Portierung von Unix Version 7-Kommandos auf den RISC-Prozessor ECO32

Norman Ulbrich

1. April 2004

Inhaltsverzeichnis

Kapitel 1

Einführung

Diese Arbeit ist Teil eines größeren Projektes, dessen Ziel es ist, einen virtuellen RISC-Prozessor unter dem Betriebssystem UNIX laufen zu lassen.

Die Wahl fiel aus verschiedenen Gründen auf UNIX in der Version 7:

- Die Quelltexte von UNIX bis Version 7 sind mittlerweile von Caldera veröffentlicht und dürfen frei verwendet, kopiert und geändert werden.
- UNIX ist ein stabiles und effizientes System, das in vielen Bereichen Anwendung findet.[1]
- Der UNIX-Kern ist relativ klein; somit war seine Portierung halbwegs überschaubar.
- UNIX besitzt Multitasking-, Multiprocessing- und Multiuser-Funktionalität.
- Die Betriebssystemarchitektur ist so entworfen, daß die mitgelieferten Programme leicht miteinander kombiniert und zu mächtigeren Programmen zusammengesetzt werden können. – Das ganze ist also größer als die Summe seiner Teile.

Der Prozessor, auf dem UNIX laufen soll, heißt *ECO32*. Entwickelt wurde er von Prof. Dr. Geisse, der auch den Simulator und die virtuelle Platte, das Compilerbackend und weitere in diesem Zusammenhang nötige und nützliche Programme geschrieben hat.

[1] Allerdings handelt es sich dabei um Nachfolgeversionen – Version 7 wird wahrscheinlich gar nicht mehr verwendet.

1.1 Aufbau dieses Berichtes

Ich werde zunächst einige allgemeine Dinge zur Portierung und den verwendeten virtuellen Maschinen beschreiben. Anschließend gehe ich auf altes und neues C ein und erkläre schließlich, wie ich bei der Portierung der Quelltexte vorgegangen bin.

`Datei-` und `Programmnamen` sowie `Parameter von Programmen` erscheinen im Schreibmaschinenstil. Auch sämtlicher Quellcode ist auf diese Art formatiert; wenn es sich um mehr als eine Zeile handelt, ist er zusätzlich abgesetzt.

Kapitel 2

Probleme der Portierung

Beim Wechsel von einem System auf ein anderes möchte man natürlich alle vorhandenen Programme weiter benutzen können. In binärer Form funktioniert das aber nur dann, wenn das alte und das neue System die gleiche Prozessorarchitektur haben und verschiedene Bitmuster auf die gleiche Weise als Befehle interpretieren. Wesentlich häufiger ist also der Fall, daß die Quelltexte der alten Programme speziell für die neue Maschine übersetzt werden müssen.

Auch das ist oft kein Problem, da einfach ein Compiler für den neuen Computer genommen wird[1], mit dem alle Programme neu erstellt werden. In diesem Fall ist es leider nicht so einfach, da der verwendete Compiler `lcc` ANSI-konforme C-Quelltexte erwartet; die vorhandenen Sourcen allerdings in Kernighan und Ritchie-C geschrieben wurden.

Auf die genauen Unterschiede zwischen ANSI-C und *K&R-C* werde ich in Kapitel 7 eingehen. ANSI-C ist jedoch weniger fehleranfällig, weshalb die Verwendung eines „alten“ Compilers wenig sinnvoll erscheint. Vielmehr ist es das Ziel, die vorhandenen Quelltexte zu „ansifizieren“, soll heißen in ANSI-Form zu bringen, um die damit verbundenen Vorteile nutzen zu können.

Dabei können Probleme der verschiedensten Art auftreten. Durch unterschiedliche Größe von Variablen entstehen leicht Probleme, wenn nicht der `sizeof`-Operator verwendet wurde. Manche Konstrukte sind inzwischen veraltet und nicht mehr erlaubt; diese Stellen müssen zum Teil von Hand nachgebessert werden. Externe Namen haben laut ANSI-C-89-Standard eine garantierte Länge von nur sechs Zeichen in Kleinbuchstaben [6]; so etwas könnte leicht zu Linker-Problemen führen, wenn die „abgeschnittenen“ Funktionsnamen nicht mehr eindeutig sind.

[1] Genauer: ein Compiler, dessen Backend Code für den neuen Computer erzeugt.

Kapitel 3

Der ECO32-Prozessor

Der virtuelle Prozessor *ECO32* ist eine Entwicklung von Herrn Prof. Dr. Geisse. Das Projekt war ursprünglich dazu gedacht, als „Vorbereitung“ für eine MMIX-Entwicklung zu dienen. MMIX ist ein von Donald E. Knuth erdachter Prozessor von einiger Komplexität, für den der Erfinder allerdings kein Betriebssystem bereitstellen will. Herr Dr. Geisse hat die Absicht, Version 7 von UNIX auf diesen Prozessor zu portieren, möchte sich aber durch ein Projekt mit dem einfacheren ECO32-Prozessor erst mal vorsichtig an dieses große Projekt herantasten.

ECO32 ist ein RISC-Prozessor mit einfacher aber doch moderner Architektur, wobei das Design eng an den Mips-Prozssor angelehnt ist. Es handelt sich um ein 32-Bit-System mit User- und Kernelmodus und weiteren Features, aber ohne Gleitpunkteinheit. Diese Einschränkung wurde getroffen, weil eine derartige Erweiterung das System um einiges komplexer gemacht hätte. Das war vor allem deswegen unerwünscht, weil geplant ist, den Prozessor einmal in einem FPGA zu realisieren, was die Möglichkeiten naturgemäß einschränkt.

3.1 Architektur

Der ECO32-Prozessor ist ein virtueller 32-Bit-Prozessor mit big endian-Architektur. Er verfügt über 32 Register, wobei die Register $2 bis $29 Allzweckregister sind. Das Register $0 enthält immer 0; $31 enthält die Rücksprungadresse bei Funktionsaufrufen. $30 wird im Falle einer Exception mit dem Instruction Pointer gefüllt. Der Assembler nutzt $1, um die Begrenzung von Konstanten auf eine Länge von 16 Bit vor dem Programmier zu verstecken. Darüber hinaus gibt es vier Spezialregister: ein Flagregister und drei Register zur Kommunikation mit dem Translation Lookaside Buffer.

3.2 Befehlssatz

Der ECO32-Prozessor kennt 61 RISC-Befehle, die alle eine Länge von genau 32 Bit haben.

3.3 Der ECO32-Simulator

Der ECO32-Simulator ist in ANSI-C geschrieben und läuft unter den Betriebssystemen Linux und UNIX. Er enthält umfangreiche „Hardware“-Debuggingmöglichkeiten und verfügt mittlerweile sogar über ein Grafikinterface. Der Simulator ist unter [4] kostenlos verfügbar.

Kapitel 4

UNIX Version 7

Die Version 7 des Betriebssystems UNIX war die erste, die nicht nur auf einem PDP-11-Rechner ausführbar war. Bis dahin hatten sich die Entwickler Brian W. Kernighan und Dennis M. Ritchie in der Implementierung des Kernels auf das Stack-Layout der PDP-11 verlassen, um bestimmte Funktionen zu realisieren. In Version 7 wurde diese Abhängigkeit entfernt, so daß das System nun auf jeder Plattform lauffähig war, für die man C-Code übersetzen konnte.

Seitdem gab es viele Weiterentwicklungen von UNIX, doch einige Fans sind noch heute der Meinung, Version 7 sei „die beste, die es je gegeben hat“.

Seit dem Jahr 2002 ist der Quellcode der 16-Bit-UNIX-Versionen 1 bis 7 für PDP-11-Maschinen frei verfügbar, so daß man nun die nötigen Schritte machen kann, um das System auch auf einer anderen Maschine zum Laufen zu bringen.

Dank der Arbeit von Dennis Kuhn existiert bereits eine portierte Version des UNIX-7-Kerns. [3] Die Quelltexte sind bereits ansifiziert und für den ECO32-Prozessor übersetzt.

Kapitel 5

Der PDP-11-Simulator

PDP-11 war der Name eines weit verbreiteten Rechnertyps mit 16 Bit-Architektur, der von der Firma DEC hergestellt wurde. Das Gerät hatte eine CISC-CPU mit 8 Registern, von denen R0 bis R5 universell nutzbar waren. R6 diente zur Adressierung des Stack, und R7 enthielt den Instruction Pointer.

Der Simulator für die PDP-11-Maschine wurde von Robert M. Supnik entwickelt. [5] Hier ein „Screenshot“ einer bootenden PDP-11:

```
velociraptor:~/unix-v7-2/run$ ./pdp11 run.conf

PDP-11 simulator V2.3d
boot
Boot
: hp(0,0)unix
mem = 177344
# RESTRICTED RIGHTS: USE, DUPLICATION, OR DISCLOSURE
IS SUBJECT TO RESTRICTIONS STATED IN YOUR CONTRACT WITH
WESTERN ELECTRIC COMPANY, INC.
FRI MAY  8 09:40:19 EDT 1970

login: root
Password:
#
```

Dieser Rechnertyp ist schon lange veraltet. Mittlerweile sind 64 Bit-Systeme in aller Munde, und auch CISC hat sich gegen das schnellere RISC nicht durchsetzen können.[1]

[1] Zumindest tendiert z. Zt. wieder alles zu RISC.

Allerdings macht es immer wieder Spaß, sich anzuschauen, unter welchen (für heutige Verhältnisse prähistorischen) Bedingungen die Leute früher gearbeitet haben. Wer durch moderne grafische Oberflächen und IDE's verwöhnt ist, sollte sich ab und zu vor eine PDP-11 setzen und versuchen, mit `ed` ein C-Programm zu schreiben...

Kapitel 6

Die vorhandenen Programme

Im Verhältnis zu den Programmen, die bei einer aktuellen UNIX- oder Linuxdistribution mitgeliefert werden, ist die Menge der zu portierenden Programme relativ gering.[1] Es existieren einige nur bedingt komfortable Befehle, an die sich heutige Anwender erst gewöhnen müssen – erst recht, wenn sie grafische Oberflächen gewohnt sind!

Die Beschränkung auf diese wenigen und einfachen Programme hat allerdings ein paar Vorteile:

- Die alten Quelltexte werden garantiert unterstützt. Neuere Programme nutzen möglicherweise Bibliotheken oder Systemaufrufe, die es damals noch gar nicht gab, und sind somit nicht auf den alten Kernel portierbar.

- Die Zahl der Programme ist begrenzt und genau festgelegt. Das läßt sich deswegen als Vorteil ansehen, weil man sonst fragen müßte, welche der etlichen tausend aktuellen Programme portiert werden sollen und welche nicht.

- Die alten Programme erfordern keine speziellen Compiler oder Compileroptionen, wie es bei moderner Software zum Teil erforderlich ist[2]. Dadurch kann ein recht einfacher Compiler wie der `lcc` verwendet werden.

Felix Grützmacher hat eine ANSI-konforme C-Bibliothek und eine einfache Shell implementiert und damit einen wichtigen Beitrag auch zur Portierung der anderen Programme geleistet.

[1] Hieran kann man anschaulich feststellen, daß das GNU-Projekt erst später entstanden ist.

[2] Das wahrscheinlich beste Beispiel hierfür ist der Linux-Kernel.

Die Liste aller vorhandenen Programme sieht wie folgt aus:

```
1          ac         adb      ar         arcv      as
at         awk        bas      basename   bc        cal
calendar   cat        cb       cc         checkeq   chgrp
chmod      chown      clri     cmp        col       comm
cp         crypt      cu       date       dc        dcheck
dd         deroff     df       diff       diff3     du
dump       dumpdir    echo     ed         egrep     enroll
eqn        expr       f77      factor     false     fgrep
file       find       grep     icheck     join      kill
ld         learn      lex      lint       ln        login
look       lookbib    lorder   ls         m4        mail
make       man        mesg     mkdir      mv        ncheck
neqn       newgrp     nice     nm         nohup     nroff
od         osh        passwd   pcc        plot      pr
prep       primes     ps       pstat      ptx       pwd
quot       ranlib     ratfor   refer      restor    rev
rm         rmdir      roff     sed        sh        size
sleep      sort       sp       spell      strip
struct     stty       su       sum        sync      t300
t300s      t450       tabs     tail       tar       tbl
tc         tee        tek      test       time      tk
touch      tp         tr       troff      true      tsort
tty        uniq       uucp     uulog      uux       vplot
vpr        wc         who      write      xget      xsend
yacc       yes
```

6.1 Nicht portierte Programme

In der vorhanden UNIX-Version gibt es einige Programme, die aber nicht ansifiziert werden brauchen, da sie Gleitpunktzahlen verwenden. Weder das Compilerbackend noch der ECO32-Simulator unterstützen Gleitpunktarithmetik, und die entsprechenden Programme sind auch nicht wirklich wichtig, weswegen man sie vernachlässigen kann. Es handelt sich um folgende Programme:

```
adb     as      awk     bas     c     cpp    f77
graph   iostat  lint    pcc     plot  prof   random
ratfor  sa      spline  struct  tp    units
```

Unter den hier genannten Programmen befinden sich auch Programme, die keine Gleitpunktzahlen verwenden, die aber einfach nicht benötigt werden. Hierunter fallen die C-Compiler `cc` und `pcc`, da sie sowieso nur K & R-C unterstützen; sie zu verwenden, würde einen Rückschritt bedeuten. Damit entfällt auch der Präprozessor `cpp`, da ein solcher im Programmpaket des `lcc` enthalten ist. Ebenso überflüssig sind damit auch `lint`, da auch dessen Fähigkeiten auf altes K & R-C beschränkt sind, und das `mip`-Verzeichnis, das Maschinenunabhängige Teile des `pcc` und von `lint` enthält.

Die Programmiersprache Fortran ist inzwischen weit weniger wichtig als früher. Der entsprechende Compiler `f77` enthält (natürlich) Fließkommabehandlung; `struct` als `indent`-Variante für Fortran wurde aus Zeitgründen nicht portiert (und dürfte – nicht nur in heutiger Zeit, sondern besonders ohne Fortran-Copmiler – wenig nützlich sein). Entsprechendes gilt für den Fortrandialekt Ratfor und den gleichnamigen Compiler.

`tp` wurde aus anderen Gründen nicht portiert: es scheint sich auf eine Eigenart des Compilers zu verlassen, daß keine Füllbytes in Strukturen eingefügt werden. Da so etwas nicht garantiert werden kann (der ANSI-Standard sagt ausdrücklich, daß das passieren darf), bedeutet es eine enorme Arbeitsersparnis, auf `tp` zu verzichten, anstatt es anzupassen. Der Verzicht bedeutet aber keine wirkliche Einschränkung, da `tp` ein Archivierungsprogramm ist, mit `tar` aber ein verbreiteteres Programm dieser Art zur Verfügung steht.[3]

[3]Sogar die Manpage von `tp` sagt im `Bugs`-Abschnitt indirekt, man solle doch lieber `tar` benutzen... [7]

Kapitel 7

Unterschiede zwischen Kernighan & Ritchie-C und ANSI-C

Die Programmiersprache C wurde anfang der 70er Jahre von Brian W. Kernighan und Dennis M. Ritchie entwickelt. C ist die Sprache, in der das UNIX-Betriebssystem geschrieben ist, und daher ist es die Standardprogrammiersprache auf dieser Plattform. Aus heutiger Sicht merkt man jedoch an vielen Stellen, daß die Sprache nicht besonders gut geplant war.

7.1 Nachteile von K & R-C

Funktionen, die nicht deklariert waren, hatten den Rückgabetyp `int` und konnten auf irgendeine Weise aufgerufen werden. Der Compiler hatte keine Möglichkeit, die Typen der aktuellen mit denen der formalen Parameter zu vergleichen. Deklarationen sahen z. B. so aus:

```
double atof();
long atol();
char *strncpy();
int fprintf();
```

In den Funktionsdefinitionen war dann der Typ angegeben:

```
double atof(str);
    char *str;
long atol(str);
    char *str;
```

```
char *strncpy(str1, str2, n);
    char *str1, *str2; /*n ist implizit int*/
int fprintf(f, str, args)
    FILE *f;
    char *str;
        /*args wird von fprintf zum Ermitteln der
        eigentlichen Werte verwendet*/
```

Es gab keine Möglichkeit für den Compiler, die Parametertypen zu überprüfen, und es war kein Problem, einen völlig falschen Typ zu verwenden:

```
strncpy(str1, 5, str2); /*Fehler! Muss
                         strncpy(str1, str2, 5) heissen!*/
```

Obwohl dies wahrscheinlich das größte Problem war, gab es noch viele weitere:

- Da es keinen `const`-Modifizierer gab, waren alle Variablen beschreibbar.
- Alle Funktionen hatten einen Rückgabewert; `void`-Funktionen gab es noch nicht. Wenn es keinen wirklichen Rückgabewert gab, wurde i. d. R. `0` zurückgegeben und der Wert an der Aufrufstelle ignoriert. Allerdings kam es auch vor, daß es kein `return` gab – mit etwas Glück wurde der Rückgabewert einer solchen Funktion nicht verwendet...
- Ebenso fehlte der Pointer auf `void`; der „allgemeine“ Pointer, der auf jedes Objekt zeigen konnte, war `char *`.
- Typ-Promotion fand bei *jedem* Aufruf statt: was kleiner als ein `int` war, wurde zu `int` vergrößert, `float` wurde zu `double`.

Funktionen konnten *irgendwelche* Argumente übergeben bekommen; sowohl eine falsche Anzahl als auch falsche Typen waren möglich, was eine häufige Fehlerquelle war. Der einzige „Vorteil“ dessen war, daß man Parameter, die in einem speziellen Fall nicht benötigt wurden, nicht zu übergeben brauchte (was die Lesbarkeit erhöhen oder verringern konnte – je nachdem ob man gerade den aktuellen Codeverlauf verfolgte und die Zahl der Parameter „stimmte“, oder ob man einen anderen Aufruf der betreffenden Funktion als Beispiel suchte).

7.2 ANSI-C

Im Jahre 1989 wurden viele der Macken von C „ausgebessert“, als der ANSI-Standard *C89* verabschiedet wurde. Es gab zahlreiche Neuerungen, und einige alte Schreibweisen waren nun nicht mehr zulässig. Aus Gründen der Kompatibilität wurde der alte Deklarationsstil aber nicht ungültig.

7.2.1 Funktionsdeklarationen

Die wichtigste Neuerung war die neue Schreibweise für Funktionsdeklarationen: neben dem Funktionsnamen und Rückgabetyp wurden nun auch die Typen der Parameter angegeben, wobei optional auch noch der Name möglich war. Damit sehen die oben angegebenen Deklarationen in C89 wie folgt aus:

```
double atof(const char *);
long atol(const char *);
char *strncpy(char *str1, const char *str2, size_t n);
int fprintf(FILE *stream, const char *format, ...);
```

Dabei ist `...` der *ellipse*-Operator; er zeigt an, daß noch weitere Argumente von beliebigen Typen folgen. Er wird vor allem von der `printf`-Familie angewendet, bei der die angegebene Zeichenkette Aufschluß darüber gibt, um welchen Typ es sich bei den nachfolgenden Variablen tatsächlich handelt. Durch Einbinden des Headers `stdargs.h` können aber auch Anwendungsentwickler diesen Mechanismus nutzen (mit dem man auch in ANSI-C falsche, zu wenige oder „zu viele“ Parameter übergeben kann).

7.2.2 Typmodifizierer

In ANSI-C wurden mit den Typmodifizierer `const`, `volatile` und `signed` neue Schlüsselworte definiert. `const` erklärt ein Datenfeld für readonly: es ist nicht möglich, Variablen eines `const`-Typs etwas zuzuweisen. Das Schlüsselwort `volatile` verbietet dem Compiler, einen Variablenwert zu verwenden, der bereits in ein Register geladen wurde; es *muß* ein Zugriff in den Speicher erfolgen. Dies ist vor allem dann sinnvoll, wenn eine Variable von außerhalb des Programms geändert werden kann (z. B. könnte das Betriebssystem in einer Variablen signalisieren, daß ein bestimmtes Ereignis eingetreten ist). `signed` ist einfach die Umkehrung von `unsigned`, wodurch Variablen jetzt auch explizit vorzeichenbehaftet sein können.

7.2.3 Neue Typen

Neu hinzugekommen sind in ANSI-C die Typen `long double` und `void`. Konsequenterweise geben Speicherverwaltungsfunktionen wie `malloc` nun auch den Typ `void *` zurück, anstatt wie früher ein `char *` (das vielleicht sogar noch in ein `int` verpackt war – siehe dazu [2]).

7.2.4 Di- und Trigraphen

Im Präprozessor von ANSI-C-Compilern sind sogenannte *Di-* und *Trigraphen* definiert. Dabei handelt es sich im Sequenzen aus zwei bzw. drei Zeichen, die für *ein* anderes Zeichen stehen. Es gibt folgende Zuordnung:

Digraph	Trigraph	„eigentliches“ Zeichen
<%	??<	{
%>	??>	}
<:	??(	[
:>	??)	]
	??/	\
	??'	^
	??!	|
	??-	~
%:	??=	#
%:%:		##

Diese Form von Escapesequenz wurde eingeführt, um auch dann gültige Programme schreiben zu können, wenn der eingestellte Zeichensatz die Symbole gar nicht unterstützt. Im Dänischen haben die Vokale Æ, æ, Ø, ø, Å und å Bitmuster, die in (den „meisten“) anderen Ländern z. B. für `[`, `]`, {, } und \ verwendet werden (siehe [8]).

Ein einfacher Test ergab, daß der `lcc` Digraphen überhaupt nicht erkennt. Da alte Compiler wohl erst recht nicht über solch ein Feature verfügt haben, und auch ein `grep` über die Quelldateien keinen Erfolg hatte[1], kann dieses Problem ignoriert werden.

Anders sieht es bei Trigraphen aus: der `lcc` unterstützt diese sehr wohl. Allerdings kommen solche Zeichenfolgen in den Quelltexten überhaupt nicht vor, was man wieder durch ein simples `grep` herausbekommt. Trigraphen brauchen daher nicht behandelt zu werden.

[1] Bei der Suche nach `<%` wird man zwar fündig, aber alle Fundstellen befinden sich innerhalb von Zeichenketten, bei denen Angaben in einer Art EBNF gemacht werden (z. B. `printf("\n<%s>\n\n", file);`).

7.2.5 Der Verkettungsoperator

ANSI-C definiert den Verkettungsoperator `##` zum „aneinanderhängen“ von Token. Sinnvoll ist er hauptsächlich in Makros wie z. B. `TEXT`, das die übergebene Zeichenkette „normal“ oder als Unicode-String übersetzt, je nachdem ob das Präprozessorsybol `UNICODE` definiert ist:

```
#ifdef UNICODE
#   define TEXT(str) L ## str
#else
#   define TEXT(str) str
#endif
```

Beim Präprozessorlauf würde `TEXT("foo")` zur Unicode-Zeichenkette `L"foo"` expandiert, wenn `UNICODE` definiert ist, ansonsten zur „gewöhnlichen“ Zeichenkette `"foo"`.

Der Verkettungsoperator ist dabei notwendig, da `L "foo"` etwas anderes ist als `L"foo"`: das erste sind *zwei* Token (die einen Compilerfehler verursachen würden), das zweite ist ein Unicodestring.

7.3 Weitere Unterschiede

Die alten Compiler waren recht einfach gestrickt und konnten als Parameter oft nur `int`-, `long`-, `double`- und Zeigerwerte verarbeiten und als Funktionswerte zurückgeben. Strukturen und Unionen waren nicht möglich; für solche Konstrukte mußten Zeiger auf die Objekte verwendet werden. Nach dem ANSI-Standard muß ein C-Compiler aber damit klarkommen. Da die Möglichkeiten aber erweitert und nicht eingeschränkt wurden, ist dies bei der Portierung kein Problem – die alte Methode ist ja weiterhin möglich.

In K & R-C ist der „Namensraum“ von Strukturen und Unionen global: man konnte überall jeden einzelnen Feldnamen sehen und seinen Offset berechnen. Dadurch war es möglich, auf Datenfelder zuzugreifen, die gar nicht existieren – der Compiler verwendete einfach den entspechenden Byteoffset für den Zugriff. Das „schöne“[2]

```
union { int _cheat; };
#define Lcheat(a) ((a)._cheat)
```

aus dem Quelltext der Bourneshell nutzte diese Möglichkeit aus, um jedem Datentyp ein `int` zuweisen zu können. Benötigt wurde der `Lcheat`,

[2]Hier läßt sich natürlich darüber streiten, was man unter „schön“ versteht.

weil im Shellquelltext viele Operationen Zeiger und Zahlen wild durcheinandermischen, man mit Zeigern aber keine Bitoperationen durchführen darf. `Lcheat(p)&=1;` ist (bzw. war) jedoch auch für die Zeigervariable `p` erlaubt.

Durch diesen „flachen“ Namensraum war es auch möglich, eindeutig benannte Datenfelder anzusprechen, obwohl nicht der ganze „Pfad“ angegeben war. Z. B. war es bei der Deklaration

```
struct s {
    int i;
    union u {
        int j;
        char *s;
    }
} s_var;
```

möglich, auf `s_var.j` zuzugreifen. Beim verwendeten Compiler `lcc` ist dies nicht erlaubt; der Zugriff muß (korrekt) über `s_var.u.j` erfolgen.

Ein weiterer Unterschied sind die Formatspezifizierer der `printf`-Familie. Durch das ANSI-Kommitee wurde festgelegt, daß einem „langen“ Ganzzahltyp einfach ein `l` vorangestellt wird. Damit sind `%ld` und `%li` gültige Typangaben für Werte vom Typ `long int`.[3] Vor dieser Festlegung war mit `%D` Großschreibung gebräuchlich, was nun zum Fehler führen dürfte. Bei einem `long double`-Wert wird dem `f` ein `L` vorangestellt (was auf ECO32 natürlich nicht vorkommt).

In C ist eine verkürzte Schreibweise möglich, um einen Variableninhalt zu ändern, wobei der vorhandene Wert in die Berechnung einfließt. Die Anweisung

```
var = var + 3;
```

kann man in C schreiben als

```
var += 3;
```

Dabei gibt es nicht nur den Operator `+=`, sondern auch noch `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=` und `^=`. In den Urzeiten von C sah der Operator noch anders aus: `+=` hieß `=+`, um die Zuweisung „schön ordentlich“ auf der linken Seite stehen zu haben. Dies wurde geändert, da im alten Stil das Vorhandensein oder Nichtvorhandensein von Whitespaces das Ergebnis beeinflußt hat: es ist ein Unterschied, ob man schreibt `x=-3;` oder `x= -3`; im ersten Fall wird `x` um drei verringert, im zweiten auf minus drei gesetzt.

[3] Wobei `%i` und `%li` erst seit der Standardisierung gültig sind; vorher war `%d` die einzige erlaubte `int`-Angabe.

Bei Definition einer neuen Variablen war es außerdem erlaubt, den Zuweisungsoperator wegzulassen: `int i 0;` legte eine Variable mit dem Namen `i` an und wies ihr den Wert `0` zu. Für den selben Zweck muß heute `int i = 0;` geschrieben werden.

Vor der Einführung des ANSI-Standards waren Funktionsnamen nicht offiziell reserviert – eine Funktion durfte heißen wie sie will, z. B. auch `malloc` oder `strcpy`. Mittlerweile sind Funktionsnamen aus der Standardbibliothek und sowie einige weitere Bezeichner verboten; Anwendungsprogrammierer dürfen sie nicht verwenden (für eine Liste siehe [6]). Da der Linker sich bei Mehrfachdefinitionen sofort beschwert[4], gibt es sozusagen eine automatische Prüfung für solche Namen (allerdings nur für die Funktionen, die auch wirklich in den Bibliotheken vorkommen). Die Umbenennung erfolgt dann per Hand: an den Funktionsnamen wird einfach ein _ angehängt. (Siehe dazu auch [9].) Es wurden auch Funktionen auf diese Weise umbenannt, die nicht Teil von ANSI-C sind, deren Name aber unter UNIX mit einer bestimmten Funktion assoziiert ist (z. B. `lock`). Außerdem gibt es an vielen Stellen Variablen mit dem gleichen Namen wie Funktionen; auch an diese wurde einfach ein _ angehängt (z. B. `utime`, `tmpnam` und `nlist`).

Sogar das Linkerverhalten war früher anders als heute: wann immer der Linker ein Symbol mehrfach fand, band er es auf dieselbe Adresse. Somit konnten in verschiedenen Dateien Variablen mit demselben Namen *definiert* werden; im späteren Programm gab es diese Variable jedoch nur einmal, und alle Module griffen auf denselben Speicherplatz zu. In ANSI-C ist das anders: hier hat jede Variable ihren eigenen Speicherplatz, und modulübergreifende Verwendung kann nur durch `extern`-Deklaration erfolgen. (Leider gibt der `lcc` an dieser Stelle keine Linker-Warnung aus.)

7.4 Eigenheiten von C

Im folgenden Punkt handelt es sich um keinen Unterschied, aber dieser Punkt trat bei der Portierung auf: es ist implementationsabhängig, ob ein `char` `signed` oder `unsigned` ist. Das hat vor allem Einfluß auf Typumwandlungen: ein `signed char` wird in `signed int` umgewandelt, ein `unsigned char` in ein `unsigned int`. Dies ist wichtig bei Vergleichen und Zuweisungen, in

[4]Das ist hier in der Tat ein Vorteil; zumindest eine Warnung ist extrem sinnvoll. In [6] läßt sich ein Fall nachlesen, in dem ein Programmierer eine Funktion `mktemp` nannte. Die ebenfalls genutze Funktion `getwd` wurde nun vom Linker so gebunden, daß sie nicht mehr die Bibliotheksfunktion `mktemp` aufrief, sondern die vom Programmierer definierte. Letztere erwartete aber *drei* Argumente statt einem, und je nachdem, was sich zu der Zeit für (zufällige) Werte auf dem Stack befanden, meldete das Programm einen Fehler.

denen nicht nur der Typ `char` vorkommt: einem `signed char` mit der Standardgröße ein Byte kann nicht der Wert `128` zugewiesen werden (im heutzutage üblichen Zweierkomplement würde die Variable anschließend den Wert `-128` enthalten).[5] Außerdem hat es Einfluß auf die Offsetberechnung, wenn ein `char` als Arrayindex verwendet wird.

Bisweilen gab der Compiler eine Fehlermeldung darüber aus, daß der Typ eines formalen Parameters nicht mit dem Typ der übergebenen Variable übereinstimmte. Nichteingeweihte finden dies besonders dann seltsam, wenn der Code etwa so aussieht wie in folgendem Beispiel:

```
void foo(const char **p) { }

int main(int argc, char **argv) {
    foo(argv);
    return 0;
}
```

Da denkt man sich schnell: wieso geht das nicht; schließlich läßt sich `char *` doch auch bei allen Stringfunktionen in `const char *` umwandeln. Hier liegt die Sache aber anders: nur dem Typ, auf den *direkt* gezeigt wird, darf ein Modifizierer hinzugefügt werden. Dieser Typ ist `char *`; ihm ein `const` hinzuzufügen, ergäbe `char * const`. Die Compilerwarnung ist also berechtigt – aber sie ist harmlos und kann durch eine einfache Typumwandlung beseitigt werden. (Für eine ausführlichere Erklärung des Sachverhalts siehe [6].)

[5] In diesem Beispiel ist ein Byte 8 Bit groß. Das ist in C zwar keinesfalls festgelegt (der „wahre“ Wert ist in `limits.h` angegeben und heißt `CHAR_BIT`), hat sich im Laufe der Jahre aber fest etabliert.

Kapitel 8

Der Portierungsvorgang

Hier geht es nun wirklich darum, aus den alten K & R-Sourcen ANSI-konforme Quelltexte zu machen. Dazu sind mehrere Schritte sinnvoll: erst werden die Funktionsköpfe behandelt, anschließend wird eine Headerdatei mit den „programminternen“ Deklarationen erstellt, und schließlich werden die für die benutzten Funktionen nötigen Headerdateien eingebunden.

8.1 `ansi_f` von Dennis Kuhn

Mein „Vorgänger“ Dennis Kuhn hat für die Ansifizierung der Kernelquellen ein Programm mit dem Namen `ansi_f` geschrieben. Dieses Programm wurde von mir nur leicht verändert und für einen großen Teil der Ansifizierung verwendet.

8.1.1 Ansifizierung der Funktionsköpfe

`ansi_f` erkennt die Typen der Parameter und aufgrund des `return`-Wertes auch den Rückgabetyp von Funktionen und schreibt einen ANSI-konformen Funktionskopf:

```
<Rückgabetyp> <Funktionsname> (<typisierte Parameterliste>)
```

Die einzige an `ansi_f` vorgenommene Änderung dient zum korrekten Erkennen von Zeigern auf Funktionen.

An einigen wenigen Stellen stürzt das Programm leider noch ab. Da dies mit dem Aufbau der Eingabedaten zusammenhängt, war es am einfachsten, die Funktionen mit den „schlechten“ Eingaben ans Ende der Quelldatei zu verschieben. Dadurch wurde möglichst viel Programmtext verarbeitet; nur

die entsprechende(n) Funktion(en) mußten in diesem Schritt von Hand angepaßt werden.[1]

Diese Lösung ist zwar nicht gerade schön, aber sie ist zweckmäßig und führt zum Ziel – und da der Fehler nur in relativ wenigen Fällen auftritt, ist sie wahrscheinlich auch schneller, als das Programm auf die Fehlerursache hin zu untersuchen oder neu zu schreiben.

Ein Problem mit dem Rückgabetyp von Funktionen hat `ansi_f` allerdings dann, wenn statt des Makros `NULL` der `int`-Wert `0` verwendet wird. In solchen Fällen wird oftmals `int` als Rückgabewert angenommen, ob der „eigentliche“ Typ z. B. `char *` ist. Solche Fehler werden vom Compiler bemerkt und müssen anschließend von Hand korrigiert werden.

8.1.2 Erstellen einer Header-Datei

Wenn `ansi_f` mit der Option `--header` aufgerufen wird, gibt es die Deklarationen der im Quelltext definierten Funktionen aus. Diese Ausgabe kann man einfach in eine Datei umlenken, wodurch eine Headerdatei für den entsprechenden Quelltext entsteht. Jetzt fehlen noch die Deklarationen für Bibliotheksfunktionen, welche auf andere Weise erstellt und eingefügt werden.

Tatsächlich ist das Erstellen von Headerdateien aber anders realisiert, siehe dazu 8.4.7 und 8.4.8.

8.2 Einfügen von `#include`-Anweisungen

Natürlich kann man leicht alle programminternen Funktionsdeklarationen an den Anfang des Quelltextes schreiben; sehr übersichtlich ist das aber nicht. Außerdem verwendet jedes Programm auch Funktionen, die es nicht selber definiert: Funktionen aus der Standardbibliothek.

[1] Häufig trat z. B. der Fall auf, daß die geschweifte Klammer direkt nach dem Funktionskopf folgte (und dazwischen womöglich noch Parametertypen aufgeführt waren). Damit kommt das Programm nicht zurecht; es erwartet

```
<optionaler Typ> <Name> (<optionale Parameterliste>)
    <optionale Parametertypen>
{
    <Funktionsrumpf>
}
```

Aus unerfindlichen Gründen gerät der Prozeß bei manchen anderen Sachen in eine Endlosschleife. Das ist insofern besonders unschön, weil man ihn daraufhin `kill`en muß. Allerdings war das Programm ja eigentlich auch nur für die Kernelquellen gedacht und brauchte gar nicht allgemein zu funktionieren.

Für die Deklaration dieser Funktionen müssen sowieso die entsprechenden Headerdateien per `#include` eingebunden werden. Es ist also naheliegend, erst die Standardheader und anschließend den Programmheader zu „includieren“.

8.2.1 Finden der benötigten Header der Standardbibliothek

Da die Standardbibliothek schon immer recht umfangreich war, hat man zusammengehörige Deklarationen in gemeinsame Dateien verpackt. Das Problem ist nun, herauszufinden, welche der zahlreichen Headerdateien eigentlich verwendet werden sollen. Einfach alle zu benutzen ist zwar theoretisch denkbar aber ziemlich schlechter Stil. Darüber hinaus müßte man die Namen aller Dateien herausfinden, wobei sich da u. U. durchaus etwas ändern kann (wenn z. B. ein anderer Compiler benutzt wird). Dazu kommt noch, daß das verwenden jedes einzelnen Headers das Übersetzen der Quelltexte unnötig verlangsamen und weitere Wartung erschweren würde.

Um die *benötigten* Headerdateien zu ermitteln, muß man erst mal wissen, welche Funktionen ein Programm überhaupt benutzt, und welche davon nicht deklariert sind. Hierzu kann man leicht den GNU-Compiler `gcc` verwenden, der nicht deklarierte Funktionen als Fehler meldet und dabei die Funktionsnamen anzeigt. Diese werden mittels `awk` herausgefiltert (siehe dazu auch 8.4.6 und [3]).

Anschließend wird das Programm `grep` mit der Option `-l` aufgerufen, das in den vorhandenen Headerdateien nach der Funktionsdeklaration sucht. Als Ergebnis erhält man alle Dateien, die die Funktionsdeklaration erhalten. Mit dem Stream-Editor `sed` werden die Dateinamen nun in `#include <>`-Anweisungen eingebettet und in eine separate Headerdatei geschrieben (siehe 8.4.7), die vom eigentlichen Programm anschließend per `#include` verwendet wird.

8.2.2 Mehrfachdefinitionen

Nur wenige der Headerdateien enthielten die ratsame (in diesen Fällen sogar nötige) Konstruktion von `#ifndef #define #endif` – bei mehrfachem `#include` wurden die Anweisungen also auch mehrfach vorgenommen. Das spielt bei Funktionsdeklarationen keine Rolle (solange sich bei gleichem Funktionsnamen Rückgabe- und Parametertypen nicht unterscheiden!). Wiederholte `#define`s auf dasselbe Symbol sind genau dann erlaubt, wenn der für das Symbol einzusetzende Text derselbe ist, was bei derselben Datei ja der

Fall ist. Probleme treten aber bei `struct`s und `typedef`s auf: es gibt Compilerfehler, wenn Strukturen oder Typen mehrfach definiert werden (selbst wenn der Aufbau des Typs jedesmal gleich ist).

Das Problem kann ganz einfach gelöst werden, indem eine Präprozessorabfrage die Mehrfachincludierung verhindert (hier bei der Datei `ar.h`):

```
/*Deklarationen und Definitionen nur einmal verwenden*/
#ifndef _AR_H_
#define _AR_H_

#define    ARMAG    0177545
struct    ar_hdr {
    char    ar_name[14];
    long    ar_date;
    char    ar_uid;
    char    ar_gid;
    int     ar_mode;
    long    ar_size;
};

#endif /*_AR_H_*/
```

Nach dem ersten Einschließen der Datei ist das Makro `_AR_H_` definiert. Wenn nun ein weiteres `#include` auf die Datei erfolgt, so wird der Abschnitt zwischen `#ifndef _AR_H_` und `#endif` ignoriert, *weil* `_AR_H_` eben schon definiert ist.

Diese Verfahrensweise wurde soweit nötig bei den Headerdateien meiner Kollegen (von Hand) und ausnahmslos bei den neu generierten Headerdateien für die Programme (automatisch) angewendet.

8.3 Verzeichnisstruktur

Da die Quelltexte in mehreren Stufen ansifiziert werden, liegen sie in verschiedenen Verzeichnissen. Dadurch kann man relativ leicht den Effekt automatischer Änderungen nachvollziehen und bei Fehlern oder ungewünschten Effekten die Änderungsprogramme nachbessern. Die Verzeichnisse sind:

`old/` Hier befinden sich die Quelltexte der Originalprogramme. Diese Dateien werden nicht verändert, sie dienen nur als Ausgangspunkt für die Portierung.

`rightops/` `mkrightops` speichert hier die Quelltexte mit den modifizierten Operatoren (siehe 7.3 und 8.4.1), die anschließend von `ansi_f` bearbeitet werden (siehe 8.1).

`miss/` In diesem Verzeichnis fand die meiste Arbeit statt. Es galt nun, Sprachkonstrukte zu finden, die der ANSI-Standard nicht erlaubt, die aber von `ansi_f` nicht geändert werden (können). Dieser Schritt ist nicht nur der aufwendigste, sondern er ist auch nicht automatisch durchfürbar. Anschließend werden die `.miss`-Dateien erzeugt, Headerdateien gesucht bzw. erstellt (siehe 8.4.7 und 8.4.8) und die fertigen Sourcen im `cmd`-Verzeichnis abgelegt.

`cmd/` Hier liegen am Ende die ansifizierten Quelltextdateien und nach der Übersetzung auch die Binärdateien für ECO32.

8.4 Die Aufrufe der Ansifizierungsprogramme im Detail

Da der Automatisierungsgrad der Ansifizierung möglichst hoch sein soll, gibt es ein paar Shellskripts, die einfache, immer wiederkehrende Arbeit erledigen. Sie heißen (in der Reihenfolge der Aufrufe):

- `mkrightops.sh`
- `ansify.sh`
- `ansi_f.sh`
- `mkmainint.sh`
- `mklongmod.sh`
- `all.sh`
- `mkmiss.sh`
- `mksheader.sh`
- `mkcheader.sh`
- `ins_inc.sh`

- `findsrcs.sh`
- `compile.sh`

Die meisten dieser Shellprogramme sind nur für die Aufrufe der gleichnamigen Binärprogramme da (bis auf das `.sh` am Ende), wobei meist Ein- und Ausgabe umgelenkt werden. Bei dieser Art von Delegation wird das Programm erklärt, sonst das Shellskript selber.

Außerdem gibt es noch einige kleinere Hilfsprogramme die von den Shellskripten verwendet werden:

`stripprefix`	löscht aus Parameter eins die ersten Zeichenkette, die in Parameter zwei vorkommt. Verschiedene Zeichenketten im zweiten Parameter werden durch `;` getrennt.
`stripsuffix`	funktioniert fast wie `basename` mit zwei Parametern – allerdings wird *nur* das Suffix des ersten Parameters entfernt; der Verzeichnisname bleibt erhalten.
`stripuntil`	löscht in der angegebenen Zeichenkette alles bis vor das erste Vorkommen des zweiten Parameters.

8.4.1 `mkrightops`

Das Programm `mkrightops` wandelt Zuweisungen im alten Stil (s. 7.3 auf Seite 20) in den neuen Stil um. Es besteht nur aus der `flex`-Spezifikation eines Scanners mit entsprechend veränderter Tokenausgabe und macht sich das Standardverhalten eines solchen Scanners zunutze: Kopieren der Standardein- auf die Standardausgabe.

```
%{
#include <stdio.h>
%}

%%

/*Alles innerhalb von Zeichenketten ignorieren*/
\"([^"]|\\\")*\"    printf("%s", yytext);

/*Ignorieren von '"'*/
'\"'                printf("%s", yytext);

/*Vorsicht bei Kommentaren mit =*/
"/*"                printf("%s", yytext);
```

```
"*/"                printf("%s", yytext);
"=*/"               printf("%s", yytext);

/*HIER soll transformiert werden*/
=[+\-*/%~^&|]       printf("%c%c", yytext[1], yytext[0]);
=("<<"|">>")        printf("%c%c%c",
                        yytext[1],
                        yytext[2],
                        yytext[0]);

/*Abfangen von Pointerzugriffen wie "if (x<=*ptr)"*/
"=="                |
"<="                |
">="                |
"="                 printf("%s", yytext);
```

Es finden nun immer noch einige falsche Transformationen statt, z. B. bei `if (x=*p)`. Dies hätte theoretisch mit einem Leerzeichen vor dem `*` verhindert werden können – ein Lauf einer alten K & R-`indent`-Version wäre an dieser Stelle wohl nicht schlecht gewesen. Glücklicherweise findet aber der Compiler solche Fehler: wenn `*p` gültig ist, muß `p` ein Zeiger sein; die Multiplikation mit einem Zeigerwert ist aber verboten.

Da so etwas nicht so oft vorkommt, kann man Fehler dieser Art anschließend von Hand beseitigen und dabei gleich ein Leerzeichen einfügen, damit der Fehler an dieser Stelle nicht wieder passiert.

8.4.2 `ansify.sh` und `ansi_f.sh`

`ansify.sh` läßt `ansi_f.sh` über alle `.c`-Dateien laufen. `ansi_f.sh` wiederum delegiert die eigentliche Arbeit an Dennis Kuhn's Program `ansi_f`. Letzteres ist jedoch auf den Stil in den Kernelquellen spezialisiert; unter manchen Umständen stürzt es ab (siehe auch 8.1). Um aber nicht die komplette Behandlung aller Dateien überwachen zu müssen (siehe auch die Fußnote auf Seite 24), startet `ansi_f.sh` das Programm im Hintergrund und wartet (bis zu) fünf Sekunden auf seine Beendigung; sollte es dann noch laufen, wird es abgeschossen.[2]

[2] In Tests mit kürzeren und längeren Wartezeiten ergab sich, daß die korrekten Läufe innerhalb dieser Zeit fertig wurden; die übrigen befanden sich bereits in einer Endlosschleife.

8.4.3 mklongmod

Eine größere Herausforderung waren bereits die `printf`- und `scanf`-Formatspezifizierer. In Prä-ANSI-C-Versionen wurden „lange“ Datentypen wie `long int` (andere `long's` gab es damals auch noch gar nicht) bei den Funktionen `printf` und `scanf` durch Großbuchstaben ausgegeben:

```
long int li = 17;
printf("%D\n", li);
```

Seit ANSI-C gibt es dafür aber den „Modifier“ `l`, der vor den (gewöhnlichen) Kleinbuchstabe gesetzt wird:

```
long int li = 17;
printf("%ld\n", li);
```

Dieser Fall schien komplizierter als der vorhergehende. Die möglichen Formate schließen sich nicht alle gegenseitig aus: es kann durchaus ein `long int` geben, dem Nullen vorangestellt werden sollen.

Nach einem Blick in die V7-Manpage [7] von `printf` zeigte sich aber ein „geordneter“ Aufbau der Formatangaben: es gibt immer nur *eine* mögliche Reihenfolge, so daß auch hier ein `flex`-Scanner für die Transformation ausreicht:

```
%{
#include <stdio.h>
#include <ctype.h>
%}

DQUOTE      \"
MOD         -?0?([0-9]+|\*)?(.([0-9]+|\*)?)?l?
TYPESPEC    [doxcsu%DOXU]
FORMAT      %{MOD}{TYPESPEC}
/*Siehe dazu die V7-Manpage von printf*/

%%

/*Ignorieren von '"'*/
'{DQUOTE}'                          printf("%s", yytext);

{DQUOTE}([^%"]|\\\"|{FORMAT})*{DQUOTE}    {
    int i = 0;
    while (i < yyleng) {
```

```
        if (yytext[i] == '%') {
            printf("%%");
            i++;
            switch (yytext[i]) {
                case '%':
                    printf("%%");
                    i++;
                    break;
                default:
                    while (isdigit(yytext[i]) ||
                        yytext[i] == '-' ||
                        yytext[i] == '*' ||
                        yytext[i] == '.' ) {
                        printf("%c", yytext[i]);
                        i++;
                    }
                    if (isupper(yytext[i])) {
                        printf("l%c", tolower(yytext[i]));
                    } else {
                        printf("%c", yytext[i]);
                    }
                    i++;
            }
            continue;
        } else {
            printf("%c", yytext[i]);
            i++;
            continue;
        }
    }
}
```

8.4.4 `all.sh`

`all.sh` ruft `find` auf und läßt über alle `.c`-Dateien weitere Shellskripts laufen:

- mkmainint.sh
- mkmiss.sh
- mksheader.sh

- mkcheader.sh
- ins_inc.sh

8.4.5 mkmainint.sh

Das Shellskript `mkmainint.sh`[3] stellt sicher, daß die Funktion `main` den Typ `int` zurückgibt.

In vielen Fällen enthält die `main`-Funktion kein `return`-Statement; oft kommt statt dessen ein `exit`-Aufruf oder gar kein „richtiges“ Funktionsende vor. Das Programm `ansi_f` kann daher keinen Rückgabetyp sehen und verpaßt `main` deswegen den Typ `void`. Dies widerspricht aber dem ANSI-Standard, nach dem es für `main` drei mögliche Signaturen gibt:

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *env[]);
```

`mkmainint.sh` läßt nun einfach `sed` über die Quelltexte laufen und jedes Vorkommen von `void main` durch `int main` ersetzen. Falls die Funktion am Ende kein `return` enthält, gibt es später einen Compilerfehler; dieser ist aber schnell zu beheben – man hat also nicht mehr Arbeit, als wenn man alle `main`s nach dem Rückgabetyp `void` durchsuchen würde.

8.4.6 mkmiss.sh

Wie auch schon Dennis Kuhn in [3] geschrieben hat, eignen sich der GNU-Compiler `gcc`, `grep` und `awk` sehr gut, um die Namen von nicht deklarierten Funktionen zu ermitteln: der Compiler gibt eine Warnung samt dem Namen aus, und man muß ihn sich nur heraussuchen. Prinzipiell funktioniert das genauso wie bei Kuhn:

```
gcc -Wall -c $1 2>&1\
| grep -i "implicit declaration of function"\
| awk '{print $7}'\
| sort | uniq | ./filter.e \'\` > $MISSFILE
```

Tatsächlich bekommt der Compiler aber noch weitere Optionen mitgegeben: `-nostdinc` sorgt dafür, daß nicht die Standardverzeichnisse für `#include`-Dateien verwendet werden (etwa `/usr/include/`); `-I`-Optionen geben statt

[3] `make main int`

dessen verschiedene Pfade zu den Headerdateien für ECO32 bzw. UNIX-V7 an.

Die Namen der nicht deklarierten Funktionen stehen nun in einer Datei mit der Erweiterung `.miss` statt `.c` und werden verwendet, um maschinell die nötigen Headerdateien zu suchen und einzubinden.

8.4.7 `mksheader.sh`

Nachdem mit `mkmiss.sh`[4] die fehlenden Deklarationen ermittelt wurden, sucht `mksheader.sh`[5] in den `#include`-Verzeichnissen nach Headerdateien, die die fehlende Deklaration enthalten. Diese werden in eine neue Headerdatei geschrieben, an deren Namen statt `.c` die Erweiterung `_h.h` angehängt wird.[6]

Bei der Suche werden natürlich nur Deklarationen von Funktionen gefunden, die vom System oder Bibliotheken zur Verfügung gestellt werden. Funktionen, die in anderen Programmodulen (oder im selben Modul, aber später als der Aufruf) definiert sind, werden hierbei (noch) nicht gefunden.

Die Suche findet wie folgt statt: zuerst wird eine `#ifndef #define`-Anweisung mit dem Dateinamen in die neue Headerdatei eingefügt; `name.c` wird dabei zu `__NAME_H__`. Anschließend werden per `grep` alle Präprozessoranweisungen in der `.c`-Datei eingefügt. Dies ist nötig, weil eine Funktionsdeklaration einen Zeiger auf eine Struktur verwenden kann, (theoretische) Beispiele wäre `int free_inode(struct inode *in);` oder gar `FILE *open_logfile(void);`. Durch die Präprozessoranweisungen werden aber alle `#include`s schon vor den (noch zu erzeugenden) Funktionsdeklarationen eingefügt.

Sich nicht auf `#include`-Anweisungen zu beschränken ist nicht nur einfacher, sondern hat auch noch den Vorteil, daß bedingte Übersetzung nicht übergangen wird: möglicherweise werden im Debug-Modus andere Headerdateien verwendet. Da der Ersetzungstext nicht geändert wird, können auch die `#define`-Zeilen einfach dupliziert werden. Probleme machen dabei allerdings Makros, die über mehr als eine Zeile gehen; hier würde nur die erste Zeile kopiert und der Rest übergangen werden. Aus diesem Grund sind mehrzeilige Makrodefinitionen in `_struct.h`-Dateien (siehe 8.4.10) oder „andere“

[4] `make missfile` oder präziser `make file with missing declarations`

[5] `make system header` – system- und bibliotheksabhängige Headerdateien nach Deklarationen durchsuchen

[6] Das `_h` wurde eingefügt, weil viele Programme eigene Headerdateien mit der Erweiterung `.h` besitzen, es aber keine Dateien mit Unterstrich gibt (wenn man von einer `lint`-READ_ME mal absieht). Es geht hierbei also nur um das Verhindern von Namenskollisionen.

Headerdateien ausgelagert.[7]

Anschließend werden in verschiedenen Verzeichnissen[8] Headerdateien mit `grep` nach den fehlenden Funktionsdeklarationen durchsucht und der Verzeichnisname mit `stripprefix` entfernt (siehe 8.4). Für die gefundenen Headerdateien wird anschließend ein `#include <>` in die `_h.h`-Datei eingefügt.

8.4.8 `mkcheader.sh`

Nach den Präprozessor- und weiteren `#include`-Anweisungen werden durch `mkcheader.sh`[9] nun evtl. vorhandene `_struct.h`-Dateien in die neue Headerdatei eingefügt. Schließlich basteln `grep` und `awk` Deklarationen der im entsprechenden Quelltext vorhandenen Funktionen und schreiben sie ebenfalls (samt bedingter Überstzung) in die Headerdatei, damit auch dort alle Funktionen deklariert sind (falls eine Funktion eine andere aufruft, die erst später definiert ist). Dies ist problemlos auch dann möglich, wenn Zeiger auf Strukturen über- oder zurückgegeben werden, denn die Strukturen sind ja mittlerweile definiert.

Zuletzt wird ein `#endif` an die Datei angehängt, um die aus `mksheader.sh` noch offene `#ifndef`-Anweisung zu schließen.

8.4.9 `ins_inc.sh`

`ins_inc.sh`[10] schreibt lediglich an den Anfang der `.c`-Datei ein `#include` für die (neue) `_h.h`-Datei.

8.4.10 `_struct.h`-Dateien

In 8.4.7 wurde angesprochen, daß es Probleme bei Strukturdefinitionen geben kann: *Re*dedefinition von Strukturen führt zu Fehlern (siehe auch 8.2.2). Man kann also nicht automatisch auch Strukturen aus der `.c`-Datei in die `_h.h`-Datei kopieren, da diese dann zweimal vorhanden wären. Hier währe bedingte Übersetzung denkbar: die Strukturdefinition wird von

```
#ifndef Strukturname_DEFINED
#define Strukturname_DEFINED
```

[7] Eine alternative hierzu wäre die Verwendung von `awk` gewesen, um auch die folgenden Zeilen zu kopieren.

[8] Dabei handelt es sich um „meine“ Verzeichnisse mit z. T. angepaßten Headerdateien (geänderte Typen o. ä.), die auf den Headerdateien von Felix Grützmacher basieren, und Bibliotheksergänzungen (siehe Kapitel 9).

[9] `make complete header`

[10] `insert include's`

und

```
#endif
```

eingeschlossen. Auf Dauer würde diese Vorgehensweise aber den Namensraum mit `#define`s „zumüllen“. Außerdem läßt sich streng genommen nicht ausschließen, daß es die verwendeten Makros schon gibt; vielleicht hat der Programmierer sie aus irgendwelchen Gründen bereits selber verwendet.

Aus diesem Grund habe ich eine andere Verfahrensweise gewählt: Strukturdefinitionen eines Programms erhalten ihre eigene Headerdatei mit dem Namensanhang `_struct.h`. Wenn für ein Programm eine solche Datei existiert, wird zu Beginn der `_h.h`-Datei ein `#include` für die `_struct.h`-Datei eingefügt. Dadurch kommt im entsprechenden Programm die Strukturdefinition genau einmal vor.

8.4.11 Automatische Übersetzung – `findsrcs.sh` und `compile.sh`

Um nicht für jeden Quelltext einzeln den Compiler aufrufen zu müssen, gibt es das Shellskript `findsrcs.sh`, das für jeden Quelltext `compile.sh` aufruft; dieses wiederum startet `make` – ohne Optionen, wenn im entsprechenden Verzeichnis ein `Makefile` existiert, mit den Optionen `CC=...`, `CFLAGS=...` und dem Namen der zu erzeugenden ausführbaren Datei, falls es keines gibt.[11]

Der einzig wirkliche Nachteil dabei ist, daß `make clean` nicht funktioniert. Auch das Löschen von Dateien aus dem `cmd`-Verzeichnis „ist nicht ganz ohne“, da nur die `.c`- und `_h.h` automatisch erzeugt werden; benötigte Headerdateien oder `Makefile`s müßten von Hand wieder aus dem `miss`-Verzeichnis kopiert werden.

8.5 Handarbeit – Ansifizierung durch Compilermeldungen

Der größte Arbeitsaufwand fand zwischen der Anpassung der Formatspezifizierer (8.4.3) und der Generierung der `.miss`-Dateien (8.4.6) statt. Das automatische Suchen und Generieren von Headerdateien, das Erzeugen von angepaßten Quelltexten (die `#include`-Zeile für die neue Headerdatei) und das anschließende Übersetzen wurden etliche Male durchgeführt, um durch Fehlermeldungen die Punkte zu finden, an denen von Hand noch Änderungen

[11] Man hätte das ganze natürlich auch direkt mit `make` realisieren können, aber das Schreiben eines solchen `Makefile`s hat mich abgeschreckt.

duchrgeführt werden mußten. Das waren in den meisten Fällen nicht korrekt transformierte Funktionsdeklarationen, Übergabe von Parametern ohne Typumwandlung und ab und zu auch Zugriffe auf nicht voll angegebene Unionen (siehe 7.3).

Sehr häufig gab es auch noch vorhandene Funktionsdeklarationen, bei denen (wie es früher üblich war) nur Name und Rückgabetyp angegeben waren. Da der Compiler an solchen Stellen vor unvollständigen Deklarationen warnt, war es am einfachsten, diese Zeilen gleich ganz zu löschen – schließlich stehen modul*interne* Deklarationen in der neuen Headerdatei, und die Deklarationen von Bibliotheksfunktionen wurden automatisch gesucht und eingebunden.

Ein wenig anders hingegen wurden Programme behandelt, die aus mehreren Quelltexten bestehen. Hier gibt es oft Aufrufe, die über Modulgrenzen hinweggehen. Somit waren die gerufenen Funktionen nicht deklariert, da jeder Quelltext seine eigene `_h.h`-Datei erhält. In solchen Fällen wurde individuell entschieden, ob ein `#include` für eine „fremde“ `_h.h`-Datei eingefügt wurde, oder ob eine „spezielle“ Headerdatei besser ist – neu oder schon vorhanden und geändert –, die von mehreren Modulen verwendet werden kann und die nötigen Deklarationen enthält.

8.6 `sh` – Die Bourneshell

Als besonders Umfangreich hat sich die Portierung der Bourneshell herausgestellt, weswegen sie hier einen eigenen Abschnitt erhält. Die Bourneshell `sh` war eine der ersten Shells und hat dadurch weite Verbreitung gefunden. Sie hatte (für damalige Verhältnisse) komfortable Möglichkeiten, die auch gut zur Programmierung von Shellskripts verwendet werden konnten.

Die Bourneshell ist – je nach Sichtweise – nicht in C geschrieben, sondern in Präprozessor. Für alle C-Konstrukte gibt es Makros, mit denen der Quellcode so ähnlich aussieht wie Shellprogramme. Das mag allenfalls für den Autor Steve Bourne von Vorteil gewesen sein, der damit einen Algol-68-Dialekt nachahmte. [6]

Auch die verwendeten Typen sind allesamt über `typedef`s festgelegt. Das mag die Maschinenunabhängigkeit fördern, für die *automatische* Portierung ist dieser Aufbau jedoch tödlich. Schwierig für jede Art von Portierung ist auch das alte (bzw. beinahe nicht vorhandene) Typsystem, das die Konvertierung zwischen verschiedenen Zeigern untereinander und von und zu Ganzzahlen problemlos gestattet – obwohl danach nur noch der Compiler weiß, was der Quelltext eigentlich aussagt.

Die größte Verwirrung beim Portieren entstand aber dadurch, daß das

Programm keine `malloc`-Aufrufe durchführt, sondern mittels `sbrk` eine eigene Heap- und Stackverwaltung realisiert. Diese Technik hebelt selbst die Möglichkeit aus, „mal eben auf `malloc` umzustellen“, da sich die Funktionen gegenseitig ausschließen[12]. Die Shell verwendet aber bei vielen kleineren Berechnungen einen internen Stack, und auch in den entsprechenden Hilfsroutinen wird der Systemaufruf `sbrk` verwendet.

Beinahe das einzige, was mit der verwendeten automatischen Portierungsmethode für die Bourneshell fehlerfrei läuft, ist die Behandlung von Funktionen ohne Parameter: es wird korrekt `void` zwischen den Klammern eingefügt. Die übrigen Portierungsaufgaben müssen von Hand vorgenommen werden, wobei lediglich Compilermeldungen und `grep` helfen.

Sogar für die Ausgabe der Shell existieren eigene Funktionen: es wird kein `printf` verwendet; Zeichenketten werden mit `write` ausgegeben, Zahlen werden vorher „von Hand“ in Zeichenketten umgewandelt. Die Shell verwendet scheinbar keine einzige Standard-Funktion, sondern nur Systemaufrufe – alles andere macht sie selbst...

Auch sind viele der Funktionen nicht gerade einleuchtend. Was macht die folgende Funktion?

```
VOID Ldup(REG INT fa, REG INT fb)
{
    dup(fa|DUPFLG, fb);
    close(fa);
    ioctl(fb, FIOCLEX, 0);
}
```

Auch nach („modernisierter“) Expansion der Bournemakros kann man noch eine Weile rätseln:

```
void Ldup(register int fa, register int fb)
{
    dup(fa|DUPFLG, fb);
    close(fa);
    ioctl(fb, FIOCLEX, 0);
}
```

Der Witz an der Sache: die Funktion ist (für moderne Compiler; semantisch) gar nicht richtig. `dup` erwartet ja nur *einen* Parameter und liefert einen

[12] `man sbrk` besagt folgendes: `The behavior of brk() and sbrk() is unspecified if an application also uses any other memory functions (such as malloc(3C), mmap(2), free(3C)).`

neuen Dateideskriptor *als Rückgabewert*. Des weiteren muß ein gültiger Deskriptor hineingereicht werden; das `| DUPFLG` verändert aber den Deskriptor – es setzt ein weiteres Bit, wodurch die entstehende Zahl kein gültiger Deskriptor mehr ist. Da der Rückgabewert ignoriert wird, fallen die Fehler aber gar nicht auf...[13]

Der modifizierte Code lautet nun:

```
void Ldup(register int fa, register int fb)
{
    dup2(fa, fb);
    close(fa);
    ioctl(fb, FIOCLEX, 0);
}
```

Die Funktion biegt die Ein- oder Ausgabe von Deskriptor `fa` auf `fb` um und schließt `fa`. Bei einem `exec`-Aufruf wird auch `fb` automatisch geschlossen.

8.6.1 Signalbehandlung

Da die Bourneshell ihre eigene Speichverwaltung implementiert, fängt sie das Signal `SIGSEGV` und geht bei Speicherzugriffsverletzungen davon aus, daß nicht genügend freier Speicher verfügbar ist. Deswegen wird nun `sbrk` aufgerufen, um das Betriebssystem nach weiterem Speicher zu fragen. Das Problem dabei ist, daß ein Debugger bei Empfangen dieses Signals nicht die Signalbehandlung der Shell aufrufen läßt, sondern den Shellprozeß beendet. Somit verhindert diese Speicherverwaltung sogar Debugging.[14]

8.6.2 Umgebungsvariablen

„Der Begriff *Umgebungsvariable* ist ein Begriff aus dem Bereich der Betriebssysteme von Computern. Eine Umgebungsvariable enthält beliebige Zeichenketten, die in den meisten Fällen Pfade zu bestimmten Programmen oder Daten enthalten, sowie bestimmte Daten, die von mehreren Programmen verwendet werden können.“ – Soweit eine Definition.

[13] Ich wüßte aber wirklich gerne, wieso das vor einigen Jahrzehnten geklappt hat... – Wurde Bit 6 etwa auf damaligen Systemen ignoriert, weil ein Programm sowieso nicht so viele Deskriptoren erzeugen konnte?

[14] Allerdings hatte der damalige C-Compiler noch nicht einmal eine Option, um Debuginformationen in ein Programm mit einzubinden[15]. Die „Debug-Einschränkung“ galt damals also für alle Programme und tritt erst bei neueren Compilern zu Tage.

[15] Zumindest habe ich keine gefunden; siehe auch [7]

In ganz frühen Versuchen stürzte die Shell schon beim Start nach etwa zehn Sekunden mit der Fehlermeldung `no space` ab.[16] Dieser „Platzmangel“ war dadurch zu erklären, daß auf einen `NULL`-Pointer zugegriffen wurde, die Signalbehandlung daraufhin (wegen Verdachts auf „zu wenig Speicher“) `sbrk` aufrief und dadurch ein weiterer Zugriffsfehler entstand. Dadurch wurden Signalbehandlungsroutinen geschachtelt aufgerufen, was nach einiger Zeit komplett den verfügbaren Speicher belegt hat. – Wer hätte gedacht, daß `no space` auf einen Segmentierungsfehler hinweist...

Diese Zugriffsverletzung geschah in der Funktion `getenv`, die die Shellinternen Umgebungsvariablen an die Außenwelt anpassen soll. Das Betriebssystem besitzt dafür eine Variable mit Namen `environ`, die den (Lese)Zugriff auf die Umgebungsvariablen erlaubt. Diese Variable enthielt einen `NULL`-Pointer, wodurch ein nicht erlaubter Speicherzugriff erfolgte, der zu besagter Rekursion führte.

Daß dieser Zeiger nicht gültig war, liegt an einem Unterschied zwischen K & R-C und ANSI-C: die Anweisung `char **environ;` war früher nur eine Deklaration der Variablen, der Linker hat dann den richtigen Speicherplatz aus dem System verwendet. In ANSI-C legt die Anweisung aber eine *neue* Variable an, die in diesem Fall den ungültigen Wert enthielt. (Siehe dazu auch 7.3, Seite 21.) Richtig ist hier die Deklaration `extern char **environ;`.

Das Setzen von Umgebungsvariablen funktionierte am Anfang nicht: bei Eingabe von `TERM=vt100` erschien nach ein paar Sekunden die Fehlermeldung `no space`. (Klingt vertraut, oder?) Hier lag die Ursache in der Funktion `syslook`, die die eingegebene Zeichenkette nach eingebauten Kommandos überprüft. Auch hier gab es einen Zugriff auf einen `NULL`-Pointer, der mit einer vorherigen Abfrage (anscheinend ohne Nebenwirkungen) einfach umgangen werden kann.[17]

Was u. U. negativ auffällt, ist der Shellprompt unter Linux, wenn die Umgebungsvariable `PS1` exportiert ist.[18] Die `bash` erkennt darin Symbole mit spezieller Bedeutung. So steht z. B. `\h` für den Rechnernamen (hostname), `\w` für das aktuelle Verzeichnis (working directory) und `\d` für das aktuelle Datum. Die `sh` versteht diese Zeichen jedoch nicht und gibt sie „im Original“ aus. Der durchaus nicht unübliche Prompt `"saturn:~$"` (Rechnername : aktuelles Verzeichnis) erscheint als `"\h:\w\$"`.

[16] Das war auf dem Rechner `velociraptor` mit 2 GB Swapspace. Daß die Shell nicht „völlig“ abstürzt, sondern noch einen Fehler produziert, fiel mir zufällig und erst dann auf, als ich sie startete und mich dann jemand kurz zu sich rief... – Wie lange es auf `saturn` mit 64 GB Swapspace dauert, wollte ich nicht ausprobieren...

[17] Auch an dieser Stelle frage ich mich, wieso das früher mal funktioniert hat...

[18] Dies bezieht sich auf eine mit dem `gcc` übersetzte Version, die auf einem realen System statt im ECO32-Simulator läuft.

8.6.3 Wildcardexpansion

Als sehr knifflig hat sich auch die als *Globbing* bekannte Expansion der Shellwildcars herausgestellt. `*`[19], `?` und `[]` stehen für die regulären Ausdrücke `".*"`, `"."` und die Angabe einer Menge von Zeichen bzw. eines Bereiches.

Warum die Wildcardexpansion nicht funktioniert, habe ich leider nicht herausgefunden. Eine Zeitlang dachte ich, es hinge vielleicht damit zusammen, daß `char` beim einem Compiler `signed` und beim anderen `unsigned` ist. Da der Compiler aber per Option angewiesen wurde, *alle* `char`s `unsigned` zu erzeugen, wird es daran wohl doch nicht liegen.

8.6.4 Weitere Programmierschwierigkeiten der `sh`

Da die Bourneshell ihre Speicherverwaltung selbst erledigt, verwendet sie das Symbol `_end`, das die Adresse des ersten nicht initialisierten Speicherplatzes im Datensegment hat (nur die Adresse von `_end` ist sinnvoll, nicht sein Wert). Nun war aber auch ein `address end[];` definiert, wobei `address` eine Union von Zeigern ist. Die frühere Link-Technik (siehe 7.3) hat wohl dazu geführt, daß die beiden Symbole dieselbe Speicheradresse hatten – zumindest hat alles prima funktioniert.

Das hat sich interessanterweise geändert, wenn man mit dem `gcc` übersetzt, um (vorab) auf einem realen System zu testen. Wenn dabei die Definition `address end[];` entfernt wird, stürzt die Shell beim Start mit der bekannten Meldung `no space` ab. Läßt man die Zeile aber in der Datei stehen, so funktioniert alles. Der wahre Grund dafür ist mir leider unbekannt. Möglicherweise wird so eine Menge Speicher verschwendet – aber ich war schon froh, daß ich den Grund für diesen Absturz gefunden habe; schließlich lief schon mal alles, bis die „falsche“ Definition entfernt wurde.

Einem Typ den Namen `FILE` zu geben (auch wenn er Dateien repräsentiert), führt zu Verwirrung. Schließlich ist `FILE` ein offizieller ANSI-Typ, auch wenn in C eigentlich nur mit `FILE *` gearbeitet wird. Dennoch stellt man sich ständig Fragen wie „wieso greift man hier auf Interna der Struktur zu?“, „wo ist das passende `fopen`?“ oder auch „wieso kann `FILE` als Funktionsparameter eingesetzt werden, wenn die alten Compiler keine Strukturen übergeben konnten?“[20].

Da `stdio.h` nicht verwendet wird, stellt dies nicht unbedingt einen Fehler da – wenn man davon absieht, daß der Name `FILE` reserviert ist und daher

[19] Gibt es eigentlich einen Zusammenhang zwischen dem Zeichen `*` und seinem ASCII-Code: 42 ?

[20] Die Erklärung hierfür ist simpel: `FILE` entsteht durch `typedef struct fileblk *FILE;`

geändert werden mußte. Ein wirkliches Problem tritt aber an anderer Stelle auf: viele Funktionen haben Namen, die (zumindest heute) reserviert sind. Der Linker beschwert sich zum Glück über diese Mehrfachdefinitionen, so daß man die „verbotenen“ Namen ändern kann (siehe auch Seite 21).

Insgesamt stellen die `sh`-Quelltexte beinahe eine Sammlung von Negativbeispielen dar – so sollte man nicht programmieren. Es erschwert das Verständnis (vor allem für andere, wie ich erfahren mußte...) und ist allenfalls für Compiler geeignet. Die Bourneshell wäre nicht so erfolgreich gewesen, wenn man vor Gebrauch ihren Quellcode hätte lesen müssen...

Kapitel 9

Ergänzung der Bibliotheken

Einige der portierten Programme benutzen Funktionen, die nicht Teil von ANSI-C sind. Diese waren daher auch nicht in der Bibliothek von Felix Grützmacher vorhanden, wurden aber gebraucht, damit die Programme korrekt gebunden werden konnten.

An dieser Stelle war es also nötig, die alten Bibliotheken zu ergänzen. Die dafür nötigen Quelltexte wurden bei der UNIX-Distribution mitgeliefert und standen daher zur Verfügung. Um sie verwenden zu können, mußte prinzipiell das gleiche getan werden wie bei den zu portierenden Programmen: die Quelltexte mußten ansifiziert werden, damit der Compiler sie akzeptierte. Anschließend war eine Headerdatei für jedes Modul zu schreiben, damit beim Übersetzen der Programme eine Typprüfung stattfinden konnte.

Um Konflikte mit den bereits vorhandenen Bibliotheken und Headerdateien zu vermeiden, habe ich meine Bibliothekserweiterung in einem separaten Verzeichnis entwickelt. `make` kopiert beim Erstellen die erzeugten `.s`-Dateien in das Verzeichnis, in das auch Felix Grützmacher seine übersetzten Dateien ablegte.

Da bei immer nur bei gelegentlichen Linkerfehlern eine Bibliothek ansifiziert werden mußte, fand dieser Vorgang interaktiv statt. Meist waren es nur wenige Funktionen, die in einem Bibliotheksquelltext vorkamen, und die Funktionsköpfe zu ändern und – in manchen Fällen – `#include`-Anweisungen hinzuzufügen, ging dann relativ schnell. Alles andere hätte sowieso von Hand gemacht werden müssen.

Kapitel 10

Vergleich der alten und neuen Programme

Die Unterschiede zwischen den originalen und den ansifizierten Quelltexten sind an einigen Stellen sehr deutlich, an anderen recht unauffällig.

Gering sind die Differenzen z. B. beim Miniprogramm `yes`; hier die alte Version:

```
main(argc, argv)
char **argv;
{
    for (;;)
        printf("%s\n", argc>1? argv[1]: "y");
}
```

Der Unterschied zur neuen Version besteht im wesentlichen in der zweiten Zeile: die Parametertypen sind in der Klammer mit angegeben. Das automatisch generierte `#include "yes_h.h"` ist natürlich nur in der neuen Version vorhanden.

```
#include "yes_h.h"
int main(int argc,char **argv)
{
    for (;;)
        printf("%s\n", argc>1? argv[1]: "y");
}
```

`yes_h.h` verwendet `stdio.h`, um `printf` zu deklarieren. Des weiteren enthält es die Deklaration von `main`. `yes_h.h` sieht wie folgt aus:

```
#ifndef __YES_H_H__
```

```
#define __YES_H_H__

/* Durchgereichte Praeprozessoranweisungen aus
 * der .c-Datei*/

/* #include's fuer Funktionen des Programms*/
#include <stdio.h>

/* Jetzt kommen die Funktionsdeklarationen des
 * Programms (inklusive bedingter Ueberseztung)*/
int main(int argc,char **argv);

#endif
```

Wesentlich größere Unterschiede findet man zwischen der alten und neuern Version von `at.c`. Der Kürze halber hier nur Ausschnitte aus einer `diff`-Ausgabe (mit Leerzeilen statt der hier eher sinnlosen Zeilenangaben):

```
< int utime;  /* requested time in grains */
---
> int utime_;  /* requested time in grains */

< char **environ;
< char *prefix();
< FILE *popen();
---
> extern char **environ;

< main(argc, argv)
< char **argv;
---
> int main(int argc,char **argv)

<  extern onintr();

<  filename(THISDAY, uyear, uday, utime);
---
>  filename(THISDAY, uyear, uday, utime_);

>  /*UNREACHED*/
>  return 0;
```

```
< makeutime(pp)
< char *pp;
---
> void makeutime(char *pp)

< filename(dir, y, d, t)
< char *dir;
---
> void filename(char *dir,int y,int d,int t)

< onintr()
---
> void onintr(int signo)
```

Einige Unterschiede lassen sich schön erkennen[1]:

- Der Name `uname` ist reserviert und wurde daher in `uname_` geändert.
- `environ` ist eine *`extern`*e Variable; es soll keine neue Variable mit dem Namen `environ` angelegt werden.
- Die Deklarationen von `prefix` und `popen` enthalten keine Parametertypen und wurden entfernt; die vollständigen Deklarationen befinden sich in `at_h.h`.
- `main` gibt ein `int` zurück und hat die Parametertypen in den runden Klammern stehen.
- Auch bei `onintr` (eine Deklaration in `main`) fehlen die Typen der Parameter; die Deklaration wurde entfernt.
- `main` enthielt keine `return`-Anweisung (es gab „nur“ einen `exit`-Aufruf; siehe 8.4.5).
- `makeutime` und `filename` geben keinen Wert zurück; die Typen können jetzt geprüft werden, da sie *in* den runden Klammern stehen.
- `onintr` ist Parameter von `signal` und erhält seinerseits die Signalnummer als Parameter. Auch wenn die Signalnummer nicht verwendet wird, ist die Deklaration erforderlich, damit formaler und aktueller Typ des Arguments von `signal` zusammenpassen.

[1] Darum sind sie auch so gewählt...

Kapitel 11

Testen der portierten Programme

Da sich beim Ändern von Programmen immer Fehler einschleichen können, ist es nötig, im Anschluß an die Ansifizierung der Programme zu überprüfen, ob sich das Originalprogramm und die „neue Version“ unterscheiden. Dazu kann man nicht die binären Programme vergleichen, da diese für verschiedene Maschinen übersetzt wurden und sich dadurch *garantiert* unterscheiden.

Die einzig praktikable Methode besteht darin, beide Programme mit den selben Eingabedaten zu füttern (sofern vom Programm benötigt), und die dabei entstehenden Ausgabedaten zu vergleichen. Auch diese Arbeit kann man beliebig ausdehnen, da streng genommen jede Programmoption mit allen theoretisch denkbaren Eingabedaten gestestet werden müßte. Selbst ohne Optionen ist diese Aufgabe nicht in endlicher Zeit bewältigbar, da die Eingabedaten natürlich unendlich lang sein dürfen[1]. Aus diesem Grund werden einige (endliche) Testdaten festgelegt, mit denen die Programme aufgerufen werden.

Da UNIX-Programme schon seit Urzeiten etliche verschiedene Optionen haben und dieselben Buchstaben bei unterschiedlichen Programmen meist etwas anderes bedeuten, wird auch ein vollständiger „Optionentest“ schnell zur Sisyphusarbeit, wenn alle möglichen Optionskombinationen überprüft werden sollen. Wer dabei auf Nummer sicher gehen will und wenigstens alle Kombinationen von einstelligen Optionen probieren will, muß „eine Weile warten“. Bei 128 möglichen Zeichen ist nur ein zweistelliger Test noch durchführbar – und auch der dauerte in einem Test bereits etwa zwei Minuten für ein Programm![2] Auch das Weglassen der nicht druckbaren Zeichen

[1] Zur Konstruktion von simplen unendlich langen Datenströmen lese man die Manpange von `yes`.

[2] Der Aufruf `time swo 3 echo` zum Messen der benötigten Zeit bei *drei* Optionen

brächte nur einen verhältnismäßig geringen Vorteil, so daß es leichter ist, einfach alle Zeichen zu verwenden.

Auch hier kann man sich die Arbeit natürlich „vereinfachen“, indem man die optimistische Strategie fährt: wenn die einzelnen Sachen klappen, wird das Ganze auch irgendwie gutgehen. Da nur sichergestellt werden soll, daß sich alte und portierte Programme gleich verhalten, spielt es keine Rolle, wenn ungültige Optionen angegeben werden – wenn es einen Fehler gibt, sollte er bei beiden Programmen auftreten (und sich auf die gleiche Weise äußern).

Problematisch wird es aber, wenn Optionen nicht nur aus *einem*, sondern aus *mehreren* Zeichen bestehen, oder wenn noch weitere Parameter angegeben werden müssen (z. B. `mount /dev/hda3 /home`, `dd if=/dev/hda1 of=sicherung` oder `find . -type d -name '*src' -print`[3]).

Im Gegensatz zu modernen Versionen akzeptierten die damaligen Programme („zum Glück“) nur recht wenige Optionen, was das Testen ein wenig leichter macht. Einige Programme können vollständig automatisch getestet werden; bei vielen ist es jedoch empfehlenswert, von Hand einige sinnvolle Optionen durchzuprobieren und die Ergebnisse (automatisch) zu vergleichen. Dies sind z. B. `sed` und `tar`. Schwierig bis unmöglich automatisch zu testen ist `login`, da sich hier die Ausgabe nicht so einfach umleiten läßt. Selbst interaktive Tests von `login` sind zum gegenwärtigen Entwicklungsstand nicht so einfach, weil dazu ein gewisser Teil des UNIX-Verzeichnisbaums aufgebaut werden müßte.

Wegen komplexer Optionen schwierig zu testen sind:

```
ar    at     basename  cmp  cp     dd    diff   diff3  ed
find  kill   lex       ln   login  ls    mount  mv     pwd
rm    rmdir  sed       sh   tar    test  time   touch  uniq
yacc
```

Das Problem bei diesen Programmen ist, daß „sinnvolle“ Parameter und Eingaben nötig sind, um halbwegs repräsentative Tests zu haben. `ls` braucht für die gleiche Ausgabe auch denselben Verzeichnisinhalt, bei manchen Optionen streng genommen dieselben Namen, Größen, Zeiten, `inode`s usw. . . – hier dürfte ein automatischer Vergleich der Ausgaben mit normalen Mitteln kaum möglich sein; man müste schon irgendwie tricksen. Aber auch beim simplen

lieferte auf einer realen Maschine bereits den Wert `real: 336m` – das ist für „ordentliches“ Testen mit regelmäßigen Korrekturen bereits für *ein* Programm viel zu lange. . . – Da der Wert *exponentiell* ansteigt, braucht man bei der aktuellen Geschwindigkeit von Computern nicht über noch längere Zeichenketten nachzudenken.

[3]Die Optionen sind einem Linux-System entnommen, treffen prinzipiell aber auch auf das alte UNIX zu.

`time` muß ein Programmname angegeben werden, wobei zu bezweifeln ist, daß dasselbe Programm auf zwei verschienen (auch noch virtuellen) Maschinen in derselben Zeit abläuft. Von komplizierten Eingaben wie für `sh` oder gar `ed` will ich gar nicht erst reden...

Es bleibt hier wieder nur die Hoffnung, daß – in den meisten Fällen – dieselbe Ausgabe der Programme auf korrektes Funktionieren hindeutet.

Leider lassen sich aber gar nicht alle Programme testen. Bedauerlich ist dies vor allem bei `ps`. Es greift auf die Kerneldatei `/unix` zu, die auf ECO32 ein völlig anderes Format hat als auf der PDP-11. Von daher läßt es sich nicht nur nicht testen, sondern es läuft noch nicht mal...

11.1 Automatisches Testen

Hier liegt die Betonung auf *automatisch* – schließlich habe ich weder Zeit noch Lust, etliche Programmaufrufe mit immer neuen Optionen durchzuführen und dann auch noch die Gleichheit der Ausgaben auf ECO32 und der PDP-11 nachzuweisen...

Aber so etwas ist ja zum Glück nicht nötig – wofür gibt es denn schlielich Computer?! Ein Programm, daß sowohl unter K & R-C als auch unter ANSI-C übersetzbar ist, war zwar eine Herausforderung, aber wirklich schwierig war es nicht. Nötig dafür war allerdings der Präprozessor, damit – je nach System und Compiler – der alte oder der neue Stil der Funktionsköpfe verwendet wurde.

Das Programm `swo`[4] bekommt die maximale Länge des zu generierenden Parameters[5] und den Namen eines Programms übergeben. Anschließend generiert es die möglichen Optionen (im Sinne von `a`, `b`, ..., `z`, `aa`, `ab`, `ac`, ...) und startet das angegebene Programm mit dieser Zeichenkette (und optional zusätzlich übergebenen Parametern).

```
/* swo - start with options
 * Startet ein Programm mit genertierten und allen
 * uebergebenen Optionen.
 *
 * Da dieses Programm auf dem ECO32- und dem
 * PDP11-Simulator laufen soll, muss es ANSI-C- und
 * K&R-C-kompatibel sein!
 */
```

[4] `start with options`

[5] Es sollten nicht mehr als *2* Buchstaben getestet werden; siehe Seite 46.

```
#ifdef __STDC__
#    ifndef ECO32
#        define ECO32
#    endif
#else
#    define PDP11
#endif


#include <stdio.h>
#include <ctype.h>
/* Diese Headerdateien gibt's auf der PDP-11 nicht */
#ifdef ECO32
#    include <unistd.h>
#    include <string.h>
#    include <stdlib.h>
#    include <limits.h>
#else
#    define CHAR_MAX 127
#endif


#define MAX_CHARS 127
#define LOGFILE "swo.log"
#define COMMAND_INDEX 2


#ifdef PDP11
/* Damals gab es scheinbar noch kein memset... */
/* Durch den Extrablock wird ein evtl. aeusseres i
 * ueberdeckt.
 */
#    define memset(ptr,val,count)\
    {\
        int i;\
        for (i = 0; i < count; i++)\
        ((char *)ptr)[i] = val;\
    }
/*Deklaration von realloc*/
char *realloc();
char *calloc();
```

```
#endif


#ifdef ECO32
char *getargs(const char *program,
    const char *generated_arg,
    const char **given_arg);
int execlvp(const char *file,
    const char *arg,
    const char **argv_in);
#else
char *getargs();
int execlvp();
#endif


#ifdef ECO32
int main(int argc, const char *argv[])
#else
int main(argc, argv)
    int argc;
    char **argv;
#endif
{
    int i, pid, anz_chars, retval;
    unsigned char args[MAX_CHARS];
    char *progcall = NULL;
    FILE *logfile;

    if (argc < 3) {
        fprintf(stderr, "syntax: %s <number of chars> "
            "<command> [options...]\n", argv[0]);
        return 1;
    }

    anz_chars = atoi(argv[1]);
    if (anz_chars > MAX_CHARS) {
        fprintf(stderr, "the maximum number is %d\n",
            MAX_CHARS);
        abort();
    }
```

```
memset(args, 0, sizeof(args));
logfile = fopen(LOGFILE, "wb");
if (logfile == NULL) {
    perror("fopen-error");
    return 1;
}


i = 0;
while (i < anz_chars) {
    i = 0;
    args[i]++;
    /*"Uebertraege" bearbeiten*/
    while (args[i] >= CHAR_MAX) {
        args[i] = 0;
        i++;
        args[i]++;
    }

    switch (pid = fork()) {
        case -1:
            perror("fork-error");
            return 1;
            /*NOTREACHED*/
            break;
        case 0:
            execlvp(argv[COMMAND_INDEX], (char *) args,
                &argv[COMMAND_INDEX + 1]);
            perror("exec-error");
            return 1;
            /*NOTREACHED*/
            break;
        default:
            /* Einen sh-verstaendlichen Aufruf ins
             * Logfile schreiben
             */
            progcall = getargs(argv[COMMAND_INDEX],
                (char *) args,
                &(argv[COMMAND_INDEX + 1]));
            if (progcall == NULL)
```

```
                    perror("getargs returned NULL");
                else fprintf(logfile, "%s\n", progcall);
                wait(&retval);
                fprintf(logfile, "%d\n\n",
                    (retval >> 8) & 0377);
                break;
        }
    }

    return 0;
}


#ifdef ECO32
char *getargs(const char *program,
    const char *generated_arg,
    const char **given_args)
#else
char *getargs(program, generated_arg, given_args)
    char *program, *generated_arg, **given_args;
#endif
{
    int i, len;
    static char *str = NULL;
    char *tmp;

    len = strlen(program) + 1 /*' '*/ +
        strlen(generated_arg) + 1 /*' ' oder '\0'*/;
    if (given_args != NULL) {
        for (i = 0; given_args[i] != NULL; i++) {
            len += strlen(given_args[i]) + 1;
                                        /*' ' oder '\0'*/
        }
    }
    /* Auf der PDP-11 scheint realloc mit NULL nicht zu
     * funktionieren
     */
    if (str == NULL) str = calloc(len, sizeof(char));
    else {
        tmp = realloc(str, len * sizeof(char));
        if (tmp == NULL) return NULL;
```

```
        else str = tmp;
    }

    /* Frueher gab sprintf den String zurueck, NICHT die
     * Anzahl der geschriebenen Zeichen.
     * Diesen Unterschied zu finden, hat mich ueber eine
     * Stunde gekostet...
     */
    sprintf(str, "%s ", program);
    len = strlen(program) + 1;
    sprintf(str + len, "%s", generated_arg);
    len += strlen(generated_arg);
    if (given_args != NULL) {
        for (i= 0; given_args[i] != NULL; i++) {
            sprintf(str + len, " %s", given_args[i]);
            len += strlen(given_args[i]) + 1;
        }
    }

    return str;
}


#ifdef ECO32
int execlvp(const char *file,
    const char *arg,
    const char **argv_in)
#else
int execlvp(file, arg, argv_in)
    char *file, *arg, **argv_in;
#endif
{
    char **argv;
    int i, j, len;

    for (i = 0, len = 0; argv_in[i] != NULL; i++) len++;

    argv = (char **) calloc(len + 2, sizeof(char *));
                /*+2: file am Anfang und NULL am Ende*/
    if (argv == NULL) return 1;
```

```
    argv[0] = (char *) file;
    argv[1] = (char *) arg;
    for (i = 0, j = 2; argv_in[i] != NULL; i++, j++)
        argv[j] = (char *) argv_in[i];
    argv[j] = NULL;

    /* execvp ist auf ECO32 und System V verschieden
     * deklariert:
     * ECO32:
     *        int execvp(const char *file,
     *            const char *argv[]);
     * System V (saturn):
     *         int execvp(const char *file,
     *             char *const argv[]);
     */
#ifdef ECO32
    return execvp(file, (const char **) argv);
#else
    return execvp(file, argv);
#endif
}
```

`swo` schreibt dabei eine Logdatei mit dem Namen `swo.log`, in der jeder Aufruf vollständig enthalten ist:

<Programmname> <generierte Zeichenkette> <angegebene Optionen>

In die folgende Zeile wird der Wert geschrieben, den das Programm an den Aufrufer zurückgegeben hat.

Ein schönes Demonstrationsbeispiel für `swo` ist: `swo 1 echo test`. Dabei gibt `echo` jedes ASCII-Zeichen aus, jeweils gefolgt von der Zeichenkette `test`. Hier ein Ausschnitt aus der Ausgabe:

```
7 test
8 test
9 test
: test
; test
< test
= test
> test
? test
```

```
@ test
A test
B test
C test
D test
E test
```

Natürlich funktioniert das ganze mit den meisten anderen Programmen nicht so gut wie mit `echo`, aber es ist dennoch ein einfacher Weg, um einige Zeichenkombinationen als Programmparameter durchzuprobieren. Die Ausgabe läßt sich dabei in eine Datei umlenken, um sie später mit der Ausgabe auf dem anderen System vergleichen zu können. Sogar die Logdateien sind für diesen Test verwendbar; allerdings bezieht sich hier „Gleichheit“ auf den Exitcode des Programms.

Zum Umleiten der Standardausgabe ist ist die Bourneshell `sh` nötig, da Felix Grützmacher's Shell keine Umlenkung in Dateien unterstützt. Eine Alternative wäre das Schreiben eines Programmes („`redirect`“ oder so), das seine Eingabe in eine Datei umleitet, und das Verwenden einer Pipe (dies gestattet die Shell) oder – wenn die Ausgaben nicht stören – die Benutzung von `tee`. – Das Umlenken der Standardeingabe hingegen ist bei der Verwendung von Pipes kein Problem: hierfür kann man einfach `cat` verwenden.

Nachdem im PDP-11- und im ECO32-Simulator `swo` mit allen Programmen gestartet und die Ausgabe entsprechend umgeleitet wurde, kann der Inhalt der virtuellen Platten extrahiert werden. Die entsprechenden „Logdateien“ lassen sich nun per `diff` miteinander vergleichen.

Leider genügt die Ausgabe von `diff` noch nicht, um wirklich Unterschiede zwischen den Original- und den portierten Programmen (und damit „Fehler“ in letzteren) nachweisen zu können. Viele Programme greifen auf Dateien zu, die es in dem noch unvollständigen UNIX-System einfach nicht gibt: `spell` z. B. sucht seine Wörterbücher in `/usr/dict/`. `/etc/passwd` und `/etc/group` können einfach vom alten System übernommen werden; sie ggf. anzupassen wäre nicht schwer. Für `learn` aber werden verschiedene Lektionen benötigt. Zwar sind sie mit der UNIX-Distribution mitgeliefert, aber sie müssen erst gebrauchsfähig gemacht werden. Es ist fraglich, ob man `make` mit *so* etwas testen sollte – wenn mit `learn` etwas nicht funktioniert, beginnt man vielleicht dort zu suchen, obwohl bei `make` etwas nicht stimmt.[6]

Trotz dieser und anderer Schwierigkeiten (siehe Kapitel 12) habe ich interaktiv schon einige Programme gestestet. Fehlerfrei *schienen* zu funktionieren:

[6]Das erinnert an die ungünstige Situation, in der man ein vermeintlich fehlerhaftes Programm debuggt, obwohl *bei einem der Tests* ein Fehler gemacht wurde.

`accton`	Hier gibt es keine Ausgabe, die man überprüfen könnte. Eine angegebene Datei (sie muß existieren!) wird aber verändert.
`basename`	Das Entfernen des Verzeichnisnamens und – optional – eines Suffixes funktioniert.
`cal`	Der Kalender von 2004 wurde korrekt ausgegeben.
`cat`	Dateien wurden korrekt ausgegeben; Lesen der Standardeingabe funktioniert jedoch nicht.
`echo`	Die angegebenen Zeichenketten wurden alle ausgegeben. (Bei der `sh` kann man auch Umgebungsvariablen erstellen und diese anschließend ausgeben.)
`file`	Das ist natürlich nur schwer „komplett“ zu testen, da man hierfür etliche verschiedene Arten von Dateien bräuchte. Bei einigen simplen Dateien (`/etc/passwd`, `/etc/group`, `/unix` und einer leere Datei) kamen jedoch halbwegs sinnvolle Ergebnisse (zweimal `ASCII text`, `Data` und `empty`).
`ln`	Hardlinks scheinen zu funktionieren; auch `ls` meldet für „beide“ Dateien, denselben `inode` und dieselbe Größe und Modifikationszeit. – Symbolische Links gab es damals nocht nicht; siehe [7].
`pwd`	Man muß sich für den Test zwar merken, in welches Verzeichnis man gewechselt hat, aber dann kann man es leicht überprüfen.
`sync`	Da gibt es auch nicht viel, das schief gehen könnte...
`wc`	Zählen von Zeilen, Worten und Zeichen aus Dateien oder der Standardeingabe funktioniert; auch die Optionen klappen.
`yes`	Ohne weitere Angaben wird `y` ausgegeben, sonst der erste Parameter. Auch hier „kann“ eigentlich nichts falsch laufen.

Natürlich ersetzen diese spontanen Tests keine wirkliche Testreihe. Sie sind aber aber ein Anzeichen dafür, daß die Portierung prinzipiell funktioniert hat.

Interessant ist allerdings die Frage, wie man z. B. `sed` und `ed` vollständig testen will. Eine kleine Textdatei mit `ed` zu bearbeiten, hat funktioniert – aber wie ist das bei größeren Dateien? `ed` besitzt etliche Kommandos; diese in allen möglichen Reihenfolgen durchzuprobieren, ist kaum möglich. Hier bleibt einem nur die Verfahrensweise, die auch große Softwarekonzerne mehr oder weniger erfolgreich anwenden: hoffen, daß alles funktioniert, und im Fehlerfall nachbessern.[7]

[7] Bei der Entwicklung neuer Software kann man das Testen allerdings ganz anders realisieren als hier. Es lassen sich von Anfang an automatisierte Tests in die Entwicklung einbauen, was bei dieser Portierung leider nicht möglich ist. Ein Grund dafür ist, daß man (noch) nicht auf der Zielmaschine direkt entwickeln, übersetzen und sofort testen kann.

Kapitel 12

ToDo – was bisher nicht erledigt wurde

Leider war es mir nicht möglich, in der mir zur Verfügung stehenden Zeit *alle* Probleme zu lösen. Bedauerlich ist in diesem Zusammenhang, daß während der Tests ein Fehler im UNIX-System aufgefallen ist, der nicht innerhalb kurzer Zeit behoben werden konnte. Aus diesem Grund konnte das Testen der Programme nicht wirklich stattfinden.

Außerdem funktioniert das Shellglobbing noch nicht: die Bourneshell expandiert keine Wildcards (siehe 8.6.3)

Literaturverzeichnis

[1] Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language (dt.)*, ISBN 3-446-13878-1

[2] Peter Schnupp: *Von C zu C – Problemlos portieren,* Hanser Programmtexte; ISBN 3-446-15945-2

[3] Dennis Kuhn: *Porting the Unix v7 kernel to the RISC Processor Eco32*

[4] Prof. Dr. Hellwig Geisse; die Internetadresse für den ECO32-Simulator: *http://telexx.mni.fh-giessen.de/ECO32/*

[5] Prof. Dr. Hellwig Geisse; die Internetadresse für den PDP11-Simulator: *http://telexx.mni.fh-giessen.de/PDP-11/*

[6] Peter Van der Linden: *Expert C Programming,* 1994, PTR Prentice Hall, ISBN 0-13-177429-8

[7] *UNIX Programmer's Manual,* Seventh Edition, Volume 1; als Postscript-Datei enhalten im Softwarepaket des ECO32-Simulators [4]

[8] Bjarne Stroustrup: *Die C++-Programmiersprache,* 3., aktualisierte und erweiterte Auflage, 1998, Addison-Wesley, ISBN 0-201-88954-4

[9] Herb Sutter: *Exceptional C++,* 2000, Addison-Wesley, ISBN 3-8273-1711-8

[10] Helmut Kopka: *LaTeX – Band 1: Einführung,* 3., überarbeitete Auflage, 2000, Pearson Studium, ISBN 3-8273-7038-8

Index