portierung sind von anderen entweder unverändert oder nur leicht modifiziert übernommen worden. Eine Überarbeitung ist aber sicher noch erforderlich. Als Basis diente eine Vorversion der GNU C-Bibliothek Version 2.1, datiert vom 19. Oktober 1997. Diese Version ist noch nicht sehr ausgereift. Es waren einige kleine Anpassungen an den portablen Teilen erforderlich, die hier nicht abgedruckt sind.

D.3.1 Verschiedenes

D.3.1.1 C-Startcode `start.s`

```
        .file   "crt1.s"
;
; This file provides the C startup code (and a bit of assembler glue).
;
.section        .itext
        .global __crt_dummy
__crt_dummy:
        causex  #40960000,#0,#0
;
        .global XIopen
interface_procedure XIopen,1
parameters 6
locals 6
begin
;
XMstdin         = 0
XMstdout        = 1
XMstderr        = 2
XMrootdir       = 3
XMcurrdir       = 4
XMenv           = 5
;
        .global XPmodcap
XPmodcap        = 0     ; array where all modcaps are stored
        .global XPmctmpalloc
XPmctmpalloc    = 1     ; pointer to list of mc allocated for current funcall
        .global XPmctmplast
XPmctmplast     = 2     ; pointer to last element of the above list
;
        .global XPmcs
XPmcs           = 3     ; array where pointers to all mcs are stored
;
        .global XPstrpool
XPstrpool       = 4     ; pointer to free string list (for parameters)
        .global XPstrpoollast
XPstrpoollast   = 5     ; pointer to last element of the above list
        .global XPstrtmpalloc
XPstrtmpalloc   = 6     ; pointer to list of strings allocated for curr funcall
        .global XPstrtmplast
XPstrtmplast    = 7     ; pointer to last element of the above list
;
        .global XPheap
XPheap          = 8
XPstack         = 9
;
        loadcb  c14,#0                  ; parameter list
```

```
        crrseg  #0,#0,#20,c15           ; first retained segment, some pointers
        crshseg c15,#0,#$100*4,#$100,c0 ; allocate pointer array for mcs
        storec  c0,XPmcs(c15)
        crshseg c15,#0,#$100*4,#$100,c0 ; allocate pointer array for modcaps
        loadc   c13,0(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; stdin text module
        storec  c1,XMstdin(c0)
        loadc   c13,1(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; stdout text module
        storec  c1,XMstdout(c0)
        loadc   c13,2(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; stderr text module
        storec  c1,XMstderr(c0)
        loadc   c13,3(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; root directory
        storec  c1,XMrootdir(c0)
        loadc   c13,4(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; current directory
        storec  c1,XMcurrdir(c0)
        loadc   c13,5(c14)
        crshseg c15,#$4000,#1,#1,c1
        movcap  (c13),(c1)              ; environment
        storec  c1,XMenv(c0)
        storec  c0,XPmodcap(c15)        ; save in retained heap
        ret
end
;
        .global XIclose
interface_procedure XIclose,2
parameters 0
locals 0
begin
        ret
end
;
        .global XImain
interface_procedure XImain,4
parameters 1
locals 1
begin
XSdatasz        = 8388608
XSstacks        = 8388608
        .global _start
_start:
        setl    #-1                     ; no line numbers yet
        loadcb  c13,#17                 ; get tbase
        loadcb  c15,#16                 ; get rbase
        loadc   c0,XSconst(c13)         ; constant segment - 0
        loadc   c14,XSdata(c13)
        loadc   c1,XPheap(c15)
        loadc   c2,XPstack(c15)
        cmpci   c1                      ; already a valid segment?
        bne     reuseheapstack*4
        crshseg c15,#0,#XSdatasz,#0,c1  ; data (heap) segment - 1
        storec  c1,XPheap(c15)
        crshseg c15,#0,#XSstacks,#0,c2  ; stack segment - 2
        storec  c2,XPstack(c15)
reuseheapstack:
        lda     #data_size_of(XSdata)/4
        bmovw   (c14),(c1)              ; set up initialised variables
;
        ldi     i0,#XSstackoffset       ; initialise stack pointer
        addi    i0,#8                   ; reserve space for empty frame
```

```
        clr     -8(cx)[i0]              ; return address 0
        clr     -4(cx)[i0]              ; dynamic link address 0
        ldax    i0                      ; keep address of the first frame
        addi    i0,#20                  ; reserve space for our frame
                                        ; 2 words linkage, 3 words outgoing
        lda     #__crt_dummy            ; pretend that this is the return addr
        sta     -20(cx)[i0]
        stax    -16(cx)[i0]             ; store dynamic link
;
        loadcb  c14,#0                  ; get parameter list
        loadc   c8,(c14)                ; first parameter: command line string
        ldi     i1,#__end               ; first free memory location
        ldi     i2,#0                   ; current parameter position
        ldax    (c8)                    ; length of command line
        refc    c8,#4,ax                ; data portion of string
        ldi     i3,#0                   ; current position in command line
cmd_wsp:
        cmpi    i3,ax                   ; still within string?
        bhs     cmd_done*4              ; no, finished
        ldba    (c8)[i3]
        addi    i3,#1                   ; skip to next character
        cmpa    #$09                    ; Tab?
        beq     cmd_wsp*4
        cmpa    #$20                    ; Space?
        beq     cmd_wsp*4
cmd_narg:
        sti     i1,__Argv(cx)[i2]       ; start new argument
        addi    i2,#4
        cmpi    i2,#Argvsize-8          ; two entries left?
        blo     cmd_cont*4
        causex  #40960001,#0,#0         ; out of arg space (command line)
cmd_cont:
        cmpa    #$5c                    ; Backslash?
        beq     cmd_esc*4
        cmpa    #$22                    ; Double quotes?
        beq     cmd_quot*4
cmd_copy:
        stba    (cx)[i1]                ; copy argument character
        addi    i1,#1
cmd_nextch:
        cmpi    i3,ax                   ; end of command line?
        bhs     cmd_arg*4
        ldba    (c8)[i3]
        addi    i3,#1
        cmpa    #$09                    ; Tab?
        beq     cmd_arg*4
        cmpa    #$20                    ; Space?
        bne     cmd_cont*4
cmd_arg:
        lda     #0                      ; add end of string
        stba    (cx)[i1]
        addi    i1,#1
        bunc    cmd_wsp*4               ; continue with (possible) new arg
cmd_esc:
        cmpi    i3,ax
        bhs     cmd_endchar*4           ; Backslash at end of command line
        ldba    (c8)[i3]                ; read character after backslash
        addi    i3,#1
        bunc    cmd_copy*4              ; write char, continue copying
cmd_quot:
        cmpi    i3,ax
        bhs     cmd_endchar*4           ; no closing quotes found
        ldba    (c8)[i3]
        addi    i3,#1
        cmpa    #$22                    ; check for end of quoted string
        beq     cmd_nextch*4
        cmpa    #$5c                    ; backslash?
        bne     cmd_quotcopy*4
```

```
        cmpi    i3,ax
        bhs     cmd_endchar*4           ; backslash at end of line, quoted
        ldba    (c8)[i3]
        addi    i3,#1
cmd_quotcopy:
        stba    (cx)[i1]                ; copy character
        addi    i1,#1
        bunc    cmd_quot*4
cmd_endchar:
        causex  #40960003,#0,#0         ; closing quote or char after \ missing
cmd_done:
        clr     __Argv(cx)[i2]          ; mark end of argument list
;
        ldi     i3,i2
        lshfti  i3,#-2                  ; calculate argc
        sti     i3,-4(cx)[i0]           ; pass argc parameter to main
        lda     #__Argv
        sta     -8(cx)[i0]              ; pass argv parameter to main
        addi    i2,#4                   ; advance to environment area
        lda     i2                      ; calculate envp
        adda    #__Argv
        sta     -12(cx)[i0]             ; pass as third parameter
;
        loadc   c11,XPmodcap(c15)       ; segment with pointers
        loadc   c11,XMenv(c11)          ; pointer to modcap
        cmpcapi (c11)                   ; invalid modcap?
        beq     env_none*4              ; no environment modcap, no environment
        ldi     i3,___mcs_free(cx)      ; is there any mcs in the freelist?
        bpl     reusemcs*4
        crshseg c11,#$8000,#1,#1,c10    ; create MCS for environment
        lda     #0
        sta     ___mcs_free(cx)         ; put it into the freelist
        sti     i3,XPmcs(c15)           ; shortcut (avoids putting it back
        inc     ___mcs_alloc(cx)        ; after the close)
        storec  c10,XPmcs(c15)
reusemcs:
        ldi     i3,___mcs_alloc(cx)     ; yes, but take shortcut
        loadc   c10,XPmcs(c15)[i3]
        loadc   c8,XPstrpool(c15)       ; is there a string available?
        cmpci   c8                      ; already a valid segment?
        bne     reusestring*4
        crshseg c11,#0,#68,#2,c8        ; create string, 64 bytes prealloc
        storec  c8,(c8)                 ; save pointer to itself
        storec  c8,XPstrpool(c15)
        storec  c8,XPstrpoollast(c15)
reusestring:
        clr     (c8)                    ; string is empty
        sti     i0,___i0save(cx)                ; save registers that may be clobbered
        sti     i1,___i1save(cx)
        sti     i2,___i2save(cx)
        procall (c10),XPmodcap(c11),c12 ; prepare open
        lda     #0
        pval    #0                      ; first param 0, open read_only
        pval    #0                      ; second param 0, wait forever
        pval    #0                      ; third param 0, open exclusive
        call                            ; open environment
        ldi     i2,___i2save(cx)
        ldi     i1,___i1save(cx)
        ldi     i0,___i0save(cx)
env_read:
        sti     i0,___i0save(cx)        ; save registers that may be clobbered
        sti     i1,___i1save(cx)
        sti     i2,___i2save(cx)
        precall (c10),#5,c12            ; prepare readline
        pref    #0,c12                  ; pass string
        call                            ; read line
        ldi     i2,___i2save(cx)
        ldi     i1,___i1save(cx)
```

```
        ldi     i0,___i0save(cx)
        cmpa    #0                      ; false=end of text
        beq     env_done*4              ; finished with environment
        cmpi    i2,#Argvsize-4          ; one entry left?
        blo     env_new*4                       ; yes, continue with environment
        causex  #40960001,#1,#0         ; out of arg space (environment)
env_new:
        sti     i1,__Argv(cx)[i2]       ; add new environment string
        loadc   c9,(c8)                 ; get string
        ldax    (c9)                    ; get length of string
        refc    c9,#4,ax                ; get data portion of string
        lda     ax
        bmovb   (c9),(cx)[i1]
        addi    i1,ax
        lda     #0
        stba    (cx)[i1]                ; write end of string marker
        addi    i1,#1
        bunc    env_read*4              ; read next line
env_done:
        sti     i0,___i0save(cx)        ; save registers that may be clobbered
        sti     i1,___i1save(cx)
        sti     i2,___i2save(cx)
        precall (c10),#2,c12            ; prepare close
        call                            ; close text
        ldi     i2,___i2save(cx)
        ldi     i1,___i1save(cx)
        ldi     i0,___i0save(cx)
env_none:
        clr     __Argv(cx)[i2]          ; mark end of environment list
        addi    i1,#3                   ; round up to the next word
        andi    i1,#$fffffffc
        sti     i1,___rorig(cx)         ; first available location for heap
        sti     i1,___curbrk(cx)        ; set break right after data
        ldi     i1,#XSdatasz
        sti     i1,___rlimit(cx)        ; set heap limit
        invc    c8                      ; clean up temporary CRs
        invc    c9
        invc    c10
        invc    c11
        invc    c12
        invc    c13
        invc    c14
        invc    c15
;
        jsub    #___libc_init_first     ; initialise the C library
        jsub    #_main                  ; call main(int, char *[], char *[])
        stax    -4(cx)[i0]              ; pass return value as first param
        jsub    #_exit                  ; provided by ANSI C library
        causex  #40960002,#0,#0         ; should never get here
end
;
.section        .data
        .global ___mcs_free
___mcs_free:
        word    -1                      ; index of first mcs in the free list
        .global ___mcs_alloc
___mcs_alloc:
        word    0                       ; index of the next mcs to be allocated
        .global ___modcap_free
___modcap_free:
        word    -1                      ; index of first modcap in freelist
        .global ___modcap_alloc
___modcap_alloc:
        word    6                       ; next modcap to be allocated
;
.section        .udata
        .global ___i0save
___i0save:
```

```
        word
        .global ___i1save
___i1save:
        word
        .global ___i2save
___i2save:
        word
        .global ___i3save
___i3save:
        word
Argvsize        = 400                   ; MUST be the size of __Argv
__Argv: .common 400
;
.section        .udata_start
        blockb  data_size_of(XSdata)
        align
;
.section        .udata_end
        align
        .global __HeapStart
__HeapStart:
        word                            ; first available heap location
        .global __end
__end:
;
```

D.3.1.2 Deklaration von MONADS-Spezifika `mpc/mpc.h`

```
#ifndef _SYS_MPC_H
#define _SYS_MPC_H

#undef CLASS

typedef void *modcap;
typedef void *mcallseg;
typedef void *semaphore;

#define stdin_mc (modcap) 0xe0000000
#define stdout_mc (modcap) 0xe0000001
#define stderr_mc (modcap) 0xe0000002
#define rootdir_mc (modcap) 0xe0000003
#define currdir_mc (modcap) 0xe0000004
#define environ_mc (modcap) 0xe0000005

#define CREATE_METHOD(NUM, NAME, RETVAL, PARAMS...)                    \
  RETVAL NAME(modcap *, modcap , ## PARAMS);
#define OPEN_METHOD(NUM, NAME, RETVAL, PARAMS...)                      \
  RETVAL NAME(mcallseg *, modcap , ## PARAMS);
#define METHOD(NUM, NAME, RETVAL, PARAMS...)                           \
  RETVAL NAME(mcallseg , ## PARAMS);

modcap modcap_alloc(void);
void modcap_free(modcap);
void modcap_copy(modcap, modcap);
void modcap_copy_owner(modcap, modcap);
void modcap_invalidate(modcap);
int modcap_invalid(modcap);

#ifdef _MPC_STUB
extern int __mcs_free;
extern int __mcs_alloc;
extern int __modcap_free;
extern int __modcap_alloc;
#endif

#endif /* _SYS_MPC_H */
```

D.3.1.3 C-Klassendefinition `basic_text.h`

```
#ifndef _basic_text
#define _basic_text

#include <mpc/mpc.h>

#define CLASS basic_text

OPEN_METHOD(1, basic_text_open, void, int, int, int)
METHOD(2, basic_text_close, void)

METHOD(4, basic_text_readch, int, int *)
METHOD(5, basic_text_readline, int, char **)
METHOD(6, basic_text_writech, void, int)
METHOD(7, basic_text_writeline, void, char *)
METHOD(8, basic_text_writechars, void, char *)
METHOD(9, basic_text_erase, void)

#endif
```

D.3.1.4 Programmende `_exit.c`

```
/* Copyright (C) 1991, 1994, 1995, 1996, 1997 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.  */

#include <unistd.h>
#include <stdlib.h>

/* The function '_exit' should take a status argument and simply
   terminate program execution, using the low-order 8 bits of the
   given integer as status.  */
void
_exit (status)
     int status;
{
  status &= 0xff;
  __asm__("ldta %0" : : "m" (status));
  __asm("ret");
}
```

D.3.2 Eingabe- und Ausgabefunktionen

D.3.2.1 Eingabefunktion `read.c`

```
/* Copyright (C) 1991, 1995, 1996, 1997 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.  */

#include <errno.h>
#include <unistd.h>
#include <stddef.h>
#include <mpc/basic_text.h>

extern mcallseg __fdmcs[];

/* Read NBYTES into BUF from FD.  Return the number read or -1.  */
ssize_t
__libc_read (int fd, void *buf, size_t nbytes)
{
  int ch;
  size_t rcnt;
  int end = 0;

  if (nbytes == 0)
    return 0;
  if (fd < 0)
    {
      __set_errno (EBADF);
      return -1;
    }
  if (buf == NULL)
    {
      __set_errno (EINVAL);
      return -1;
    }

  switch (fd) {
  case 0:
    if (__fdmcs[fd] == NULL) {
      basic_text_open(&(__fdmcs[fd]), stdin_mc, 0, 0, 0);
    }
    rcnt = 0;
    while (rcnt < nbytes || end) {
      switch (basic_text_readch(__fdmcs[fd], &ch)) {
      case 0:
        ((unsigned char *)buf)[rcnt++] = (unsigned char) ch;
        break;
      case 1:
        ((unsigned char *)buf)[rcnt++] = '\n';
        break;
      case 2:
        end = 1;
        break;
```

```
      }
    }
    return rcnt;
  default:
    return 0;
  }
  __set_errno (ENOSYS);
  return -1;
}

weak_alias (__libc_read, __read)
weak_alias (__libc_read, read)
```

D.3.2.2 Ausgabefunktion `write.c`

```
/* Copyright (C) 1991, 1995, 1996 Free Software Foundation, Inc.
This file is part of the GNU C Library.

The GNU C Library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

The GNU C Library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Library General Public License for more details.

You should have received a copy of the GNU Library General Public
License along with the GNU C Library; see the file COPYING.LIB.  If
not, write to the Free Software Foundation, Inc., 675 Mass Ave,
Cambridge, MA 02139, USA.  */

#include <errno.h>
#include <unistd.h>
#include <stddef.h>
#include <mpc/basic_text.h>

extern mcallseg __fdmcs[];

/* Write NBYTES of BUF to FD.  Return the number written, or -1.  */
ssize_t
__libc_write (int fd, const void *buf, size_t nbytes)
{
  size_t i;

  if (nbytes == 0)
    return 0;
  if (fd < 0)
    {
      __set_errno (EBADF);
      return -1;
    }
  if (buf == NULL)
    {
      __set_errno (EINVAL);
      return -1;
    }
  switch (fd) {
  case 0:
    return nbytes;
  case 1:
    if (__fdmcs[fd] == NULL) {
      basic_text_open(&(__fdmcs[fd]), stdout_mc, 1, 0, 0);
    }
    for (i = 0; i < nbytes; i++) {
```

```
      if (((unsigned char *) buf)[i] == '\n') {
        basic_text_writeline(__fdmcs[fd], "");
      } else {
        basic_text_writech(__fdmcs[fd], ((unsigned char *) buf)[i]);
      }
    }
    return nbytes;
  case 2:
    if (__fdmcs[fd] == NULL) {
      basic_text_open(&(__fdmcs[fd]), stderr_mc, 1, 0, 0);
    }
    for (i = 0; i < nbytes; i++) {
      if (((unsigned char *) buf)[i] == '\n') {
        basic_text_writeline(__fdmcs[fd], "");
      } else {
        basic_text_writech(__fdmcs[fd], ((unsigned char *) buf)[i]);
      }
    }
    return nbytes;
  default:
    return 0;
  }

  __set_errno (ENOSYS);
  return -1;
}

weak_alias (__libc_write, __write)
weak_alias (__libc_write, write)
```

Literaturverzeichnis

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilerbau*, 1988, Addison-Wesley, ISBN 3-89319-151-8

[2] Jack Davidson und Christopher W. Fraser, *Register Allocation and Exhaustive Peephole Optimization*, Software Practice and Experience 14 (9), September 1984, S. 857-866

[3] Christopher W. Fraser und David R. Hanson, *A Retargetable C Compiler: Design and Implementation*, 1995, Addison-Wesley, ISBN 0-8053-1670-1

[4] Brian W. Kernighan und Dennis M. Ritchie, *The C programming language*, 1978, Prentice-Hall, ISBN 0-13-110163-3

[5] John Rosenberg, *MONADS-PC Instruction Set: MONADS-PC Technical Report 1*, Version 5.3, Februar 1994

[6] John Rosenberg und J. L. Keedy, *Object Management and Addressing in the MONADS Architecture*, Proceedings of the International Workshop on Persistent Object Systems, 1987

[7] John Rosenberg und Jörg Siedenburg, *MONADS-PC Assembler Manual: MONADS-PC Technical Report 3*, Version 3.0, April 1994

[8] John Rosenberg und Jörg Siedenburg, *MONADS-PC System Management Instructions: MONADS-PC Technical Report 5*, Version 3.1, Dezember 1994

[9] Axel Schmolitzky, *MONADS-Pascal Sprachbeschreibung*, interner Bericht Abteilung Rechnerstrukturen, Juni 1995, Fakultät für Informatik, Universität Ulm

[10] Richard M. Stallmann, *Using and Porting GNU CC for Version 2.7.2*, 1996, Free Software Foundation, Boston, Massachusetts, ISBN 1-882114-36-1

[11] Andreas Wickner, *Portierung eines C-Compilers auf eine Rechnerarchitektur mit capability-orientierter Adressierung*, 1991, Diplomarbeit an der Universität Bremen, Studiengang Informatik

[12] Reinhard Wilhelm und Dieter Maurer, *Übersetzerbau: Theorie, Konstruktion, Generierung*, 2. Auflage 1997, Springer Verlag, ISBN 3-540-61692-6

Erklärung

Ich erkläre hiermit, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, 18. Dezember 1997

Klaus Espenlaub
Matr.-Nr. 0259259