
fd0-Linux

mit uClibc und Busybox

Stand: 10. August 2002
Version 0.6, beta

erstellt von:
Michael Bär
`micb@easy-penguin.de`
<http://www.easy-penguin.de/>

Erstellt mit L^AT_EX 2_ε sowie dia und xfig

Inhaltsverzeichnis

1	Einleitung	4
1.1	Geschichte	4
1.2	Gegenwart	4
1.3	Benutztes System	4
2	Vorbereitung der Entwicklungsumgebung	5
2.1	Verwendete Pakete und Dokumente	5
2.2	Vorbereitungen	6
2.3	Temporäres root-Filesystem	7
2.3.1	ramdisk	7
2.3.2	loopback-Device	8
2.4	Kernel	9
2.5	uClibc und Busybox	9
2.5.1	NFS	9
2.5.2	uClibc einrichten und benutzen	10
2.5.3	busybox konfigurieren und installieren	10
2.5.4	Syslinux installieren	11
3	initrd erstellen	12
3.1	Ramdisk preparieren	12
3.1.1	Verzeichnisse erstellen	12
3.1.2	Devices und /etc erstellen	12
3.1.3	busybox	12
3.1.4	uClibc runtime environment	12
3.1.5	Bibliotheken bekannt machen	12
3.1.6	Erzeugen einer initial-Ramdisk	12
3.1.7	Kernel Kompilieren	13
3.2	Zusätzliche Programme	13
3.2.1	sfdisk	13
3.2.2	Keyboard-map	13
3.2.3	Kompilieren von Samba	13
3.2.4	strip	14
3.2.5	Kernel	14
3.2.6	Elementare Optionen	14

<i>INHALTSVERZEICHNIS</i>	3
4 Preparieren der Diskette	16
4.1 Die Größe	16
4.2 Formatieren	16
4.3 Dateien übertragen	16
4.4 Syslinux	17
5 Startup und Skripte	18
5.1 Grundlagen des Linuxstarts	18
5.2 Startup der Diskette	18
5.2.1 rcS	18
5.2.2 startup.sh	18
5.2.3 options	21
5.2.4 postboot	22
5.2.5 scripts.tgz	22
5.3 Besonderheiten für die Scriptprogrammierung	23
5.3.1 Erklärung	23
5.3.2 Übersicht	24
5.4 Fehlermeldungen	24
6 Arbeiten mit der Bootdiskette	25
6.1 Besonderheiten der Bootdiskette	25
6.1.1 Mounten per NFS	25
6.1.2 Mounten per SMB	25
7 Anhang	26
7.1 Einstellungen in den config-Files	26
7.1.1 Einstellungen in der busybox-Config	26
7.1.2 Einstellungen in der uClibc-Config	28
7.1.3 Einstellungen in der Kernel-Config	29
7.1.4 makedev	31

1 Einleitung

1.1 Geschichte

Die erste Version dieser Anleitung entstand 2001 während meines praktischen Studiensemesters bei der IBM Deutschland GmbH.

Das eigentliche Projekt war einen Installationsprozess für verschiedene Linux-Distributionen (SuSE und RedHat) zu implementieren. Dieser wurde dazu gebraucht, um Schulrauminstallationen für Linuxkurse neu aufzuspielen.

Für die Installation musste ich von folgendem ausgehen: Es existiert kein lauffähiges Betriebssystem auf den jeweiligen Rechnern.

Also musste ich dafür sorgen, dass zumindest ein kleines, lauffähiges, System auf diesen Rechner installiert wurde. Dieses erledigte ich mit eben dieser Bootdiskette, die das komplette System in den Hauptspeicher schreibt. Somit war auch gewährleistet, dass keine überflüssigen Datensegmente auf der Festplatte waren, wenn das Image zurückgespielt wurde.

Soweit zur Vorgeschichte.

1.2 Gegenwart

Eigentlich brauche ich die Diskette nun nicht mehr und ist eigentlich nur noch ein netter Zeitvertreib. Allerdings macht es immer wieder auf ein neues Spass, herauszufinden, wieviele Programme man auf eine Diskette packen kann ;-)

Ich habe zur Urversion einige Programme entfernt, benutze natürlich neuere Pakete und die Startupscripte wurde auch ein wenig aufgeräumt, da ich manche Stellen einfach nicht mehr benötige.

Aber im Kern ist die Diskette immer noch die gleiche - sie basiert immer noch auf der kleinen, unscheinbaren Anleitung im Unterverzeichnis *bootfloppy* des busybox-Paket[2].

1.3 Benutztes System

Die erste Bootdiskette hatte ich auf einem Standard SuSE 7.2 (ich will kein Wort hören ;-)
) System erstellt. Dieses System war auf einem IBM ThinkPad 390x installiert.

Zusätzlich benutzte ich noch meinen Privatrechner, damals mit einem Mandrake 7.2 System.

Ich hatte keine besonderen Zusatzpakete der Distributionen installiert. Es handelte sich um Standardinstallationen.

In der Zwischenzeit bin ich nicht mehr bei IBM und die Bootdiskette wird nur noch auf meinem Privatrechner weiterentwickelt. Es handelt sich hierbei in der Zwischenzeit um ein Mandrake 8.1 System.

Somit wäre der Beweis erbracht, dass sich die Diskette auf jedem Rechner erstellen lassen sollte.

2 Vorbereitung der Entwicklungsumgebung

In diesem Kapitel wird *nur die Entwicklungsumgebung* eingerichtet, unter der die Bootdiskette erstellt wird. Am Ende dieses Kapitels werden wir soweit sein, dass wir ein leeres Rootfilesystem für die Diskette haben und die wichtigsten Programme soweit fertig sind, dass wir sie in das Rootfilesystem ¹kopieren können.

2.1 Verwendete Pakete und Dokumente

Für die Erstellung dieser Bootdiskette habe ich folgende Pakete benutzt:

- busybox <<http://busybox.lineo.com/>>
Bei *busybox* handelt sich um ein einzelnes Programm, das die grundlegenden Funktionen vieler Linux-Befehle beherrscht. Der Vorteil von *busybox* ist die sehr geringe Größe des Programmes (ca. 200kByte). Weiter unten habe ich die Befehle aufgelistet, die durch dieses Programm abgedeckt werden (Siehe Seite 26, Kapitel 7.1.1).
Verwendete Version: 0.60.2
(Download: <ftp://oss.lineo.com/busybox/>)
- uClibc <<http://cvs.uclinux.org/>>
uClibc sind kleine c-Bibliotheken, die für den Einsatz in Embedded Systems entwickelt wurden. Sie haben gegenüber den Systemeigenen Bibliotheken den Vorteil, dass sie sehr wenig Platz auf der Bootdiskette benötigen. Ausserdem werden die kompilierten Programme nicht so groß wie mit den Standard Bibliotheken.
Verwendete Version: 0.9.12
(Download: <http://www.uclibc.org/>)
- syslinux <<http://syslinux.zytor.com/>>
Syslinux ist, ähnlich wie Lilo, ein Startprogramm für Linux. Der Vorteil von Syslinux ist weniger die Größe, als vielmehr die Tatsache, dass man eine Diskette nur einmal Einrichten muss. Danach kann man Kernel und initial Ramdisk auf diese Diskette kopieren, ohne dass man jemals wieder den Bootsektor neu beschreiben muss (wie bei Lilo üblich).
Alternativ kann man auch das RPM/DEP-Paket der Distribution installieren.
Verwendete Version: 1.63
- sfdisk
<http://www.turbolinux.com.cn/pub/turbolinux/workstation/6.0-cn/i386/misc/src/disk_partition/sfdisk>
Mit diesem Programm kann man Partitionen automatisch aus einem Script erstellen lassen.
Verwendete Version: 3.07 - 980518

¹Ein root-Filesystem ist ein Filesystem mit allen wichtigen Verzeichnissen und Kommandos. Nicht zu verwechseln mit der initrd, die nur eine gepackte Datei des root-Filesystems ist.

- lilo <<http://www.ibiblio.org/pub/Linux/system/boot/lilo/>>
Ich habe den Originalbootloader kompiliert, damit ich unabhängig von der Distribution meinen eigenen anlegen kann. Somit bin ich unabhängig von den Launen der Hersteller.
Verwendete Version: 22.3
- e2fsprogs <<http://e2fsprogs.sourceforge.net/>>
Diese Tools werden benötigt, um das Filesystem auf der Festplatte anzulegen. Ich habe die neueste Version verwendet, damit ich die Journalingfähigkeit des ext2 Filesystems (=ext3) nutzen kann.
Verwendete Version: 1.27
- Kernel <<http://www.kernel.org/>>
Im Prinzip ist es egal, welchen Kernel benutzt. Allerdings ist der Kernel 2.2 doch um einiges kleiner, als der Kernel 2.4. Trotzdem benutze ich den 2.4er Kernel, da dieser mit neuerer Hardware (large file support, Speicher > 1 GB) besser zurechtkommt.
Den Kernel 2.0 sollte man in Frieden ruhen lassen. Er hat seine Schuldigkeit getan.
Verwendete Version: 2.4.18
- Samba <<http://www.samba.org/>>
Die hier verwendete Version ist die letzte aus der 1.x Serie. Ich habe diese Version aus Platzgründen verwendet. Alle neueren Samba-Pakete sind doch ein wenig zu groß für eine Diskette, wenn man noch ein paar andere Programme mit draufpacken möchte.
Verwendete Version: 1.9.18p8

2.2 Vorbereitungen

Um eine Bootdiskette erstellen zu können, müssen einige Dinge auf einem laufenden Linux installiert werden.

Es wird empfohlen die gleichen Namen zu verwenden, die ich auch verwendet habe. Denn ich werde den Rest der Anleitung eben diese benutzen.

- ramdisk/loopback-Device
Um das root-Filesystem zu erstellen benötigt man eine eigene Partition. Entweder man legt eine 4 MB große Partition auf der Festplatte an oder man benutzt ein loopback-Device oder eine Ramdisk.
Ich benutze entweder ein loopback-Device oder eine Ramdisk. Was das jeweils ist, und wie es eingerichtet wird, folgt etwas später.
- uClibc
Diese Bibliotheken muss man auf dem Rechner installieren, auf dem man die Pakete für die Bootdiskette erstellen möchte. Der Grund ist ganz einfach. Die Pakete müssen beim Kompilieren auf die Bibliotheken gelinkt werden, die zur Laufzeit zur Verfügung stehen werden. Also auf unsere uClibc.
Man muss sich um seine Systembibliotheken im übrigen keine Sorgen

machen. Die uClibc wird parallel dazu installiert und nur benutzt, wenn man dies explizit festlegt.

2.3 Temporäres root-Filesystem

Das root-Filesystem enthält ein vollständiges Linux System mit allen Verzeichnissen und einem Standardfilesystemtyp. Ich verwende immer ext2². Der Grund ist einfach der, dass ext2 im Kernel eincompiliert sein sollte, da viele Systeme noch diesen Filesystemtyp benutzen. Jedes weitere unterstützte Filesystem benötigt zusätzlich Speicher - und der ist auf einer Diskette sehr kostbar.

Um dieses Filesystem anzulegen, muss man natürlich kein physikalische Gerät (Festplatte oder ähnliches) zur Verfügung stellen. Linux hilft uns dabei auf andere Art.

Im Prinzip gibt es zwei Möglichkeiten, um ein temporäres Dateisystem anzulegen: ramdisk oder loopback.

Beide Möglichkeiten haben den Vorteil, dass man nichts an der ursprünglichen Plattenpartitionierung ändern muss. Es sind jeweils Möglichkeiten, um virtuelle Platten zu erstellen. Entweder in einer Datei (loopback) oder im Hauptspeicher (ramdisk). Kleiner Tip am Rande zum loopback: Wer ein ISO-Image einer CD-Rom auf seiner Platte rumliegen hat (z.B. die Mandrake-CDs) kann man diese per loopback Mounten und hat Zugriff auf die Daten ohne diese brennen zu müssen. Aus diesem temporären root-Filesystem wird später die initrd³

Welche dieser Möglichkeiten genutzt wird, ist im Prinzip völlig egal.

Ich gehe im Folgenden von diesen Vorgaben aus:

Mountpunkt: /mnt/rootfs

Datei für loopback-Device: /mnt/initrd

2.3.1 ramdisk

Die ramdisk muss natürlich vom Kernel unterstützt werden:

```
cat /usr/src/linux/.config — grep ^CONFIG_BLK_DEV_RAM
```

Ausgabe: CONFIG_BLK_DEV_RAM=y und die Größe CONFIG_BLK_DEV_RAM_SIZE=4096

Die meisten Standardkernel tun dies von Haus aus. Beachtet: Ich rede hier vom Kernel des Hostsystems, also dem System auf dem die Bootdiskette entwickelt wird, nicht vom Kernel für die Bootdiskette, der dies unterstützen muss!

Danach muss man nachprüfen, ob ein ramdisk-Device existiert. Diese findet man im Verzeichnis /dev. Ein ll /dev/ram1 sollte ein ähnliches Ergebnis zu Tage fördern:

```
brw-rw---- 1 root disk 1, 1 Oct 23 13:43 /dev/ram1
```

² extended filesystem 2 ist das Standardfilesystem von Linux

³initial ramdisk: Diese Datei enthält ein vollständiges Linux-Dateisystem und wird vom Kernel in den Hauptspeicher geladen und dort gehalten

Sollte das System so ein Device besitzen, kann man den nächsten Schritt übergehen. Ansonsten muss man mit **mknod /dev/ram1 b 1 1** ein solches Device erzeugen. Am besten erzeugt man mehrere dieser Devices. **mknod /dev/ram{n} b 1 {n}** . {n}=Ziffer des Device, oben die "1".

Danach muss man den benötigten Platz zur Verfügung stellen. Dies erledigt man mit dem Befehl **dd if=/dev/zero of=/dev/ram1 bs=1k 2500** . Damit werden 2500 1k-Blöcke vom Device "zero" (Nullstream) ins Device "ram1" (Ramdisk) kopiert.

Nun muss man noch ein Filesystem auf dieser Ramdisk anlegen. Dies erledigt man mit dem Befehl **mke2fs -m0 -i1400 /dev/ram1 2500** . Damit wird ein ext2-Filesystem angelegt. Wir stellen dem Superuser keinen extra Platz zur Verfügung (-m0) und wollen 1400 bytes per inode. Die Ramdisk soll 2500 Blocks groß sein.

Nun sollte man diese Ramdisk mounten und wie eine normale Partition benutzen können (**mount /dev/ram1 /mnt/rootfs**) .

2.3.2 loopback-Device

Das loopback-Device ist ein Device, das es einem Anwender erlaubt, eine Datei wie ein Filesystem anzusprechen. Natürlich muss man diese Datei zuvor noch bearbeiten.

Auch das loopback-Device muss natürlich ebenfalls vom Kernel unterstützt werden. Dies kann man mit folgendem Befehl überprüfen:

```
ll /lib/modules/$(uname -r)/kernel/drivers/block/loop*
```

Natürlich kann es sein, dass dieser Befehl keine Ausgabe erzeugt, z.B. wenn man einen Kernel selbst kompiliert hat. Aber dann wissen wir ja, was wir einkompiliert haben, oder? Wenn nicht, kann man auch in der Datei *.config* nachschauen:

```
cat /usr/src/linux/.config — grep ^CONFIG_BLK_DEV_LOOP.
```

Ausgabe sollte ähnlich wie diese aussehen **CONFIG_BLK_DEV_LOOP=m** Danach muss man überprüfen, ob die loopback-Devices auch im **/dev** Verzeichnis existieren. **ll /dev/loop***

```
brw-r----- 1 root  root  7,  0 Aug  8 17:01 /dev/loop0
brw-r----- 1 root  root  7,  1 Aug  8 17:01 /dev/loop1
brw-r----- 1 root  root  7,  2 Aug  8 17:01 /dev/loop2
brw-r----- 1 root  root  7,  3 Aug  8 17:01 /dev/loop3
brw-r----- 1 root  root  7,  4 Aug  8 17:01 /dev/loop4
brw-r----- 1 root  root  7,  5 Aug  8 17:01 /dev/loop5
brw-r----- 1 root  root  7,  6 Aug  8 17:01 /dev/loop6
brw-r----- 1 root  root  7,  7 Aug  8 17:01 /dev/loop7
```

Es kann auch sein, dass diese Dateien nur Links auf ein Unterverzeichnis sind. Das ist völlig egal. Die Devices existieren trotzdem.

Wir haben uns also nun überzeugt, dass der Kernel loopback-Devices unterstützt und können uns nun das loopback-Device einrichten.

Zuerst erstellen wir eine leere Datei. Dies geschieht, wie immer, mit dem Befehl:

```
dd if=/dev/zero of=/mnt/initrd bs=1k count=4k
```

Somit haben wir eine 4 Megabyte große leere Datei angelegt. Als nächstes müssen wir

dieser Datei ein loopback-Device zuordnen: `losetup /dev/loop0 /mnt/initrd`. Somit ist nun die Datei `initrd` dem loopback-Device `/dev/loop0` zugeordnet. Nun muss man noch ein Filesystem “auf” dieser Datei anlegen: `mke2fs -m0 /dev/loop0`.

Nun kann man dieses Filesystem mit `mount -o loop -t ext2 /mnt/initrd /mnt/rootfs` in das Dateisystem einhängen und ganz normal Daten in das Verzeichnis `/mnt/rootfs` kopieren.

2.4 Kernel

Hier ist es nun an der Zeit, die Kernel-Quellen in ein beliebiges Verzeichnis auszupacken. Ich nehme an, dass die Quellen im `/ftp`-Verzeichnis liegen und ins Verzeichnis des Benutzer `root` ausgepackt werden sollen `cd /root && tar xvf /ftp/linux-2.4.18.tar.gz`. Um Anleitungskonform zu bleiben, erstelle ich nun einen Link von `/usr/src/linux` auf dieses neue Kernel-Verzeichnis `cd /usr/src && ln -s /root/linux linux`.

Es hat sich bisher allerdings bewährt, dass dieser Link *nicht* gesetzt wird. Denn diese Diskette hat im Normalfall nichts mit dem eigentlichen System zu tun. Nur um einen Kernel zu kompilieren muss das Kernelverzeichnis an keinem bestimmten Ort im Verzeichnisbaum sein. Wer darauf verzichtet, den Link zu setzen, muss dies im weiteren Verlauf der Anleitung allerdings berücksichtigen.

Es bietet sich auch an, dem Kernel eine spezielle Kennung zu geben (“Extraversion”). Diese Variable findet man im Makefile des Bootdisketten-Kernel (im folgenden nenne ich diesen Kernel ebenfalls so) in der vierten Zeile. Ich nenne meine Bootdisketten-Kernel immer *bootdisk* (EXTRAVERSION = -bootdisk) Der Vorteil ist einfach der, dass die Module für diesen Kernel in ein eigenes Unterverzeichnis kopiert werden, und somit nicht aus Versehen die Module das Kernel für den Host ersetzt werden.

2.5 uClibc und Busybox

Den Kernel hätten wir. Nun fehlen nur noch die beiden Pakete uClibc und busybox, dann haben wir ein lauffähiges Linux-System. Die Konfiguration dieser Pakete werden wir nun erledigen.

2.5.1 NFS

Wer NFS verwenden will, muss 3 Dinge beachten.

- In der busybox-Config.h muss man die Zeile

```
#define BB_FEATURE_NFSMOUNT
```

einkommentieren.

- In der uClibc Config muss man die Zeile

```
INCLUDE_RPC = true
```

einkommentieren.

- Wenn man keinen portmapper-Prozess am laufen hat, *muss* man ein Dateisystem mit der Option **-o nolock** mounten. Abhängig von der Kernel Version kann es auch sein, dass man noch zusätzlich die Version des verwendeten NFS-Server angeben muss.

Ein Kommando könnte zum Beispiel so aussehen:

```
mount -o nolock,nfsvers=2 nfserver:/home /home
```

2.5.2 uClibc einrichten und benutzen

Auch hier nehmen ich an, dass das uClibc-Paket im /ftp-Verzeichnis liegt. Nun wechseln wir wieder in das Hauptverzeichnis des Benutzer *root* und geben folgenden Befehl ein:

```
cat /ftp/uClibc-0.9.12.tar.bz2 | bunzip2 | tar xvf -
```

Danach wechseln wir in das, neue, Verzeichnis *uclibc*.

Im Unterverzeichnis **extra/Configs/** gibt es mehrer **Config** Dateien für verschiedene Rechnerarchitekturen.

Von diesen Dateien muss man diejenige auswählen, deren Dateierweiterung die benutzte Architektur beschreibt (also in meinem Fall *i386*). Diese Datei **Config.i386** muss man in das Hauptverzeichnis von uClibc (*/root/uclibc*) kopieren und in **Config** umbenennen.

Meine Einstellungen findet Ihr im Kapitel 7.1.2 auf Seite 28. Allzu viele Änderungen sollte man nicht vornehmen müssen.

Danach muss man die Programme und Bibliotheken mit **make** kompilieren und installieren.

Damit wäre nun die Entwicklungsumgebung installiert und man kann diese nutzen.

Dies erledigt man ganz einfach dadurch, dass man an der erste Stelle der Systempfadvariable *PATH* den Pfad zu den uClibc-Binaries einträgt (alternativ kann man auch in jedem Makefile die *cc*-Direktive auf den uClibc-gcc umbiegen). Der Befehl

```
export PATH=/usr/i386-linux-uclibc/bin:$PATH
```

ist wohl schneller eingetippt, als die Änderungen in den Makefiles.

Eines sollte man nun jedoch beachten: Man muss alle Kompilervorgänge in genau diesem Terminal ausführen, da nur in diesem Terminal der Pfad angepasst wurde.

Noch ein weiterer Punkt gilt es zu beachten: Die Bibliotheken müssen jedes mal neu erstellt werden, wenn man einen neueren Kernel-Tree einsetzt (also z.B. statt 2.2 den 2.4).

2.5.3 busybox konfigurieren und installieren

Und wieder nehmen ich an, dass das busybox-Paket im /ftp-Verzeichnis liegt. Nun wechseln wir wieder in das Hauptverzeichnis des Benutzer *root* und geben folgenden Befehl ein:

```
tar xvzf /ftp/busybox-0.60.2.tar.gz
```

Nun wechseln wir in das, neue, Verzeichnis */root/busybox-0.60.2*.

Danach muss man die Programme auswählen, die von busybox zur Verfügung gestellt werden sollen. Dies erledigt man in der Datei **Config.h**. Auch hierzu habe ich meine Konfiguration abgedruckt, im Kapitel 7.1.1 auf Seite 26.

Danach wird wie mit **make** compiliert. Die endgültige Installation wird später durchgeführt.

2.5.4 Syslinux installieren

Wird, wie üblich, mit **make** und **make install** installiert.

3 initrd erstellen

3.1 Ramdisk preparieren

3.1.1 Verzeichnisse erstellen

Die benötigten Unterverzeichnisse legt man mit dem Befehl

mkdir {dev,lib,proc,usr,var,sbin,root,bin,mount,cdrom,floppy,home,tmp,data,scripts}
an.

3.1.2 Devices und /etc erstellen

Im Unterverzeichnis **bootfloppy/** von busybox gibt es das Script *makedev.sh*. Es ist wohl einfacher, dieses Script mit **./mkdevs.sh /mnt/ramdisk/dev** aufzurufen als die ganzen Devices von Hand anzulegen.

Im selben Verzeichnis **bootfloppy/** gibt es auch das Verzeichnis **etc/**. Dieses enthält die wichtigsten Dateien. Kopieren kann man dieses Verzeichnis mit

cp -aR etc /mnt/ramdisk .

Es fehlen allerdings die Devices für das Diskettenlaufwerk und die SCSI-Devices. Ein Script für diese Devices findet Ihr im Kapitel 7.1.4 auf Seite 31.

3.1.3 busybox

Dies ist relativ einfach. Man muss einfach nur in das busybox-Quellverzeichnis wechseln und den Befehl **make PREFIX=/mnt/rootfs install**

3.1.4 uClibc runtime environment

Die für die Laufzeit notwendigen Dateien von uClibc werden mit dem Befehl

make PREFIX=/mnt/rootfs install_target auf die Ramdisk kopiert.

3.1.5 Bibliotheken bekannt machen

Damit die Programme auch wissen, welche Bibliotheken sie benutzen sollen, muss man ihnen diese bekannt machen. Dies geschieht mit dem Befehl

/usr/i386-linux-uclibc/bin/ldconfig -r /mnt/rootfs .

3.1.6 Erzeugen einer initial-Ramdisk

Eigentlich habe wir die ganze Zeit nichts anderes gemacht. Allerdings sind wir noch nicht ganz fertig. Die initial-Ramdisk sollte eigentlich ein gzip-File sein. Mit dem Befehl

dd if=/dev/ram1 bs=1k | gzip -v9 > rootfs.gz erstellt man ein solches File, das man dann in dieser Form auch auf die Bootdiskette kopieren kann. Aber dazu später.

Wenn man diese Datei wieder in eine Ramdisk extrahieren möchte, dann erreicht man dies mit dem Befehl **dd if=rootfs.gz bs=1k | gunzip > /dev/ram1** . Allerdings

können wir mit dieser Ramdisk noch keine Module laden. Dafür sollte man zuerst einen Kernel kompilieren.

Um aus einem loopback-Device eine initrd zu machen, muss man statt dem Device `/dev/ram1` das "Device" `initrd` auswählen. Ansonsten läuft alles gleich ab:

dd if=/mnt/initrd | gzip -v9 > rootfs.gz Das Extrahieren kann man sich sparen, da die Datei `initrd` ja immer noch existiert.

3.1.7 Kernel Kompilieren

Darauf möchte ich hier nicht weiter eingehen. Jeder, der selber eine Bootdiskette basteln möchte sollte zuerst einmal in der Lage sein, einen Kernel zu bauen. Ich gehe auch davon aus, dass der symbolische Link `/usr/src/linux` auf den Kernel zeigt, den man auf die Bootdiskette kopieren möchte.

Trotzdem sollte man eines ansprechen. Wenn man Module erzeugt, dann muss man diese ebenfalls auf die Bootdiskette kopieren. Es bietet sich hierfür das Verzeichnis `/lib/modules` an. Man kann, wenn man es unbedingt möchte, die ganzen Module in ein Unterverzeichnis `/lib/modules/"kernelversion"` kopieren.

Wichtig ist jedoch, dass man ein Dependencyfile anlegt, in dem die Abhängigkeiten der Module drinstehen. Dieses File erstellt man einfach mit dem Befehl

depmod -b /mnt/ramdisk -a 2.4.18 -F /usr/src/linux/System.map

3.2 Zusätzliche Programme

3.2.1 sfdisk

sfdisk wird einfach nur mit **make** kompiliert und anschließend in das `/bin` Verzeichnis der initial-Ramdisk kopiert.

3.2.2 Keyboard-map

Die keyboard-map ist eine Datei, die die Scancodes der Tastatur und die entsprechenden ASCII-Zeichen umwandelt.

Diese map-Datei kann mit **loadkmap < de.map** geladen werden.

Erstellt wird diese Datei, indem man auf dem Host-Rechner die Datei **dumpkmap > de.map** aus dem root-Filesystem ausführt.

Somit wird das Mapping der aktuellen Einstellung übernommen.

3.2.3 Kompilieren von Samba

Vom Samba-Paket wird zum Glück nur **smbmount** und **smbmnt** benötigt. Somit kann man das Paket wie üblich auspacken und mit **make smbmount** und **make smbmnt** diese Dateien erstellen.

3.2.4 strip

Nein, *strip* ist kein Befehl, der auf die Diskette mit draufkommt. Trotzdem ist er ein sehr wichtiger, wenn nicht sogar DER wichtigste, Befehl für das erstellen eines Diskettenlinux. Der Befehl *strip*, der auf jedem System vorhanden ist, sorgt dafür, dass überflüssige Symbole aus einem Binary-File entfernt werden. Der Platzgewinn ist teilweise enorm (z.B. mke2fs von über 300kB auf 70kB!). Und wie wir alle wissen: Platz auf einer Diskette ist kostbar ;-)

3.2.5 Kernel

Ich will hier kein Kernel-Tutorial schreiben. Deshalb nur meine Konfiguration (Kapitel 7.1.3, Seite 29) und elementare Optionen. Allerdings sind die Hardwaretreiber, natürlich, an meine Hardware angepasst.

3.2.6 Elementare Optionen

- **CONFIG_BLK_DEV_RAM=y**
(Block devices/Ram disk support)
Eine Ramdisk wird nur dann benutzt, wenn ich dies dem Kernel explizit mitteile. Da der Kernel zum Bootzeitpunkt unter Umständen noch kein root-Filesystem besitzt, kann er auch nicht auf Module zugreifen. Deshalb fest einkompilieren.
- **CONFIG_BLK_DEV_RAM_SIZE=4096**
(Block device/Default RAM disk size)
Hier wird die maximale Größe eines Ramdisk-Devices festgelegt. 4 MB sollten für eine initial Ramdisk genügen, da man mehr auch schwer mit dem Kernel zusammen auf einer Diskette unterbringen kann.
- **CONFIG_BLK_DEV_INITRD=y**
(Block devices/Initial RAM disk (initrd) support)
Mit dieser Option wird es erst ermöglicht, dass der Kernel eine initial-Ramdisk benutzen kann.
- **CONFIG_FAT_FS=y**
CONFIG_MSDOS_FS=y
(File systems/DOS FAT fs support)
(File systems/MSDOS fs support)
Auf der Bootdiskette liegen einige Scripte, die nach dem Booten gestartet werden müssen. Das Filesystem auf der Bootdiskette ist ein Standard FAT12. Deshalb muss man dem Kernel die Fähigkeit geben, ein FAT12 Dateisystem zu mounten.
- **CONFIG_NFS_FS=y**
(File systems/Network file systems/NFS file system support)
Der Kernel muss das NFS-Filesystem unterstützen, da die Linux-Image

per NFS exportiert werden. Diese Option sorgt dafür, dass der Kernel NFS unterstützt. Allerdings benötigt man immer noch Tools, die NFS beherrschen (was die auf der Diskette können).

4 Preparieren der Diskette

4.1 Die Größe

Um die Diskette sinnvoll nutzen zu können, muss sie etwas höher formatiert werden als normal. Die ist unter Linux aber wirklich kein Problem. Das Kommando **fdformat** hilft uns dabei. Mit diesem Befehl kann man ein Device formatieren. Wenn man sich im Verzeichnis **/dev** einmal alle Devices mit dem Prefix **fd0** anschaut `ls /dev/fd0*` sieht man, dass es mehr als nur das Device **fd0** gibt. Die anderen Devices haben als Anhang die Größe der jeweiligen Diskette. Ich benutze das Device **/dev/fd0u1680**. Somit ist die Diskette also knapp 1.7 MB groß, anstelle der 1.44 MB einer normalen Diskette.

In Zeiten von Gigabyte großen Festplatten mag dies lächerlich erscheinen - aber nochmal: Auf einer Diskette ist jedes Byte wertvoll. Wer sich länger mit Diskettenlinux (oder in letzter Konsequenz mit *embedded-Linux*) beschäftigt, wird irgendwann auch ein Byte-Hamsterer. Das kann ich jedem schriftlich geben ;-)

Mit dem Befehl `fdformat /dev/fd0u1680` wird, eine Diskette im ersten Diskettenlaufwerk vorrausgesetzt, die Diskette auf diese Größe formatiert.

Normalerweise sollte das **fdformat** die Daten für diese Diskette auf den Bildschirm schreiben. Bei dieser Diskettengröße sollte die Ausgabe etwa so aussehen:

Doppelseitig, 80 Spuren, 21 Sektoren/Spur, Totale Kapazität: 1680kB

Dies ist für den nächsten Schritt nicht ganz unerheblich.

4.2 Formatieren

Nun, nachdem die Diskette auf eine brauchbare Größe gebracht wurde, wird es Zeit das Filesystem anzulegen. Hierfür verwende ich das Standard DOS/FAT12. Dafür benötige ich den Befehl **mformat**. Sollte der auf einem System nicht vorhanden sein, dann vermutlich deshalb, weil das mtools-Paket nicht installiert ist. Am besten ihr installiert es mit den Linux-eigenen Paketmanagern (rpm/dpk, z.B. mtools*.rpm).

Da wir nicht mehr mit dem Standarddevice **fd0** arbeiten, sondern mit dem Device **fd0u1680**, müssen wir dies auch den mtools mitteilen. Dafür editiert man in der Datei **/etc/mtools.conf** am besten den Eintrag für das Laufwerk **b**. Ich denke nicht, dass noch jemand ein zweites Diskettenlaufwerk benutzt, oder? Der neuen Eintrag sollte in etwa so aussehen:

drive b: file="/dev/fd0u1680" exclusive. Nun kann die Diskette mit dem Laufwerksbuchstaben **b**: angesprochen werden.

Der Befehl um diese Diskette zu formatieren lautet somit wie folgt: `mformat -t 80 -n 21 b:`

4.3 Dateien übertragen

Nun sind wir im Besitz einer jungfräulichen Diskette. Es ist nun an der Zeit, zumindest die wichtigsten Dateien zu übertragen.

Für einen ersten Versuch ist dies eigentlich nur der Kernel, die `initrd` (initial Ramdisk) und die beiden Dateien von Syslinux (siehe 4.4).

Kopiert werden die Dateien jeweils mit dem **mcopy** Kommando.

`mcopy /usr/src/linux/arch/i386/boot/bzImage b:linux`

`mcopy initrd.gz b:`

4.4 Syslinux

Nun ist es an der Zeit, den Bootloader zu installieren. Das Interessante an Syslinux ist die Tatsache, dass eine normale DOS-Diskette benutzt wird. Somit kann man die Dateien (Kernel, initial-Ramdisk, ggf. zusätzliche Konfigurationsfiles) auf diese Diskette kopieren, wann immer man möchte. Würde man den Lilo verwenden, müsste man jedesmal den Bootsektor neu schreiben, wenn man den Kernel austauscht.

Die Konfiguration von Syslinux ist sehr einfach. Es genügt, wenn man folgendes in ein File **syslinux.cfg** einträgt und es mittels **mcopy** auf die Bootdiskette kopiert:

```
DISPLAY message.txt
DEFAULT linux
APPEND initrd=rootfs.gz root=/dev/ram0 init=/sbin/init
TIMEOUT 10
PROMPT 1
```

- **DISPLAY message.txt**
Diese Option bewirkt, dass vor dem Booten der Inhalt der Datei *message.txt* angezeigt wird. Diese Datei wird wie eine Textdatei behandelt, d.h. dass nur ASCII-Zeichen decodiert werden. Wenn diese Datei nicht vorhanden ist wird der Bootvorgang trotzdem fortgesetzt.
- **DEFAULT linux**
Mit dieser Option wird der Dateinamen des zu benutzenden Standardkernel festgelegt.
- **APPEND initrd=rootfs.gz root=/dev/ram0 init=/sbin/init**
APPEND ist eine Bezeichnung für die Optionen, die einem Kernel vor dem Booten übergeben werden können. Hier wird der Dateiname der *initrd*, das Device, in das die *initrd* ausgepackt wird und der Name, der Datei übergeben, die nach dem Booten ausgeführt wird.
- **TIMEOUT 10**
Gibt an, wieviele 1/10 s der Bootprompt auf eine Eingabe wartet. In diesem Fall 10/10 s, also 1 s.
- **PROMPT 1**
Wenn hier eine 1 steht, dann wird der Bootprompt angezeigt. Steht hier eine 0, dann nicht.

Danach noch den Befehl **syslinux -s /dev/fd0** absetzen und eine bootfähige Diskette wurde erstellt.

5 Startup und Skripte

Dieses Kapitel stellt Bootskripte dar, die ich für die ursprüngliche Version der Bootdiskette erstellt hatte. Die Skripte erheben keinen Anspruch auf Vollständigkeit, sauberen Code oder Sinnhaftigkeit ;-)

Diese Skripte sollen als Beispiele und Anregung dienen. Denn das ist der Teil, in dem sich jeder selbst austoben darf und alles das verpfuschen darf, was er sich bei der Installation auf dem Hauptrechner nicht traut ;-)

5.1 Grundlagen des Linuxstarts

Der Startvorgang von Linux geschieht immer nach dem gleichen Schema. Der Bootloader (syslinux, lilo oder grub) startet zuerst den Kernel. Wenn der Kernel vollständig geladen ist wird die `initrd` ausgepackt (wenn sie nicht schon vor dem Laden des Kernels ausgepackt wurde⁴). Nach dem Kernelstart wird die Datei `linuxrc` im Hauptverzeichnis der `initrd` ausgeführt. Sollte die Datei nicht vorhanden sein, wird der `init`-Prozess gestartet, dessen Kontrolldatei die allseits bekannte (sie ist doch bekannt, oder?) `/etc/inittab` ist. Wenn man Busybox installiert, ist `linuxrc` ein Link auf `/bin/busybox`. Im Prinzip kann `linuxrc` ein Shellscript oder eine binäre Datei sein. Wer ausführliche Informationen zum Bootprozess von Linux benötigt, sei auf das Tutotial von Werner Almesberger verwiesen.[1]

5.2 Startup der Diskette

Um die Aktionen auf der Diskette zu automatisieren muss während des Systemstarts eine Datei ausgeführt werden. Diese Datei befindet sich auf der Diskette und heisst `/etc/init.d/rcS`. Diese Datei wird automatisch beim Systemstart ausgeführt. Man kann auch eine andere Datei starten. Dies wird in der Datei `/etc/inittab` eingestellt. Die Datei, die beim Systemstart ausgeführt wird, steht in der Zeile mit dem Schlüsselwort **sysinit**.

5.2.1 rcS

Ich habe am Ende dieser Datei `/etc/init.d/rcS` die Zeile `./etc/startup.sh` eingefügt. Dieses Script erstellt ein paar Ramdisks für die Verzeichnisse `/tmp`, `/scripts` und `/data`. Diese Verzeichnisse müssen auch im Rootfilesystem angelegt werden.

Im Prinzip kann man zwar die Verzeichnisse anlegen, nachdem das Rootfilesystem geladen ist, doch manchmal kommt es vor, dass der Speicherplatz nicht ausreicht.

Man könnte dies auch alles in die Datei `/etc/init.d/rcS` eintragen. Ich bin allerdings eher ein Fan von mehreren Dateien. Anyway. Ich will hier niemandem etwas vorschreiben. ;-)

5.2.2 startup.sh

Die Datei **startup.sh** wird dazu verwendet, die Basisfunktionen der Linux-Umgebung zu initialisieren (Netzwerk und SCSI). Zusätzlich werden noch 3 Ramdisks angelgt.

⁴syslinux entpackt zuerst die `initrd` in den Hauptspeicher, bevor der Kernel geladen wird

Dann wird eine Datei **scripts.tgz** von der Diskette geladen und ausgepackt. In dieser Datei befinden sich mehrere Dateien. Unter anderem befindet sich ein Script Namens **postboot** in diesem File. Dieses wird am Ende von **startup.sh** ausgeführt und erlaubt es dem Anwender, nach dem Systemstart weitere Kommandos ablaufen zu lassen. Dies wird notwendig, da das Script **startup.sh** fest im Rootfilesystem eingebettet ist. Es wäre mühsam, für jede Änderung das Rootfilesystem auszupacken, die Änderung durchzuführen und dann das Rootfilesystem neu zu erstellen.

Nachdem die Datei **scripts.tgz** ausgepackt wurde, befindet sich auch die Datei **options** im Verzeichnis /scripts. In dieser Datei stehen die Netzwerkoptionen drin, die in **startup.sh** verwendet werden. Wenn die Optionen in **options** entsprechend gesetzt werden, dann kann man **startup.sh** auch in einen interaktiven Modus bringen.

Es werden die Module für die Netzwerkkarte und ggf. für das SCSI-System geladen. Danach wird die komplette Netzwerkkumgebung aufgesetzt (Hostname, Domain, DNS, Defaultgateway, IP-Adresse, Netmask).

Wenn das Netzwerk dann steht, kann man sich auch per NFS zu einem NFS-Server verbinden.

```
#!/bin/sh

##### init
# erstelle 4mb ramdisks fuer verzeichnisse
for i in 1 2 3; do
    dd if=/dev/zero of=/dev/ram$i count=4096 bs=1k
    mke2fs /dev/ram$i -m0 -i1400 4096
done
mount /dev/ram1 /tmp
mount /dev/ram2 /scripts
mount /dev/ram3 /data

# hier entpacke ich das scripts-archiv auf das system
mount -t msdos /dev/fd0u1680 /floppy
cd /
tar xvzpf /floppy/scripts.tgz
umount /floppy

# in der datei functions stehen wichtige functionen drin. wenn die nicht
# gestartet werden kann, dann wird das script beendet
if [ -f /etc/startup.functions ]; then
    . /etc/startup.functions
else
    exit 1
fi

# im scripts-archiv existiert eine datei options, in der die
# hardwareinformationen des rechners abgelegt werden
if [ -f /scripts/options ]; then
    . /scripts/options
else
    echo "Keine Optionendatei vorhanden."
    echo "Es ist sinnlos, das startup-Script weiter auszuführen."
    exit 2
fi

##### tastatur
# ggf. ein alternatives tastaturlayout laden
if [ "$KEYMAP" != "" -o "$KEYMAP" != " " ]; then
    loadkmap < $KEYMAP
fi

##### network
# wenn das netz gestartet werden soll, dann wird nach dem treibermodul
# fuer die netzwerkkarte gesucht. alternativ kann man dies auch von
# hand eingeben.
```

```

if [ $NETWORK -eq 1 ]; then
    if [ "$NETWORK_MODULE" != "kernel" ]; then
        if [ "$NETWORK_MODULE" = "" -o "$NETWORK_MODULE" = " " ]; then
            echo "Das Netzwerk wurde aktiviert, aber kein Modul wurde angegeben"
            echo "Bitte den Namen des Netzwerk-Moduls eingeben"
            echo -n "(return für fest im Kerneleinkompilierte Module)"
        read NETWORK_MODULE

        echo "Bitte Optionen für das Netzwerk-Modul eingeben"
        echo -n "(return für keine)"
    read NETWORK_OPTIONS

    if [ "$NETWORK_MODULE" != "" -o "$NETWORK_MODULE" != " " ]; then
        modprobe $NETWORK_MODULE $NETWORK_OPTIONS
    fi
    else
        modprobe $NETWORK_MODULE $NETWORK_OPTIONS
    fi
    fi

    ifconfig $NETWORK_STRING $IP_ADDR netmask $NETMASK

    if [ "$DEFAULT_GW" != "" -o "$DEFAULT_GW" != " " ]; then
route add default gw $DEFAULT_GW
fi
    if [ "$DNS_SEARCHSTRING" != "" -o "$DNS_SEARCHSTRING" != " " ]; then
echo "search $DNS_SEARCHSTRING" > /etc/resolv.conf
fi
    if [ "$DNS_SERVER" != "" -o "$DNS_SERVER" != " " ]; then
echo "nameserver $DNS_SERVER" >> /etc/resolv.conf
fi
    if [ "$HOSTNAME" != "" -o "$HOSTNAME" != " " ]; then
hostname $HOSTNAME
fi
fi

##### scsi #####
if [ $SCSI -eq 1 ]; then
    if [ "$SCSI_MODULE" != "kernel" ]; then
        if [ "$SCSI_MODULE" = "" -o "$SCSI_MODULE" = " " ]; then
            echo "SCSI wurde zwar aktiviert, aber ohne Modul"
            echo "Name des SCSI-Kernelmoduls eingeben"
            echo -n "(return für fest im Kerneleinkompilierte Module)"
        read SCSI_MODULE

        echo "Optionen für das SCSI-Kernelmoduls eingeben"
        echo -n "(return für keine)"; read SCSI_OPTIONS
        if [ "$SCSI_MODULE" != "" -o "$SCSI_MODULE" != " " ]; then
            start_scsi $SCSI_MODULE $SCSI_OPTIONS
        fi
    else
        start_scsi $SCSI_MODULE $SCSI_OPTIONS
    fi
    fi

fi

##### nfs verbindung
if [ $NFS -eq 1 ]; then
    if [ "$NFS_SERVER" = "" -o "$NFS_SERVER" = " " ]; then
        echo "NFS nicht aktiviert"
    elif [ "$NFS_SERVER" = "ask" ]; then
        echo "NFS wurde zwar aktiviert, jedoch wird manuelle Eingabe erwünscht:"
        echo -n "Bitte die IP-Adresse des NFS-server eingeben      : "
    read NFS_SERVER
        echo -n "Bitte Name NFS-Share auf dem NFS-Server eingeben : "
    read NFS_SHARE
        echo -n "Bitte den lokalen mountpunkt eingeben      : "
    read NFS_DIRECTORY
        mount -o nolock,nfsvers=2 $NFS_SERVER:$NFS_SHARE $NFS_DIRECTORY
    else
        mount -o nolock,nfsvers=2 $NFS_SERVER:$NFS_SHARE $NFS_DIRECTORY
    fi
fi

##### das folgende script kann für zusätzliche kommandos genutzt werden
if [ -f /scripts/postboot ]; then
    . /scripts/postboot

```

```
fi
```

5.2.3 options

In Options können die Einstellungen für die Module und das Netzwerk vorgenommen werden. Die einzelnen Variablen sind, denke ich, ziemlich eindeutig und werden kurz beschrieben.

```
#!/bin/sh

# format VARIABLE=WERT
# Wert bei boolean: 0 für nein und 1 für ja und OHNE
# Anführungszeichen!!
#
# Bei variabelentyp string oder option wird ein
# string verlangt.
# mit Anführungszeichen!

# bei folgenden Module kann man die Option "kernel"
# (case-sensitive!) angeben, wenn man keine module
# für den treiber erstellt hat:
# SCSI_MODULE und NETWORK_MODULE

#####
##### HARDWARE #####
#####

##### Tastaturtabelle #####
# tastaturtabelle (string)
# bitte den GANZEN ABSOLUTEN pfad angeben!
KEYMAP="/etc/de.map"

##### NETWORK #####
# netzwerk starten (boolean)
# 1 heißt ja, 0 das Gegenteil
NETWORK=1
# name des nic-module (string)
NETWORK_MODULE="3c59x"
# nic-options (option)
# hier kann man optionale optionen angeben
# im normalfall wird dies nicht benötigt
NETWORK_OPTIONS=""

##### SCSI #####
# start scsi (boolean)
SCSI=1
# scsi-module to load (string)
SCSI_MODULE="aic7xxx"
# options for scsi (option)
SCSI_OPTIONS=""

##### nfs #####
# wenn NFS auf 1 gesetzt wird, dann wird NFS
# benutzt. 0 ist das Gegenteil. (boolean)
NFS=1
# NFS-Server (string)
# wenn ihr hier "ask" (case-sensitive) eingibt,
# wird während dem booten nach der einstellung
# gefragt
NFS_SERVER="192.168.10.1"
# verzeichnis auf dem NFS-server (string)
NFS_SHARE="/home/share"
# lokaler nfs-mountpoint (string)
NFS_DIRECTORY="/home"

#####
##### SYSTEM #####
#####
```

```

# ip-adresse des rechners (string)
IP_ADDR="192.168.10.10"
# Netzmaske für dieses System (string)
NETMASK="255.255.255.0"
# Default Gateway für dieses System
DEFAULT_GW="192.168.10.1"
# DNS-Server für dieses System (string)
DNS_SERVER="192.168.10.1"
# Searchstring für den DNS-Server (string)
DNS_SEARCHSTRING="example.netz"
# NETWORK_STRING ist die nic, auf die alle
# einstellungen "gemappt" werden... (string)
NETWORK_STRING="eth0"
# hostname (string)
HOSTNAME="client"

```

5.2.4 postboot

Mit **postboot** ist es möglich, nach dem Start einige Kommandos ablaufen zu lassen (ähnlich *rc.local*). Man kann auch andere Scripte aufrufen. Das Script ist hauptsächlich als erweiterung gedacht, und muss nicht vorhanden sein, damit *startup.sh* fehlerfrei funktioniert.

```

#!/bin/ash

echo "Alle verfügbaren Kommandos: "
ls -al /bin

echo "Rufe ich halt nochmal startup.sh auf: "
. /scripts/startup.sh
echo "die letzte Aktion war dämlich ;-)"

```

5.2.5 scripts.tgz

scripts.tgz ist ein gzipped-tar-Archiv des Verzeichnisses */script*, das sich irgendwo auf der Platte befinden kann. In diesem Verzeichnis muss sich auf jeden Fall die Datei **options** befinden. Zusätzlich kann noch die Datei **postboot** in diesem Verzeichnis sein. Diese beiden Dateien werden vom Script *startup.sh* aufgerufen und müssen ausführbare Dateien sein (x-Rechte gesetzt). Alle anderen Dateien werden von **startup.sh** ignoriert.

Möchte man andere Dateien ausführen, so kann man dies über das Script **postboot** erledigen. Um andere Scripte ausführen zu können muss man wissen, dass sich das Script auf der Diskette im Pfad *"/scripts"* befindet. Nun kann man relativ zu diesem Pfad arbeiten (oder gleich absolut).

Der Inhalt eines solchen Archives kann so aussehen (**tar tvzf scripts.tgz**):

```

drwxr-xr-x root/root          0 2002-01-24 13:35:17 scripts/
-rwxr-xr-x root/root       2285 2002-01-24 13:35:00 scripts/options
-rwxr-xr-x root/root        352 2002-01-21 14:54:54 scripts/postboot

```

Man sieht, auf jeden Fall muss das Verzeichnis "scripts" vor den Dateinamen stehen. Nur dann werden diese Dateien auch an die richtige Stelle auf der Diskette ausgepackt. Ausserdem muss die Erweiterung des Archiv ".tgz" sein.

Was besonders wichtig ist: **Die Größe der Scripte darf niemals über 4 MB steigen, da sonst die Ramdisk zu klein ist!**

5.3 Besonderheiten für die Scriptprogrammierung

5.3.1 Erklärung

Die auf den Bootdisketten benutzten Programme entsprechen weitestgehend den standard GNU-Tools, die bei GNU/Linux zum Einsatz kommen.

Jedoch mussten die Programmierer Zugeständnisse machen. So bricht z.B. selten ein Programm ohne hässlichen segfault (segmentation fault) ab. Dies ist ziemlich ungünstig für die Fehlersuche. Auch können manche Befehle nicht den kompletten Optionensatz ihrer GNU-Brüder und Schwestern (z.B. grep!). Deshalb habe ich hier eine kleine Tabelle mit Besonderheiten, die mir während der Scriptprogrammierung aufgefallen sind. Die Liste könnte vermutlich täglich erweitert werden.

5.3.2 Übersicht

GNU-Programme	Busybox-Programme
head -1	head -n 1
tail -1	tail -n 1
cp /dev/null file	rm -f file; touch file
grep -v -x “*” (leere Zeilen)	grep [A-Za-z]
funktion fname {	fname {

5.4 Fehlermeldungen

- `init not found`
Dies kann zwei Ursachen haben. Zum einen könnten die Rechte falsch gesetzt sein (zumindest das `x`-Bit muss gesetzt sein). Zum anderen könnte die `init` gegen die falschen Bibliotheken gelinkt worden sein (z.B. auf die Systembibliotheken anstatt auf die `uClibc`-Bibliotheken).

6 Arbeiten mit der Bootdiskette

6.1 Besonderheiten der Bootdiskette

6.1.1 Mounten per NFS

Auf der Bootdiskette sind nicht alle Pakete installiert, die auf einem “normalen” Linux normalerweise installiert werden. Unter anderem fehlt ein Portmapper.

Somit kann man die NFS-Verbindung nicht “locken”. Was im allgemeinen keine Probleme nach sich ziehen sollte (und in der Betatestphase auch nicht tat).

Somit muss man das mount-Kommando mit der Option “nolock” ausführen. Also z.B. so **mount -t nfs -o nolock,nfsvers=2 192.168.1.1:/nfsshare /mnt**

6.1.2 Mounten per SMB

SMB ist etwas seltsam zu bedienen, wenn man nur die angenehmen Scripte der neueren Versionen kennt (ja, es geht in der Tat noch schlimmer ;-)).

Ein Kommando, um sich zu einem NFS-Verzeichnis zu verbinden sieht folgendermaßen aus:

smbmount //server/share Passwort -U benutzername -I 192.168.10.1 -c “mount /home”

alles klar? Gut, dann eben mit Erklärung ;-)

- **smbmount** ist der Befehl, der die Verbindung zum smb-Server herstellt.
- **//server/share** ist die Freigabe, bestehend aus dem Freigabename *share* und dem smb-Server *server*. Der Name des smb-Server darf **nicht** als IP-Adresse angegeben werden. Dafür gibt es eine andere Option.
- **Passwort** ist das Passwort für die Freigabe - im Klartext
- **-U benutzername** ist der Benutzername für die Freigabe
- **-I 192.168.10.1** ist die IP-Adresse des smb-server. Wenn die nmb-Namensauflösung nicht korrekt funktioniert, dann muss man die IP-Adresse mit angeben.
- **-c “mount /home”** ist die wichtigste Option, wenn man die Freigabe sinnvoll benutzen möchte. Diese Option bedeutet nichts anderes, als dass das Kommando *mount* ausgeführt wird, und die Freigabe an das Verzeichnis */home* anhängt. Ohne diese Option würden man nur eine Kommandozeilenshell auf dem smb-Server bekommen. Nicht eben prickelnd, wenn man bequem Daten übertragen möchte ;-)

Man sollte nicht unerwähnt lassen, dass SMB einige seltsame Eigenschaften während des Testens an den Tag legte. So wurde einmal eine Datei von einem Windowsserver per SMB heruntergeladen und danach wieder hochgeladen. Die Datei war danach einige Bytes kleiner, aber immer noch ausführbar. Soll heißen: So ganz würde ich SMB nicht trauen. Ich konnte auch nicht testen, ob die von mir benutzte Version moch mit einem XP-Server kommunizieren kann. Bis einschließlich Windows 2000 hat diese Version jedoch funktioniert.

7 Anhang

7.1 Einstellungen in den config-Files

Ich habe hier nur die Zeilen aufgelistet, die ich einkommentiert habe. Meiner Meinung nach ist es wenig sinnvoll, die komplette Config-Datei hier abzdrukken, da die restlichen Angaben nicht editiert werden sollten.

7.1.1 Einstellungen in der busybox-Config

```
#define BB_PIDOF
#define BB_PING
#define BB_PIVOT_ROOT
#define BB_POWEROFF
#define BB_PRINTF
#define BB_PS
#define BB_PWD
#define BB_REBOOT
#define BB_RESET
#define BB_RM
#define BB_RMDIR
#define BB_RMMOD
#define BB_ROUTE
#define BB_SED
#define BB_SLEEP
#define BB_SORT
#define BB_SWAPONOFF
#define BB_SYNC
#define BB_SYSLOGD
#define BB_TAIL
#define BB_TAR
#define BB_TEE
#define BB_TEST
#define BB_TELNET
#define BB_TFTP
#define BB_TOUCH
#define BB_TRACEROUTE
#define BB_TRUE_FALSE
#define BB_TTY
#define BB_UNIX2DOS
#define BB_UMOUNT
#define BB_UNAME
#define BB_UPTIME
#define BB_VI
#define BB_WC
#define BB_WGET
#define BB_WHICH
```

```
#define BB_WHOAMI
#define BB_XARGS
#define BB_YES
#define BB_FEATURE_SH_IS_ASH
#define BB_FEATURE_AUTOWIDTH
#define BB_FEATURE_LS_USERNAME
#define BB_FEATURE_LS_TIMESTAMPS
#define BB_FEATURE_LS_FILETYPES
#define BB_FEATURE_LS_SORTFILES
#define BB_FEATURE_LS_RECURSIVE
#define BB_FEATURE_LS_FOLLOWLINKS
#define BB_FEATURE_FANCY_PING
#define BB_FEATURE_USE_INITTAB
#define BB_FEATURE_LINUXRC
#define BB_FEATURE_REMOTE_LOG
#define BB_FEATURE_FANCY_TAIL
#define BB_FEATURE_MOUNT_LOOP
#define BB_FEATURE_NFSMOUNT
#define BB_FEATURE_MOUNT_FORCE
#define BB_FEATURE_TAR_CREATE
#define BB_FEATURE_TAR_EXCLUDE
#define BB_FEATURE_TAR_GZIP
#define BB_FEATURE_SORT_REVERSE
#define BB_FEATURE_SORT_UNIQUE
#define BB_FEATURE_COMMAND_EDITING
#define BB_FEATURE_COMMAND_TAB_COMPLETION
#define BB_FEATURE_IFCONFIG_STATUS
#define BB_FEATURE_IFCONFIG_SLIP
#define BB_FEATURE_IFCONFIG_MEMSTART_IOADDR_IRQ
#define BB_FEATURE_IFCONFIG_HW
#define BB_FEATURE_IFCONFIG_BROADCAST_PLUS
#define BB_FEATURE_WGET_STATUSBAR
#define BB_FEATURE_WGET_AUTHENTICATION
#define BB_FEATURE_HUMAN_READABLE
#define BB_FEATURE_FIND_TYPE
#define BB_FEATURE_FIND_PERM
#define BB_FEATURE_FIND_MTIME
#define BB_FEATURE_TFTP_PUT
#define BB_FEATURE_TFTP_GET
#define BB_FEATURE_VI_COLON
#define BB_FEATURE_VI_YANKMARK
#define BB_FEATURE_VI_SEARCH
#define BB_FEATURE_VI_USE_SIGNALS
#define BB_FEATURE_VI_DOT_CMD
#define BB_FEATURE_VI_READONLY
#define BB_FEATURE_VI_SETOPTS
#define BB_FEATURE_VI_SET
#define BB_FEATURE_VI_WIN_RESIZE
```

```
#define BB_FEATURE_TELNET_TTYPE
```

7.1.2 Einstellungen in der uClibc-Config

```
TARGET_ARCH=i386
NATIVE_CC = gcc
CC = $(CROSS)gcc
AR = $(CROSS)ar
LD = $(CROSS)ld
NM = $(CROSS)nm
STRIPTOOL = $(CROSS)strip
DODEBUG = false
WARNINGS=-Wall
KERNEL_SOURCE=/usr/src/linux
HAS_MMU = true
HAS_FLOATING_POINT = true
HAS_LIBM_FLOAT = true
HAS_LIBM_DOUBLE = true
HAS_LIBM_LONG_DOUBLE = false
HAS_LONG_LONG = true
HAS_LOCALE = false
LOCALE_DIR = "/usr/share/uClibc-locale/"
MALLOC = malloc-930716
UNIFIED_SYSCALL = false
DOLFS = false
INCLUDE_RPC = true
INCLUDE_IPV6 = false
DOPIC = false
HAVE_SHARED = true
BUILD_UCLIBC_LDSO=true
SHARED_LIB_LOADER_PATH=$(DEVEL_PREFIX)/lib
DEVEL_PREFIX = /usr/$(TARGET_ARCH)-linux-uclibc
SYSTEM_DEVEL_PREFIX = $(DEVEL_PREFIX)/usr
PREFIX =
```

Am Makefile ldso/utills/Makefile habe ich auch Änderungen vorgenommen, damit die Binary **ldconfig** erstellt wird.

```
TOPDIR=../..
include $(TOPDIR)Rules.mak

all: ldconfig ldd readelf

readsoname.o: readsoname.c readsoname2.c
    $(TARGET_CC) $(TARGET_CFLAGS) -c $< -o $@
    $(STRIPTOOL) -x -R .note -R .comment $*.o

ldconfig.o: ldconfig.c
```

```

$(TARGET_CC) $(TARGET_CFLAGS) -DUCLIBC_TARGET_PREFIX="\$(TARGET_PREFIX)\
    -c $< -o $@
$(STRIPTOOL) -x -R .note -R .comment $*.o

readelf: readelf.c
$(TARGET_CC) $(TARGET_CFLAGS) -static readelf.c -o $@
$(STRIPTOOL) -x -R .note -R .comment $@

ifeq ($(strip $(LIBRARY_CACHE)),)
ldconfig:
    echo "LIBRARY_CACHE disabled -- not building ldconfig"
else

```

Hier die Änderungen:

```

ldconfig: ldconfig.o readsoname.o
$(TARGET_CC) $(TARGET_CFLAGS) -static $^ -o $@
$(STRIPTOOL) -x -R .note -R .comment $@

```

That's it...

```

endif

ldd: ldd.c
$(TARGET_CC) $(TARGET_CFLAGS) -DUCLIBC_TARGET_PREFIX="\$(TARGET_PREFIX)\
    -DUCLIBC_DEVEL_PREFIX="\$(DEVEL_PREFIX)" \
    -DUCLIBC_BUILD_DIR="\$(shell cd $(TOPDIR); pwd)" \
    -DUCLIBC_LDSO="\$(UCLIBC_LDSO)" \
    -static ldd.c -o $@
$(STRIPTOOL) -x -R .note -R .comment $@

clean:
    rm -f ldconfig ldd readelf *.o *~ core

```

7.1.3 Einstellungen in der Kernel-Config

VORSICHT: Einstellungen für Kernel 2.4!

```

CONFIG_X86=y
CONFIG_ISA=y
CONFIG_UID16=y
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES=y
CONFIG_M586TSC=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_CMPXCHG=y
CONFIG_X86_XADD=y

```

```
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_RWSEM_XCHGADD_ALGORITHM=y
CONFIG_X86_L1_CACHE_SHIFT=5
CONFIG_X86_USE_STRING_486=y
CONFIG_X86_ALIGNMENT_16=y
CONFIG_X86_TSC=y
CONFIG_X86_PPRO_FENCE=y
CONFIG_HIGHMEM4G=y
CONFIG_HIGHMEM=y
CONFIG_SMP=y
CONFIG_HAVE_DEC_LOCK=y
CONFIG_NET=y
CONFIG_X86_IO_APIC=y
CONFIG_X86_LOCAL_APIC=y
CONFIG_PCI=y
CONFIG_PCI_GOANY=y
CONFIG_PCI_BIOS=y
CONFIG_PCI_DIRECT=y
CONFIG_SYSVIPC=y
CONFIG_SYSCTL=y
CONFIG_KCORE_ELF=y
CONFIG_BINFMT_AOUT=y
CONFIG_BINFMT_ELF=y
CONFIG_BINFMT_MISC=y
CONFIG_BLK_DEV_FD=y
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_SIZE=4096
CONFIG_BLK_DEV_INITRD=y
CONFIG_PACKET=y
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_IDE=y
CONFIG_BLK_DEV_IDE=y
CONFIG_BLK_DEV_IDEDISK=y
CONFIG_IDEDISK_MULTI_MODE=y
CONFIG_BLK_DEV_IDECD=y
CONFIG_BLK_DEV_IDEPCI=y
CONFIG_IDEPCI_SHARE_IRQ=y
CONFIG_BLK_DEV_IDEDMA_PCI=y
CONFIG_BLK_DEV_ADMA=y
CONFIG_BLK_DEV_IDEDMA=y
CONFIG_SCSI=m
CONFIG_BLK_DEV_SD=m
CONFIG_SD_EXTRA_DEVS=40
CONFIG_BLK_DEV_SR=m
CONFIG_SR_EXTRA_DEVS=2
```

```
CONFIG_SCSI_AIC7XXX=m
CONFIG_AIC7XXX_CMDS_PER_DEVICE=253
CONFIG_AIC7XXX_RESET_DELAY_MS=15000
CONFIG_NETDEVICES=y
CONFIG_DUMMY=m
CONFIG_NET_ETHERNET=y
CONFIG_NET_VENDOR_3COM=y
CONFIG_EL3=m
CONFIG_VORTEX=m
CONFIG_NET_PCI=y
CONFIG_VIA_RHINE=m
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_ISO9660_FS=y
CONFIG_JOLIET=y
CONFIG_MINIX_FS=y
CONFIG_PROC_FS=y
CONFIG_EXT2_FS=y
CONFIG_NFS_FS=y
CONFIG_SUNRPC=y
CONFIG_LOCKD=y
CONFIG_MSDOS_PARTITION=y
CONFIG_NLS=y
CONFIG_NLS_DEFAULT="iso8859-1"
CONFIG_NLS_CODEPAGE_850=y
CONFIG_NLS_ISO8859_1=y
CONFIG_VGA_CONSOLE=y
```

Es handelt sich bei dieser Konfiguration weder um eine besonders optimierte Version, noch handle ich nach den Grundsätzen der Kernelerstellung (kleiner Kernel und viele Module). Allerdings erscheint es mir sinnvoll, für eine überschaubare Anzahl verschiedener Hardwarekomponenten, einen monolithischen Kernel zu bauen. Dadurch wollte ich einfach Ärger mit fehlenden Modulen, fehlenden Modulabhängigkeiten oder fehlerhaften Modulkonfigurationen vermeiden.

Es macht halt keinen Spass, wenn man vor einem Rechner sitzt, die Bootdiskette wunderbar funktioniert, man aber ein wesentliches SCSI-Modul vergessen hat.

7.1.4 makedev

Dieses Script legt die fehlenden Devices für SCSI und Floppy an. Abspeichern unter beliebigem Namen, in das *dev*/-Verzeichnis des root-Filesystems kopieren und ausführen. Unter Mandrake/RedHat gibt es allerdings den tollen Befehl *MAKEDEV*. Es lohnt sich, diesen sich genauer anzuschauen, da man sich das Script dann sparen kann ;-)

```
#!/bin/sh
```

```
echo "creating fd0*-devices"
mknod -m 660 fd0d360 b 2 4
mknod -m 660 fd0h360 b 2 20
mknod -m 660 fd0h410 b 2 48
mknod -m 660 fd0h420 b 2 64
mknod -m 660 fd0h720 b 2 24
mknod -m 660 fd0h880 b 2 80
mknod -m 660 fd0h1200 b 2 8
mknod -m 660 fd0h1440 b 2 40
mknod -m 660 fd0h1476 b 2 56
mknod -m 660 fd0h1494 b 2 72
mknod -m 660 fd0h1660 b 2 92
mknod -m 660 fd0u360 b 2 12
mknod -m 660 fd0u720 b 2 16
mknod -m 660 fd0u800 b 2 120
mknod -m 660 fd0u820 b 2 52
mknod -m 660 fd0u830 b 2 68
mknod -m 660 fd0u1040 b 2 84
mknod -m 660 fd0u1120 b 2 88
mknod -m 660 fd0u1440 b 2 28
mknod -m 660 fd0u1660 b 2 124
mknod -m 660 fd0u1680 b 2 44
mknod -m 660 fd0u1722 b 2 60
mknod -m 660 fd0u1743 b 2 76
mknod -m 660 fd0u1760 b 2 96
mknod -m 660 fd0u1840 b 2 116
mknod -m 660 fd0u1920 b 2 100
mknod -m 660 fd0u2880 b 2 32
mknod -m 660 fd0u3200 b 2 104
mknod -m 660 fd0u3520 b 2 108
mknod -m 660 fd0u3840 b 2 112
mknod -m 660 fd0CompaQ b 2 4
mknod -m 660 fd0D360 b 2 12
mknod -m 660 fd0D720 b 2 16
mknod -m 660 fd0H1440 b 2 28
mknod -m 660 fd0H360 b 2 12
mknod -m 660 fd0H720 b 2 16

#! /bin/sh
# erstellt scsi devices
# eingabe: ABSOLUTER pfad, in dem die devices
# erstellt werden sollen.
# wenn kein pfad angegeben wird, dann wird der
# aktuelle pfad benutzt

if [ $# -eq 0 ]; then
    pfad=.
else
    pfad=$1
```

```
fi

count=0
for i in sda sdb sdc sdd sde sdf sdg sdh; do
    for j in $(seq 0 15); do
        if [ $j -eq 0 ]; then j=""; fi
        mknod -m 660 $pfad/$i$j b 8 $count
        let count="count + 1"
    done
done
```

Literatur

- [1] Almesberger, Werner. *Booting Linux: The history and the future*
<ftp://icaftp.epfl.ch/pub/people/almesber/booting/bootinglinux-current.ps.gz>
- [2] Andersen, Erik
\$BUSYBOX-DIRECTORY/bootfloppy/bootfloppy.txt

Index

Begriffe

- initrd

 - initial Ramdisk, 16

 - initrd, 16, 17

fdformat, 16

mtools

- mcopy, 16, 17

- mformat, 16

Programme

- busybox, 5, 12

- e2fsprogs, 6

- Kernel, 6

- lilo, 6

- Samba, 6

- syslinux, 5, 17

- uClibc, 5