

# **fli4l – Entwickler Dokumentation**

## **Version 4.0.0-testing-sunxi-r60742**

Frank Meyer

E-Mail: [frank@fli4l.de](mailto:frank@fli4l.de)

Das fli4l-Team

E-Mail: [team@fli4l.de](mailto:team@fli4l.de)

7. September 2022

# Inhaltsverzeichnis

<b>1</b>	<b>Entwickler-Dokumentation</b>	<b>4</b>
1.1	Allgemeine Regeln	4
1.2	Übersetzen von Programmen	5
1.3	Modulkonzept	5
1.3.1	mkfli4l	5
1.3.2	Aufbau	6
1.3.3	Die Konfiguration der Pakete	7
1.3.4	Die Liste der zu kopierenden Dateien	7
1.3.5	Die Prüfung der Konfiguration-Variablen	12
1.3.6	Eigene Definitionen zum Prüfen der Konfigurationsvariablen	14
1.3.7	Erweiterte Prüfungen der Konfiguration	20
1.3.8	Unterstützung verschiedener Kernelversionslinien	36
1.3.9	Dokumentation	36
1.3.10	Dateiformate	38
1.3.11	Entwickler-Dokumentation	38
1.3.12	Client-Programme	38
1.3.13	Quellcode	38
1.3.14	Weitere Dateien	39
1.4	Allgemeine Skript-Erstellung auf fli4l	39
1.4.1	Aufbau	39
1.4.2	Umgang mit Konfigurationsvariablen	40
1.4.3	Persistente Speicherung von Daten	40
1.4.4	Fehlersuche	41
1.4.5	Hinweise	42
1.5	Arbeit mit dem Paketfilter	43
1.5.1	Hinzufügen von eigenen Ketten und Regeln	43
1.5.2	Einordnen in bestehende Regeln	44
1.5.3	Erweiterung der Paketfilter-Tests	45
1.6	CGI-Erstellung für das <i>httpd</i> -Paket	46
1.6.1	Allgemeines zum Webserver	46
1.6.2	Skriptnamen	46
1.6.3	Menü-Einträge	46
1.6.4	Aufbau eines CGI-Skriptes	48
1.6.5	Sonstiges	53
1.6.6	Fehlersuche	53
1.7	Hochfahren, Herunterfahren, Einwählen und Auflegen unter fli4l	54
1.7.1	Bootkonzept	54
1.7.2	Start- und Stopp-Skripte	54
1.7.3	Hilfsfunktionen	57
1.7.4	mdev-Regeln	59

## Inhaltsverzeichnis

1.7.5	ttyI-Geräte	62
1.7.6	Skripte beim Einwählen und Auflegen	62
1.8	Paket „template“	63
1.9	Aufbau des Boot-Datenträgers	64
1.10	Konfigurationsdateien	64
1.10.1	Konfiguration Provider	65
1.10.2	Konfiguration DNS	65
1.10.3	Hosts-Datei	66
1.10.4	imond-Konfiguration	66
1.10.5	Die /etc/.profile-Datei	66
1.10.6	Skripte in /etc/profile.d/	67
1.11	Inkompatibilitäten zwischen 3.x und 4.x	67
<b>Abbildungsverzeichnis</b>		<b>68</b>
<b>Tabellenverzeichnis</b>		<b>69</b>
<b>Index</b>		<b>70</b>

# 1 Entwickler-Dokumentation

Dieses Kapitel führt in die Aspekte von fli4l ein, die hauptsächlich für jene interessant sind, die mit dem Gedanken spielen, fli4l durch die Entwicklung eigener Pakete und OPTs zu erweitern. Es wird erläutert, wie fli4l „unter der Haube“ arbeitet und wie man in gewisse Prozesse eingreifen kann, um die Funktionalität von fli4l zu verändern oder zu verbessern.

## 1.1 Allgemeine Regeln

Damit ein neues Paket in die OPT-Datenbank auf der fli4l-Homepage aufgenommen wird, müssen einige Regeln beachtet werden. Pakete, die sich nicht an diese Regeln halten, können ohne Vorwarnung aus der Datenbank entfernt werden.

1. *Keine* Kopieraktionen von Seiten des Benutzers! fli4l bietet ein ausgefeiltes System, um die Daten der fli4l-Pakete in die Installationsarchive einzupacken. Alle Dateien, die auf den Router sollen, liegen im Verzeichnis `opt/`.
2. Pakete richtig packen und komprimieren: Die Pakete müssen so aufgebaut sein, dass sie sich mühelos ins fli4l-Verzeichnis entpacken lassen.
3. Die Pakete sollen sich *vollständig* über die Konfigurationsdatei konfigurieren lassen. Ein weiteres Bearbeiten der Konfigurationsdateien darf nicht vom Benutzer verlangt werden. Schwierige Entscheidungen dem Benutzer abnehmen oder in einen erweiterten Bereich verlagern (ans Ende der Konfigurationsdatei mit einem dicken Hinweis, etwa: ONLY MODIFY IF YOU KNOW WHAT YOU DO).
4. Noch ein Hinweis zur Konfigurationsdatei: Anhand des Namens einer Variablen muss sich eindeutig erkennen lassen, zu welchem OPT sie gehört. So gehören z. B. zum `OPT_HTTPD` die Variablen `OPT_HTTPD`, `HTTPD_USER_N`, usw.
5. Bitte, bitte, macht möglichst kleine Binaries (Programme)! Das passiert automatisch, wenn ihr das FBR aus dem *src*-Paket verwendet. Denkt auch daran, unnötige Features zu deaktivieren. Bei vorkompilierten Programmen hilft u. U. ein

```
strip -R .comment -R .note <Dateiname>
```

weiter. Nichts ist frustrierender als ein Download von 2 MiB, wenn man nachher feststellt, dass 500 kiB gereicht hätten...

**Wichtig:** *Bitte auf diese Art und Weise keine Kernel-Module behandeln! Sie werden hinterher nicht mehr funktionieren!*

6. Prüft euer Copyright! Wenn ihr Dateien als Vorlagen benutzt, achtet bitte darauf, das Copyright entsprechend zu ändern. Dies gilt besonders für die Config-, Check- und Opt-Textdateien. Ersetzt hier das Copyright durch euren eigenen Namen. Bei wortwörtlich kopierter Dokumentation muss natürlich das Copyright des Original-Autors erhalten bleiben!
7. Bitte als Archivtypen nur verbreitete, freie Formate benutzen. Dazu gehören:
  - ZIP (`.zip`)
  - GZIP (`.tgz` oder `.tar.gz`)

Andere Formate wie RAR, ACE, Blackhole, LHA etc. bitte nicht verwenden. Auch Windows-Installer-Dateien (`.msi`) oder selbstextrahierende Archive und Installer (`.exe`) sind nicht zu benutzen.

## 1.2 Übersetzen von Programmen

Die für das Übersetzen von Programmen zum Einsatz auf dem fli4l-Router erforderliche Umgebung wird im separat erhältlichen *src*-Paket angeboten. Dort wird auch dokumentiert, wie sich eigene Programme für den fli4l übersetzen lassen.

## 1.3 Modulkonzept

fli4l wird in Module (Pakete) aufgeteilt, u. a. in:

- *fli4l-4.0.0-testing-sunxi-r60742* ← Das Basis-Paket
- *dns\_dhcp*
- *dsl*
- *isdn*
- *sshd*
- und viele weitere...

Mit dem Basis-Paket ist fli4l ein reiner Ethernet-Router. Für ISDN und/oder DSL ist das Paket *isdn* und/oder *dsl* in dem fli4l-Verzeichnis auszupacken. Entsprechendes gilt für die anderen Pakete.

### 1.3.1 mkfli4l

Aus den Paketen werden in Abhängigkeit von der konkreten Konfiguration eine Konfigurationsdatei namens `rc.cfg` und zwei Archive namens `rootfs.img` und `opt.img` erstellt, die alle Konfigurationsinformationen und alle benötigten Dateien enthalten. Diese Dateien werden mit Hilfe von `mkfli4l` erzeugt, welches die einzelnen Pakete einliest und auf Fehler in der Konfiguration prüft.

`mkfli4l` akzeptiert die in Tabelle 1.1 angegebenen Parameter. Fehlen sie, werden die in Klammern angegebenen Werte angenommen. Eine vollständige Liste der Optionen (Tabelle 1.1) erhält man, wenn man

`mkfli41 -h`

aufruft.

Tabelle 1.1: Parameter für `mkfli41`

Option	Bedeutung	
-c, --config	Setzen des Verzeichnisses, in dem <code>mkfli41</code> die config-Dateien der Pakete sucht (Standard: <code>config</code> )	
-x, --check	Setzen des Verzeichnisses, in dem <code>mkfli41</code> die zum Prüfen der Pakete benötigten Dateien sucht ( <code>&lt;package&gt;.txt</code> , <code>&lt;package&gt;.exp</code> und <code>&lt;package&gt;.ext</code> ; Standard: <code>check</code> )	
-l, --log	Setzen der Logdatei; <code>mkfli41</code> protokolliert Fehlermeldungen und Warnungen in dieser Datei (Standard: <code>img/mkfli41.log</code> )	
-p, --package	Angabe der Pakete, die geprüft werden sollen. Diese Option kann mehrmals angegeben werden, wenn man mehrere Pakete im Zusammenhang prüfen will. Bei Verwendung von <code>-p</code> wird allerdings grundsätzlich zuerst die Datei <code>&lt;check_dir&gt;/base.exp</code> eingelesen, um die allgemeinen regulären Ausdrücke, die vom Basis-Paket bereitgestellt werden, zur Verfügung zu stellen. Diese Datei muss also existieren.	
-i, --info	Gibt Auskunft über den Verlauf der Prüfung (welche Dateien werden gelesen, welche Prüfungen werden durchgeführt, welche besonderen Dinge traten während des Prüfprozesses auf)	
-v, --verbose	Ausführlichere Variante von <code>-i</code>	
-h, --help	Zeigt die Hilfe an	
-d, --debug	Erleichtert die Fehlersuche im Prüfprozess. Dies ist als Hilfe für Paketentwickler gedacht, die etwas genauer wissen möchten, wie die Prüfung des Paketes abläuft.	
	Debugoption	Bedeutung
	check	show check process
	zip-list	show generation of zip list
	zip-list-skipped	show skipped files
	zip-list-regexp	show regular expressions for ziplist
	opt-files	check all files in <code>opt/&lt;package&gt;.txt</code>
	ext-trace	show trace of extended checks

### 1.3.2 Aufbau

Ein Paket kann mehrere OPTs enthalten, wenn es aber nur eins enthält, ist es allerdings zweckmäßig, das Paket genauso wie das OPT zu nennen. Im Folgenden ist `<PAKET>` durch den jeweiligen Paket-Namen zu ersetzen. Ein Paket besteht aus folgenden Teilen:

- Verwaltungsdateien
- Dokumentation
- Entwickler-Dokumentation

- Client-Programme
- Quellcode
- Weitere Dateien

Die einzelnen Teile sind im Folgenden näher beschrieben.

### 1.3.3 Die Konfiguration der Pakete

In der Datei `config/<PAKET>.txt` werden vom Benutzer Änderungen an der Konfiguration des Pakets vorgenommen. Alle Variablen eines OPTs sollten einheitlich mit dem Namen des OPTs beginnen, also zum Beispiel:

```
#-----
# Optional package: TELNETD
#-----
OPT_TELNETD='no'      # install telnetd: yes or no
TELNETD_PORT='23'     # telnet port
```

Ein OPT sollte in der Konfigurationsdatei durch einen Header (siehe oben) entsprechend abgegrenzt werden. Dies erhöht die Übersichtlichkeit, zumal ein Paket ja auch mehrere OPTs enthalten kann. Die dem OPT zugehörigen Variablen sollten — ebenfalls im Interesse der Übersichtlichkeit — nicht weiter eingerückt werden. Kommentare und Leerzeilen sind erlaubt, wobei Kommentare einheitlich in Spalte 33 beginnen sollen. Ist eine Variable inklusive ihrer Belegung länger als 32 Zeichen, ist der Kommentar eine Zeile versetzt ab Spalte 33 einzufügen. Längere Kommentare werden jeweils ab Spalte 33 beginnend auf mehrere Zeilen aufgeteilt. Diese Maßnahmen sollen die Lesbarkeit der Konfigurationsdatei erhöhen.

Alle Werte hinter dem Gleichheitszeichen müssen in Hochkommata<sup>1</sup> eingefasst werden, da es sonst beim Booten zu Problemen kommen kann.

Variablen, die aktiv sind (s. u.), werden in die `rc.cfg` übernommen, alles andere wird ignoriert. Einzige Ausnahme sind Variablen mit dem Namen `<PAKET>_DO_DEBUG`. Diese dienen zur Fehlersuche in Paketen und werden pauschal übernommen.

### 1.3.4 Die Liste der zu kopierenden Dateien

Die Datei `opt/<PAKET>.txt` enthält Anweisungen, die beschreiben

- welche Dateien zu welchem OPT gehören,
- wann sie in das zu generierende `opt-` bzw. `rootfs-`Archiv übernommen werden sollen,
- welche UID<sup>2</sup>, GID<sup>3</sup> und Rechte<sup>4</sup> jede Datei bekommen soll,

<sup>1</sup>Es können sowohl einfache Hochkommata als auch doppelte Hochkommata verwendet werden. Man kann also `F00='bar'` oder auch `F00="bar"` schreiben. Die Verwendung von doppelten Hochkommata sollte allerdings die Ausnahme sein und man sollte sich vorher unbedingt darüber informieren, wie eine Unix-Shell mit einfachen und doppelten Hochkommata umgeht.

<sup>2</sup>User ID: Eigentümer der Datei

<sup>3</sup>Group ID: Gruppe der Datei

<sup>4</sup>Darf die Datei gelesen, beschrieben oder ausgeführt werden?

- welche Konvertierungen vor Aufnahme ins Archiv erfolgen sollen.

`mkfli4l` generiert darauf basierend die erforderlichen Archive.

Leere Zeilen und Zeilen, die mit `#` anfangen, werden ignoriert. In einer der ersten Zeilen sollte die Version des Paket-Dateiformats wie folgt stehen:

```
<erste Spalte>      <zweite Spalte> <dritte Spalte>
opt_format_version  1                -
```

Die restlichen Zeilen haben folgende Syntax:

```
<erste Spalte> <zweite Spalte> <dritte Spalte> <folgende spalten>
Variable       Wert           Datei           Optionen
```

1. In der ersten Spalte steht der Name einer Variable, von deren Wert das Übernehmen der in der dritten Spalte stehenden Datei abhängt. Der Name einer Variable kann beliebig oft in der ersten Spalte auftauchen, falls mehrere Dateien von ihr abhängen. Jede Variable, die in der Datei `opt/<PAKET>.txt` auftaucht, wird von `mkfli4l` markiert.

Falls mehrere Variablen auf denselben Wert geprüft werden sollen, kann auch eine Liste von Variablen (durch Kommata getrennt) verwendet werden. In diesem Falle reicht es aus, wenn mindestens *eine* Variable den in der zweiten Spalte geforderten Wert enthält. Wichtig ist dabei, dass zwischen den einzelnen Variablen *keine* Leerzeichen stehen!

Bei OPT-Variablen (also Variablen, die mit `OPT_` beginnen und typischerweise den Typ `YESNO` haben) kann das Präfix „`OPT_`“ weggelassen werden. Des Weiteren ist es unwichtig, ob Variablen in Groß- oder in Kleinbuchstaben (oder beliebig gemischt) notiert werden.

2. In der zweiten Spalte steht ein Wert. Stimmt die in der ersten Spalte stehende Variable mit diesem Wert überein und ist die Variable aktiv (s. u.), wird die Datei in der dritten Spalte übernommen. Steht eine %-Variable in der ersten Spalte, wird über alle Indizes iteriert und geprüft, ob irgendein Element des Arrays mit dem Wert übereinstimmt. Ist das der Fall, wird kopiert. Zusätzlich wird vermerkt, dass aufgrund des aktuellen Wertes der Variable eine Datei kopiert wurde.

Es ist möglich, vor den Wert ein „!“ zu schreiben. In diesem Falle wird die Prüfung negiert, d. h. die Datei wird genau dann kopiert, wenn die Variable diesen Wert *nicht* enthält.

3. In der dritten Spalte steht der Name einer Datei. Die Pfadangabe erfolgt relativ zum `opt`-Verzeichnis. Die Datei muss existieren und lesbar sein, sonst gibt es beim Generieren der Archive einen Fehler und die Generierung wird abgebrochen.

Beginnt der Dateiname mit `rootfs:`, wird die Datei in die Liste der ins `rootfs`-Archiv aufzunehmenden Dateien übernommen. Der Präfix wird vorher entfernt. Taucht dieselbe Datei sowohl mit als auch ohne `rootfs:`-Präfix auf, wird sie nur ins `rootfs`-Archiv übernommen.

Liegt die Datei unterhalb des verwendeten Konfigurationsverzeichnisses, wird sie in die Liste der aus dem Konfigurationsverzeichnis zu übernehmenden Dateien aufgenommen, andernfalls wird die unter `opt/` liegende Datei genommen.



Ist die zu kopierende Datei ein Kernel-Modul, kann man die konkrete Kernel-Version durch `${KERNEL_VERSION}` ersetzen. `mkfli41` nimmt dann die Version aus der Konfiguration und setzt sie hier ein. Dadurch kann man einem Paket Module für verschiedene Kern-Versionen mitgeben und es wird immer die für den Kern richtige Version mit auf den Router kopiert. Für Kernel-Module kann der Pfad auch vollkommen entfallen. `mkfli41` findet das Modul anhand der Dateien `modules.dep` und `modules.alias`, siehe Abschnitt „Automatische Auflösung von Abhängigkeiten für Kernel-Module“ (Seite 12).

4. In den anderen Spalten können die in Tabelle 1.2 aufgeführten Optionen für den Eigentümer, die Gruppe, die Rechte der Dateien und Konvertierungen stehen.

Einige Beispiele:

- kopiere Datei, wenn `OPT_TELNETD='yes'`, setze UID/GID auf root und die Rechte auf 755 (`rw-r-xr-x`):

```
telnetd      yes      usr/sbin/in.telnetd mode=755
```

- kopiere Datei (`OPT_BASE='yes'` gilt implizit immer); setze UID/GID auf root, die Rechte auf 555 (`r-xr-xr-x`) und konvertiere die Datei ins Unix-Format bei gleichzeitigem Entfernen aller überflüssigen Zeichen:

```
base      yes      etc/rc0.d/rc500.killall mode=555 flags=sh
```

- kopiere Kernel-Modul, wenn `PCMCIA_PCIC='i82365'`; setze UID/GID auf root und die Rechte auf 644 (`rw-r--r--`):

```
pcmcia_pcic i82365 lib/modules/${KERNEL_VERSION}/pcmcia/i82365.ko
```

- kopiere Kernel-Modul, wenn `PCMCIA_PCIC='i82365'`; setze UID/GID auf root und die Rechte auf 644 (`rw-r--r--`) (alternative Form):

```
pcmcia_pcic i82365 i82365.ko
```

- kopiere Kernel-Modul, wenn mindestens einer der Einträge in `NET_DRV_%` den Wert 3c503 besitzt; setze UID/GID auf root und die Rechte auf 644 (`rw-r--r--`):

```
net_drv_%   3c503   3c503.ko
```

- kopiere Datei, wenn die Variable `POWERMANAGEMENT` *nicht* den Wert „none“ enthält:

```
powermanagement !none etc/rc.d/rc100.pm mode=555 flags=sh
```

- kopiere Datei, wenn irgendeine der OPT-Variablen `OPT_MYOPTA` oder `OPT_MYOPTB` den Wert „yes“ enthält:

```
myopta,myoptb yes usr/local/bin/myopt-common.sh mode=555 flags=sh
```

Tabelle 1.2: Optionen für Dateien

Option	Bedeutung	Standardwert
type=	<p>Der Typ des Eintrags:</p> <p><i>local</i>     Dateisystem-Objekt  <i>file</i>       Datei  <i>dir</i>        Verzeichnis  <i>node</i>       Gerät  <i>symlink</i>   (symbolische) Verknüpfung</p> <p>Wenn vorhanden, muss diese Option an erster Stelle stehen. Der Typ „local“ steht hierbei für den Typ eines im Dateisystem existierenden Objekts und entspricht somit „file“, „dir“, „node“ oder „symlink“ (je nachdem). Die anderen Typen mit Ausnahme von „file“ können Einträge im Archiv erzeugen, die nicht im lokalen Dateisystem vorliegen müssen. Das wird z. B. benutzt, um Gerätedateien im <b>rootfs</b>-Archiv anzulegen.</p>	local
uid=	Der Eigentümer der Datei, entweder numerisch oder als Name aus passwd	root
gid=	Die Gruppe der Datei, entweder numerisch oder als Name aus group	root
mode=	Die Zugriffsrechte	<p>Dateien und Geräte:  <b>rw-r--r--</b> (644)  Verzeichnisse:  <b>rw-r--r--</b> (755)  Verknüpfungen:  <b>rw-rw-rw-</b> (777)</p>
flags= (type=file)	<p>Konvertierungen vor der Aufnahme ins Archiv:</p> <p><i>utxt</i>    Konvertierung ins Unix-Format  <i>dtxt</i>    Konvertierung ins DOS-Format  <i>sh</i>       Shell-Skript: Konvertierung ins Unix-Format, Entfernen überflüssiger Zeichen  <i>luac</i>    Lua-Skript: Übersetzung in Bytecode der Lua-VM</p>	
name=	Alternativer Name, unter dem der Eintrag ins Archiv aufgenommen wird	
devtype= (type=node)	Beschreibt den Typ des Geräts („c“ für zeichenorientierte und „b“ für blockorientierte Geräte). Muss an zweiter Stelle stehen.	
major= (type=node)	Beschreibt die so genannte „Major“-Nummer der Gerätedatei. Muss an dritter Stelle stehen.	
minor= (type=node)	Beschreibt die so genannte „Minor“-Nummer der Gerätedatei. Muss an vierter Stelle stehen.	
linktarget= (type=symlink)	Beschreibt das Ziel der symbolischen Verknüpfung. Muss an zweiter Stelle stehen.	

Dieses Beispiel ist letztlich nur eine Kurzschreibweise für:

```
myopta yes usr/local/bin/myopt-common.sh mode=555 flags=sh
myoptb yes usr/local/bin/myopt-common.sh mode=555 flags=sh
```

Und letzteres ist eine Kurzschreibweise für:

```
opt_myopta yes usr/local/bin/myopt-common.sh mode=555 flags=sh
opt_myoptb yes usr/local/bin/myopt-common.sh mode=555 flags=sh
```

- kopiere Datei `opt/usr/bin/beep.sh` ins `rootfs`-Archiv, aber benenne sie vorher in `bin/beep` um:

```
base yes rootfs:usr/bin/beep.sh mode=555 flags=sh name=bin/beep
```

Wenn im Paket eine Variable referenziert wird, die nicht vom Paket selbst definiert wird, kann es passieren, dass das entsprechende Paket nicht installiert ist. Das führt für gewöhnlich zu einer Fehlermeldung in `mkfli41`, da `mkfli41` erwartet, dass alle von `opt/<PAKET>.txt` referenzierten Variablen definiert sind.

Um diese Situation korrekt handhaben zu können, wurde die „weak“-Deklaration eingeführt. Sie hat das folgende Format:

```
weak      <Variable>      -
```

Dadurch wird die Variable definiert und ihr Wert intern auf „undefiniert“ gesetzt, wenn sie nicht bereits definiert worden ist. Dabei ist jedoch zu beachten, dass hier das „OPT\_“-Präfix *nicht* weggelassen werden darf (falls es existiert), weil sonst eine Variable *ohne* dieses Präfix definiert wird.

Ein Beispiel aus der `opt/rrdtool.txt`:

```
weak opt_openvpn -
[...]
openvpn yes usr/lib/collectd/openvpn.so
```

Ohne die `weak`-Definition würde `mkfli41` bei der Nutzung des Pakets „rrdtool“ eine Fehlermeldung anzeigen, wenn das „openvpn“-Paket nicht ebenfalls vorliegt. Mit Hilfe der `weak`-Definition kommt auch in dem Fall, dass das „openvpn“-Paket nicht vorliegt, keine Fehlermeldung.

## Konfigurations-spezifische Dateien

In manchen Situationen möchte man originale Dateien im `opt`- oder `rootfs`-Archiv durch Konfigurations-spezifische Dateien wie z.B. Host-Keys, eigene Firewall-Skripte, ... ersetzen. `mkfli41` unterstützt dieses Szenario, indem es prüft, ob eine zu kopierende Datei im Konfigurationsverzeichnis zu finden ist, und übernimmt in diesem Falle diese Datei in die Liste der ins Archiv aufzunehmenden Dateien.

Eine weitere Möglichkeit, Konfigurations-spezifische Dateien ins Archiv aufzunehmen, wird im Abschnitt [„Erweiterte Prüfungen der Konfiguration“](#) (Seite 30) beschrieben.

## Automatische Auflösung von Abhängigkeiten für Kernel-Module

Kernel-Module bauen unter Umständen auf anderen Kernel-Modulen auf. Diese müssen vor ihnen geladen werden und daher gleichfalls in das Archiv aufgenommen werden. `mkfli4l` bestimmt diese Abhängigkeiten anhand von `modules.dep` und `modules.alias`, zweier beim Kernel-Bauen generierter Dateien, und nimmt automatisch alle benötigten Module in die Archive auf. So führt z. B. folgender Eintrag

```
net_drv_%    ne2k-pci    ne2k-pci.ko
```

dazu, dass auch `8390.ko` ins Archiv aufgenommen wird, da `ne2k-pci.ko` davon abhängt.

Die notwendigen Einträge in `modules.dep` und `modules.alias` werden in das `rootfs`-Archiv mit aufgenommen und können von `modprobe` zum Laden der Treiber genutzt werden.

### 1.3.5 Die Prüfung der Konfiguration-Variablen

Mit Hilfe der Datei `check/<PAKET>.txt` können die Inhalte der Variablen auf Gültigkeit überprüft werden. Diese Überprüfung war in früheren Versionen fest im Programm `mkfli4l` eingebaut, wurde aber im Zuge der Modularisierung von `fli4l` in die Check-Dateien ausgelagert. In dieser Datei ist für jede Variable aus den Konfigurationsdateien eine Zeile vorhanden. Diese Zeilen bestehen aus vier bis fünf Spalten, welche folgende Funktionen haben:

1. Variable: Diese Spalte gibt den Namen der zu überprüfenden Variable aus der Konfigurationsdatei an. Wenn es sich dabei um eine so genannte *Array-Variable* handelt, die mehrmals mit verschiedenen Indizes auftauchen kann, wird an Stelle der Nummer ein Prozentzeichen (%) in den Variablenname eingefügt. Dieses wird immer als „\_%“ in der Mitte eines Namens bzw. „\_%“ am Ende eines Namens verwendet. Der Name kann dabei mehrere Prozentzeichen enthalten, so dass man auch mehrdimensionale Arrays realisieren kann. Dann sollte zwischen den Prozentzeichen allerdings etwas stehen, muss aber nicht, was dann allerdings zu so seltsamen Namen wie „FOO\_%\_%“ führt.

Oftmals hat man das Problem, dass bestimmte Variablen Optionen beschreiben, die man nur in bestimmten Situationen benötigt. Deshalb können Variablen als optional markiert werden. Optionale Variablen werden mit einem vorangestellten „+“ gekennzeichnet. Sie können dann da sein, müssen aber nicht. Arrays können auch mit einem „++“ Präfix versehen werden. Steht ein „+“ davor, kann das Array da sein oder ganz fehlen. Steht „++“ davor, können zusätzlich auch noch einzelne Elemente des Arrays fehlen.

2. OPT\_VARIABLE: Diese Spalte teilt die Variable einem bestimmten OPT zu. Die Variable wird nur auf Gültigkeit überprüft, wenn die hier angegebene Variable auf „yes“ steht. Gibt es keine OPT-Variable, ist hier ein „-“ anzugeben. In diesem Fall muss die Variable in der Konfigurationsdatei definiert werden, es sei denn, es wird eine Standard-Belegung definiert (s. u.). Der Name der OPT-Variable kann beliebig sein, er sollte jedoch mit dem Präfix „OPT\_“ beginnen.

Falls eine Variable von keiner OPT-Variablen abhängt, gilt sie als *aktiv*. Falls sie von einer OPT-Variablen abhängig ist, ist sie genau dann aktiv, wenn

- ihre OPT-Variable aktiv ist und
- ihre OPT-Variable den Wert „yes“ enthält.

Andernfalls ist die Variable inaktiv.

**Hinweis:** Inaktive OPT-Variablen werden, wenn sie in der Konfiguration mit „yes“ belegt werden, auf den Wert „no“ zurückgesetzt; dies wird von `mkfli4l` auch mit einer entsprechenden Warnmeldung (bspw. „OPT\_Y='yes' ignored, because OPT\_X='no'“) kommentiert. Bei transitiven Abhängigkeitsketten (OPT\_Z hängt von OPT\_Y ab, das wiederum von OPT\_X abhängt) funktioniert dies aber nur dann zuverlässig, wenn die Namen aller OPT-Variablen mit „OPT\_“ beginnen.

3. **VARIABLE\_N:** Steht in der ersten Spalte eine Variable mit einem % im Namen, wird hier die Variable angegeben, die die Häufigkeit des Auftretens der Variable beschreibt (die so genannte *N-Variable*). Ist die Variable mehrdimensional, wird die Häufigkeit des letzten Index beschrieben. Hängt die Variable von einem OPT ab, muss die N-Variable vom selben OPT oder von keinem OPT abhängig sein. Ist die Variable von keinem OPT abhängig, darf auch die N-Variable von keinem OPT abhängig sein. Gibt es keine N-Variable, ist hier ein „-“ anzugeben.

Aus Kompatibilitätsgründen mit zukünftigen `fli4l`-Versionen *muss* die hier angegebene Variable identisch sein mit der Variable in `OPT_VARIABLE`, wobei das letzte „%“ durch ein „N“ ersetzt und alles dahinter entfernt wurde. Ein Array `HOST_%_IP4` bekommt also zwingend die N-Variable `HOST_N` zugewiesen und ein Array `PF_USR_CHAIN_%_RULE_%` also die N-Variable `PF_USR_CHAIN_%_RULE_N`, und diese N-Variable ist selbst wieder eine Array-Variable mit der zugehörigen N-Variable `PF_USR_CHAIN_N`. *Alle anderen Benennungen der N-Variable werden mit zukünftigen fli4l-Versionen inkompatibel sein!*

4. **VALUE:** Diese Spalte gibt an, welche Werte für diese Variable eingegeben werden können. Es sind dabei z. B. folgende Angaben möglich:

Name	Bedeutung
NONE	Es wird keine Überprüfung vorgenommen
YESNO	Die Variable muss „yes“ oder „no“ sein
NOTEMPTY	Die Variable darf nicht leer sein
NOBLANK	Die Variable darf kein Leerzeichen enthalten
NUMERIC	Die Variable muss numerisch sein
IPADDR	Die Variable muss eine IP-Adresse sein
DIALMODE	Die Variable muss „on“, „off“ oder „auto“ sein

Werden die Werte mit einem „WARN\_“-Präfix versehen, so führt ein illegaler Wert nicht zu einer Fehlermeldung und damit zu einem Abbruch von `mkfli4l`, sondern nur zur Ausgabe einer Warnung.

Die möglichen Prüfungen werden durch reguläre Ausdrücke in `check/base.exp` definiert. Diese Datei kann erweitert werden und enthält neuerdings z. B. zusätzlich folgende Prüfungen: `HEX`, `NUMHEX`, `IP_ROUTE`, `DISK` und `PARTITION`.

Die Anzahl der Ausdrücke kann jederzeit erweitert werden, hier ist Rückmeldung von den Paket-Entwicklern erforderlich.

Zusätzlich können reguläre Ausdrücke auch direkt in den Check-Dateien angegeben werden, wobei man auch Bezug auf existierende Ausdrücke nehmen kann. Statt `YESNO` könnte man z. B. auch

```
RE:yes|no
```

schreiben. Sinnvoll ist es dann, wenn ein Test nur ein einziges Mal ausgeführt wird und relativ einfach ist. Für genauere Informationen siehe nächstes Kapitel.

5. Standard-Belegung: In dieser Spalte kann optional ein Standard-Wert für die Variable stehen, falls die Variable nicht in der Konfiguration steht.

**Hinweis:** Dies funktioniert zur Zeit jedoch nicht für Array-Variablen. Auch darf die Variable nicht optional sein, es darf also kein „+“ vor dem Variablennamen stehen.

Beispiel:

```
OPT_TELNETD      -      -      YESNO      "no"
```

Fehlt OPT\_TELNETD nun in der Konfigurationsdatei, wird „no“ angenommen und dieser Wert auch in die rc.cfg geschrieben.

Die Sache mit dem Prozentzeichen lässt sich am Besten mit einem Beispiel erklären. Nehmen wir an, in der check/base.txt steht:

```
NET_DRV_N      -      -      NUMERIC
NET_DRV_%      -      NET_DRV_N      NONE
NET_DRV_%_OPTION  -      NET_DRV_N      NONE
```

Das heißt, dass je nach Wert von NET\_DRV\_N die Variablen NET\_DRV\_N, NET\_DRV\_1\_OPTION, NET\_DRV\_2\_OPTION, NET\_DRV\_3\_OPTION, usw. überprüft werden.

### 1.3.6 Eigene Definitionen zum Prüfen der Konfigurationsvariablen

#### Einführung regulärer Ausdrücke

In der Version 2.0 gab es nur die oben angeführten sieben Werte-Bereiche, auf die Variablen geprüft werden können: NONE, NOTEMPTY, NUMERIC, IPADDR, YESNO, NOBLANK, DIALMODE. Die Überprüfung war in mkf1i41 fest eingebaut, nicht erweiterbar und beschränkte sich auf wesentliche „Datentypen“, die mit vertretbarem Aufwand geprüft werden können.

Mit der Version 2.1 wurde diese Prüfung neu implementiert. Ziel der neuen Implementierung ist eine flexiblere Prüfung der Variablen, die auch in der Lage ist, komplexere Ausdrücke zu prüfen. Deshalb werden reguläre Ausdrücke verwendet, die in einem oder mehreren separaten Dateien abgespeichert werden. Dadurch wird es zum einen möglich, Variablen zu prüfen, die im Augenblick noch nicht geprüft werden, und zum anderen können Entwickler optionaler Pakete eigene Ausdrücke definieren, um die Konfiguration ihrer Pakete prüfen zu lassen.

Eine Beschreibung regulärer Ausdrücke findet man via „man 7 regex“ oder z. B. hier: <http://unixhelp.ed.ac.uk/CGI/man-cgi?regex+7>.

#### Spezifikation regulärer Ausdrücke

Spezifizieren kann man die Ausdrücke auf zwei Wegen:

1. Paketspezifische exp-Datei check/<PAKET>.exp

Diese Datei liegt im check-Verzeichnis und trägt den gleichen Namen wie das dazugehörige Paket, also z. B. check/base.exp. Sie enthält Definitionen für Ausdrücke, die in der Datei check/<PAKET>.txt referenziert werden können. So enthält check/base.exp im

Augenblick Definitionen für die bekannten Prüfungen und `check/isdn.exp` eine Definition für die Variable `ISDN_CIRC_x_ROUTE` (das Fehlen dieser Überprüfung war der Auslöser dieser Änderungen).

Die Syntax lautet wie folgt, wobei man auch hier bei Bedarf doppelte Hochkommata verwenden kann:

```
<Name> = '<Regulärer Ausdruck>' : '<Fehlermeldung>'
```

oder am Beispiel aus `check/base.exp`:

```
NOTEMPTY = '.*[~ ]+.*'           : 'should not be empty'
YESNO     = 'yes|no'              : 'only yes or no are allowed'
NUMERIC   = '0|[1-9][0-9]*'       : 'should be numeric (decimal)'
OCTET     = '1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]'
           : 'should be a value between 0 and 255'
IPADDR    = '((RE:OCTET)\.){3}(RE:OCTET)' : 'invalid ipv4 address'
EIPADDR   = '()|(RE:IPADDR)'
           : 'should be empty or contain a valid ipv4 address'
NOBLANK   = '[^ ]+'              : 'should not contain spaces'
DIALMODE  = 'auto|manual|off'     : 'only auto, manual or off are allowed'
NETWORKS  = '(RE:NETWORK)([[:space:]]+(RE:NETWORK))*'
           : 'no valid network specification, should be one or more
             network address(es) followed by a netmask,
             for instance 192.168.6.0/24'
```

In den regulären Ausdrücken können auch Referenzen auf bereits existierende Definitionen enthalten sein. Diese werden dann einfach an der Stelle eingefügt. Dadurch ist es einfacher, reguläre Ausdrücke zu konstruieren. Eingefügt werden die Referenzen einfach durch `'(RE:Referenz)'`. (Siehe die Definition des Ausdrucks `NETWORKS` oben für ein entsprechendes Beispiel.)

Die Fehlermeldungen tendieren dazu, zu lang zu werden. Daher besteht die Möglichkeit, sie über mehrere Zeilen zu verteilen. Die folgenden Zeilen müssen dann immer mit einem Leerzeichen oder Tabulator beginnen. Beim Einlesen der `check/<PAKET>.exp`-Datei werden überflüssige Leerzeichen auf eins reduziert und Tabulatoren durch Leerzeichen ersetzt. Ein Eintrag in der `check/<PAKET>.exp` könnte dann so aussehen:

```
NUM_HEX      = '0x[[:xdigit:]]+'
              : 'should be a hexadecimal number
                (a number starting with "0x")'
```

## 2. Reguläre Ausdrücke direkt in der Check-Datei `check/<PAKET>.txt`

Manche Ausdrücke kommen nur einmal vor, dann lohnt es sich nicht, dafür einen regulären Ausdruck in einer `check/<PAKET>.exp`-Datei zu definieren. Dann kann man diesen Ausdruck einfach in die Check-Datei schreiben, z. B.:

# Variable	OPT_VARIABLE	VARIABLE_N	VALUE
MOUNT_BOOT	-	-	RE:ro rw no

MOUNT\_BOOT kann lediglich die Werte „ro“, „rw“ oder „no“ annehmen, alles andere wird abgelehnt.

Will man Bezug auf existierende reguläre Ausdrücke nehmen, fügt man einfach eine Referenz via „(RE:...)" ein. Beispiel:

# Variable	OPT_VARIABLE	VARIABLE_N	VALUE
LOGIP_LOGDIR	OPT_LOGIP	-	RE:(RE:ABS_PATH) auto

### Erweiterung existierender regulärer Ausdrücke

Fügt ein optionales Paket einen zusätzlichen Wert für eine Variable hinzu, die von einem regulären Ausdruck geprüft wird, muss der reguläre Ausdruck erweitert werden. Dies geschieht einfach durch Definition der neuen möglichen Werte durch einen regulären Ausdruck (wie oben beschrieben) und Ergänzung des bestehenden regulären Ausdrucks in einer eigenen `check/<PAKET>.exp`-Datei. Dass ein bestehender Ausdruck modifiziert werden soll, kennzeichnet ein führendes „+“. Der neue Ausdruck ergänzt den bestehenden Ausdruck, indem der neue Wert als Alternative an den bestehenden Wert angehängt wird. Verwendet ein anderer Ausdruck den ergänzten Ausdruck, gilt auch dort die Ergänzung. Die angegebene Fehlermeldung wird einfach an die vorhandene hinten angehängt.

Am Beispiel der Ethernet-Treiber könnte das wie folgt aussehen:

- Das Basis-Paket stellt eine Menge von Ethernet-Treibern bereit und prüft die Variable `NET_DRV_x` mit dem regulären Ausdruck `NET_DRV`, der wie folgt spezifiziert ist:

```
NET_DRV          = '3c503|3c505|3c507|...'
                  : 'invalid ethernet driver, please choose one'
                  : ' of the drivers in config/base.txt'
```

- Das Paket „pcmcia“ stellt jetzt zusätzliche Gerätetreiber bereit, muss also `NET_DRV` ergänzen. Das sieht dann wie folgt aus:

```
PCMCIA_NET_DRV   = 'pcnet_cs|xirc2ps_cs|3c574_cs|...' : ''
+NET_DRV          = '(RE:PCMCIA_NET_DRV)' : ''
```

Nun kann man zusätzlich auch noch PCMCIA-Treiber auswählen.

### Regulären Ausdruck in Abhängigkeit von YESNO-Variablen erweitern

Wenn man `NET_DRV` wie oben um die PCMCIA-Treiber erweitert hat, aber das Paket „pcmcia“ deaktiviert hat, könnte man dennoch einen PCMCIA-Treiber in der `config/base.txt` auswählen, ohne dass eine Fehlermeldung beim Erstellen der Archive auftritt. Um das zu verhindern, kann man den regulären Ausdruck auch abhängig von einer YESNO-Variablen in der Konfiguration erweitern. Dazu wird der Name der Variablen, die bestimmt ob der Ausdruck erweitert wird, mit runden Klammern direkt hinter den Namen des Ausdrucks gehängt. Ist die Variable aktiv und hat den Wert „yes“, wird der Ausdruck erweitert, sonst nicht.

```
PCMCIA_NET_DRV   = 'pcnet_cs|xirc2ps_cs|3c574_cs|...' : ''
+NET_DRV(OPT_PCMCIA) = '(RE:PCMCIA_NET_DRV)' : ''
```



Wird jetzt `OPT_PCMCIA='no'` gesetzt, und in der `config/base.txt` wird z. B. der PCMCIA-Treiber `xirc2ps_cs` benutzt, gibt es beim Erstellen der Archive eine Fehlermeldung.

**Hinweis:** Dies funktioniert *nicht*, wenn die Variable nicht explizit in der Konfigurationsdatei gesetzt wird, sondern ihren Wert über eine Standard-Belegung in der `check/<PAKET>.txt` erhält. In diesem Fall muss man also in der Konfigurationsdatei die Variable explizit setzen und ggf. auf die Standard-Belegung verzichten.

### Regulären Ausdruck in Abhängigkeit von anderen Variablen erweitern

Alternativ kann man auch beliebige Werte von Variablen als Bedingung verwenden, die Syntax sieht dann wie folgt aus:

```
+NET_DRV(KERNEL_VERSION=~'^3\.18\..*$') = ...
```

Wenn `KERNEL_VERSION` zu dem angegebenen regulären Ausdruck passt, also irgendein Kernel aus der 3.18er Versionsreihe genutzt wird, dann wird die Liste der erlaubten Netzwerktreiber um die angegebenen Treiber ergänzt.

**Hinweis:** Dies funktioniert *nicht*, wenn die Variable nicht explizit in der Konfigurationsdatei gesetzt wird, sondern ihren Wert über eine Standard-Belegung in der `check/<PAKET>.txt` erhält. In diesem Fall muss man also in der Konfigurationsdatei die Variable explizit setzen und ggf. auf die Standard-Belegung verzichten.

### Fehlermeldungen

Findet die Prüfung einen Fehler, erscheint eine Fehlermeldung der folgenden Art:

```
Error: wrong value of variable HOSTNAME: '' (may not be empty)
Error: wrong value of variable MOUNT_OPT: 'rx' (user supplied regular expression)
```

Beim ersten Fehler wurde der Ausdruck in einer `check/<PAKET>.exp`-Datei definiert und ein Hinweis auf den Fehler wird mit ausgegeben. Im zweiten Falle wurde der Ausdruck direkt in einer `check/<PAKET>.txt`-Datei spezifiziert, deshalb gibt es keinen zusätzlichen Hinweis auf die Fehlerursache.

### Definition regulärer Ausdrücke

Reguläre Ausdrücke sind wie folgt definiert:

Regulärer Ausdruck: Eine oder mehrere Alternativen, getrennt durch '|', z. B. „ro|rw|no“. Trifft eine der Alternativen zu, trifft der ganze Ausdruck zu (hier wären „ro“, „rw“ und „no“ gültige Ausdrücke).

Eine Alternative ist eine Verkettung mehrerer Teilstücke, die einfach aneinandergereiht werden.

Ein Teilstück ist ein „Atom“, gefolgt von einem einzelnen „\*“, „+“, „?“ oder „{min, max}“. Die Bedeutung ist wie folgt:

- „a\*“ — beliebig viele „a“s (erlaubt auch den Fall, das gar kein „a“ da ist)
- „a+“ — mindestens ein „a“
- „a?“ — kein oder ein „a“

- „a{2,5}“ — zwei bis fünf „a“s
- „a{5}“ — genau fünf „a“s
- „a{2,}“ — mindestens zwei „a“s
- „a{,5}“ — höchstens fünf „a“s

Ein „Atom“ ist ein

- regulärer Ausdruck eingeschlossen in Klammern, z. B. trifft „(a|b)+“ auf eine beliebige Zeichenkette zu, die mindestens ein „a“ oder „b“ enthält, sonst aber beliebig viele und in beliebiger Reihenfolge
- ein leeres Paar Klammern steht für einen „leeren“ Ausdruck
- ein Ausdruck mit eckigen Klammern „[]“ (siehe weiter unten)
- ein Punkt „.“, der auf irgendein einzelnes Zeichen zutrifft, z. B. trifft „.+“ auf eine beliebige Zeichenkette zu, die mindestens ein Zeichen enthält
- ein „^“ steht für den Zeilenanfang, z. B. trifft „^a.\*“ auf eine Zeichenkette zu, die mit einem „a“ anfängt und in der beliebige Zeichen folgen, etwa „a“ oder „adkadhashdkash“
- ein „\$“ steht für das Zeilenende
- ein „\“ gefolgt von einem der Sonderzeichen `^ . [ $ ( ) | * + ? { \` steht für genau das zweite Zeichen ohne seine spezielle Bedeutung
- ein normales Zeichen trifft auf genau das Zeichen zu, z. B. trifft „a“ genau auf „a“ zu.

Ein Ausdruck in rechteckigen Klammern bedeutet Folgendes:

- „x-y“ — trifft auf irgendein Zeichen zu, das zwischen „x“ und „y“ liegt, z. B. steht „[0-9]“ für alle Zeichen zwischen „0“ und „9“; „[a-zA-Z]“ steht für alle Buchstaben, egal ob groß oder klein
- „^ x-y“ — trifft auf irgendein Zeichen zu, das *nicht* im angegebenen Intervall liegt; so steht z. B. „[^ 0-9]“ für alle Zeichen, die *keine* Ziffern sind
- „[:character-class:]“ — trifft auf ein Zeichen der Zeichenklasse *character-class* zu. Relevante Standardzeichenklassen sind: `alnum`, `alpha`, `blank`, `digit`, `lower`, `print`, `punct`, `space`, `upper` und `xdigit`. So steht „[:alpha:]“ für alle Groß- und Kleinbuchstaben und ist somit identisch zu „[:lower:] [:upper:]“.

## Beispiele für reguläre Ausdrücke

Sehen wir uns das mal an einigen Beispielen an!

**NUMERIC:** Ein numerischer Wert besteht aus mindestens einer, aber ansonsten beliebig vielen Ziffern. „Mindestens ein“ drückt man mit „+“ aus, eine Ziffer hatten wir schon als Beispiel. Zusammengesetzt ergibt das:

```
NUMERIC = '[0-9]+'
```

oder alternativ

```
NUMERIC = '[:digit:]*'
```

NOBLANK: Ein Wert, der keine Leerzeichen enthält, ist ein beliebiges Zeichen (außer dem Leerzeichen) und davon beliebig viele:

```
NOBLANK = '[^ ]*'
```

bzw. wenn der Wert zusätzlich auch nicht leer sein darf:

```
NOBLANK = '[^ ]+'
```

IPADDR: Sehen wir uns das Ganze nochmal am Beispiel der IPv4-Adresse an. Eine IPv4-Adresse besteht aus vier „Octets“, die durch einen Punkt („.“) voneinander getrennt sind. Ein Octet kann eine Zahl zwischen 0 und 255 sein. Definieren wir als erstes ein Octet. Es kann

eine Zahl zwischen 0 und 9 sein:	[0-9]
eine Zahl zwischen 10 und 99:	[1-9][0-9]
eine Zahl zwischen 100 und 199:	1[0-9][0-9]
eine Zahl zwischen 200 und 249:	2[0-4][0-9]
eine Zahl zwischen 250 und 255 sein:	25[0-5]

Das Ganze sind Alternativen, also fassen wir sie einfach mittels „|“ zu einem Ausdruck zusammen: „[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]“ und haben damit ein Octet. Daraus können wir nun eine IPv4-Adresse machen, vier Octets mit Punkten voneinander getrennt (der Punkt muss mittels eines *Backslashes* maskiert werden, da er sonst für ein beliebiges Zeichen steht). Basierend auf der Syntax der exp-Dateien sieht das Ganze dann wie folgt aus:

```
OCTET  = '[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5] '
IPADDR = '(\(RE:OCTET\)\\.){3}(RE:OCTET) '
```

### Unterstützung beim Entwurf regulärer Ausdrücke

Will man reguläre Ausdrücke entwerfen und testen, kann man dazu das „regex“-Programm verwenden, das sich in dem Verzeichnis **unix** bzw. **windows** des Pakets „base“ befindet. Es akzeptiert die folgende Syntax:

```
usage: regex [-c <check dir>] <regex> <string>
```

Dabei bedeuten die Parameter Folgendes:

- **<check dir>** ist das Verzeichnis, das die Check-Dateien und damit auch die exp-Dateien enthält. Diese werden von „regex“ eingelesen, damit man auf bereits definierte Ausdrücke zurückgreifen kann.
- **<regex>** ist der reguläre Ausdruck (im Zweifelsfall immer in '...' oder "...“ angeben, wobei doppelte Anführungsstriche nur nötig sind, wenn einfache Hochkommata in dem Ausdruck vorkommen sollen)
- **<string>** ist die zu prüfende Zeichenkette

Das könnte z. B. wie folgt aussehen:

```
./i586-linux-regex -c ../check '[0-9]' 0
adding user defined regular expression='[0-9]' ('^([0-9])$')
checking '0' against regexp '[0-9]' ('^([0-9])$')
'[0-9]' matches '0'

./i586-linux-regex -c ../check '[0-9]' a
adding user defined regular expression='[0-9]' ('^([0-9])$')
checking 'a' against regexp '[0-9]' ('^([0-9])$')
regex error 1 (No match) for value 'a' and regexp '[0-9]' ('^([0-9])$')

./i586-linux-regex -c ../check IPADDR 192.168.0.1
using predefined regular expression from base.exp
adding IPADDR='((RE:OCTET)\.){3}(RE:OCTET)'
('^(((1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}(1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]))$')
'IPADDR' matches '192.168.0.1'

./i586-linux-regex -c ../check IPADDR 192.168.0.256
using predefined regular expression from base.exp
adding IPADDR='((RE:OCTET)\.){3}(RE:OCTET)'
('^(((1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}(1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]))$')
regex error 1 (No match) for value '192.168.0.256' and regexp
'((RE:OCTET)\.){3}(RE:OCTET)'
(unknown:-1) wrong value of variable cmd_var: '192.168.0.256' (invalid ipv4 address)
```

### 1.3.7 Erweiterte Prüfungen der Konfiguration

Manchmal ist es notwendig, komplexere Überprüfungen durchzuführen. Beispiele für solche komplexeren Dinge wären z. B. Abhängigkeiten zwischen Paketen oder Bedingungen, die nur erfüllt sein müssen, wenn Variablen bestimmte Werte annehmen. So muss z. B. bei Auswahl eines PCMCIA-ISDN-Adapters auch das Paket „pcmcia“ installiert werden.

Um diese Überprüfungen durchführen zu können, kann man in `check/<PAKET>.ext` (auch ext-Skript genannt) kleinere Tests schreiben. Die Sprache besteht aus folgenden Elementen:

#### 1. Schlüsselwörter:

- Kontrollfluss:
  - if (*expr*) then *statement* else *statement* fi
  - foreach *var* in *set\_var* do *statement* done
  - foreach *var* in *set\_var\_1* ... *set\_var\_n* do *statement* done
  - foreach *var* in *var\_n* do *statement* done
- Abhängigkeiten:
  - provides *package* version *x.y.z*
  - depends on *package* version *x1.y1 x2.y2.z2 x3.y3* ...
- Aktionen:
  - warning "*warning*"
  - error "*error*"

```

- fatal_error "fatal error"
- set var = value
- crypt (variable)
- stat (filename, res)
- fgrep (filename, regex)
- split (string, set_variable, character)

```

2. Datentypen: Zeichenketten, positive ganze Zahlen, Versionsnummern

3. Logische Operationen: <, ==, >, !=, !, &&, ||, =~, copy\_pending, samenet, subnet

## Datentypen

Zu den Datentypen ist zu sagen, dass Variablen auf Grund des zugehörigen regulären Ausdrucks fest einem Datentyp zugeordnet werden:

- Variablen, deren Typ mit „NUM“ beginnt, sind numerisch und enthalten positive ganze Zahlen
- Variablen, die eine N-Variable für irgendein Array sind, sind ebenfalls numerisch
- alle anderen Variablen werden wie Zeichenketten verarbeitet

Das bedeutet unter anderem, dass eine Variable vom Typ `ENUMERIC` *nicht* als Index beim Zugriff auf eine Array-Variable benutzt werden kann, auch wenn man sich vorher vergewissert hat, dass sie nicht leer ist. Der folgende Code funktioniert somit nicht:

```

# sei TEST eine Variable vom Typ ENUMERIC
if (test != "")
then
    # Fehler: You can't use a non-numeric ID in a numeric
    #         context. Check type of operand.
    set i=my_array[test]
    # Fehler: You can't use a non-numeric ID in a numeric
    #         context. Check type of operand.
    set j=test+2
fi

```

Eine Lösung für dieses Problem bietet [split](#) (Seite 29):

```

if (test != "")
then
    # alle Elemente von test_% sind numerisch
    split(test, test_%, ' ', numeric)
    # OK
    set i=my_array[test_%[1]]
    # OK
    set j=test_%[1]+2
fi

```

## Zeichenketten und Variablenersetzung

An verschiedenen Stellen werden Zeichenketten benötigt, etwa wenn eine [Warnung](#) (Seite 24) ausgegeben werden soll. In einigen Fällen, die in dieser Dokumentation beschrieben werden, wird eine solche Zeichenkette dabei nach Variablen durchsucht; werden welche gefunden, werden diese durch ihren Inhalt oder andere Attribute *ersetzt*. Diese Ersetzung wird *Variablenersetzung* genannt.

Dies soll an einem Beispiel verdeutlicht werden. Es gelte die Konfiguration:

```
# config/base.txt
HOSTNAME='fli41'
# config/dns_dhcp.txt
HOST_N='1' # Anzahl der Hosts
HOST_1_NAME='client'
HOST_1_IP4='192.168.1.1'
```

Dann werden die Zeichenketten wie folgt umgeschrieben, wenn die Variablenersetzung in dem jeweiligen Kontext aktiv ist:

```
"Mein Router heißt $HOSTNAME"
# --> "Mein Router heißt fli41"
"HOSTNAME ist Teil des Pakets ${HOSTNAME}"
# --> "HOSTNAME ist Teil des Pakets base"
"@HOST_N ist $HOST_N"
# --> " # Anzahl der Hosts ist 1"
```

Wie man sehen kann, gibt es prinzipiell drei Möglichkeiten der Ersetzung:

- **\$<Name>** bzw. **\${<Name>}**: Ersetzt den Variablennamen durch den Inhalt der Variable. Dies ist die häufigste Form der Ersetzung. Der Name muss in `{...}` stehen, wenn direkt danach in der Zeichenkette ein Zeichen kommt, das gültiger Bestandteil eines Variablennamens sein kann, also ein Buchstabe, eine Ziffer oder ein Unterstrich. In allen anderen Fällen ist die Verwendung von geschweiften Klammern möglich, aber nicht zwingend.
- **%<Name>** bzw. **%{<Name>}**: Ersetzt den Variablennamen durch den Namen des Pakets, in dem die Variable definiert ist. Dies funktioniert *nicht* bei im Skript via [set](#) (Seite 24) zugewiesenen Variablen oder bei Laufvariablen einer [foreach-Schleife](#) (Seite 31), da solche Variablen kein Paket besitzen und für Laufvariablen diese Syntax eine andere Bedeutung erhält.
- **@<Name>** bzw. **@{<Name>}**: Ersetzt den Variablennamen durch den Kommentar, der in der Konfiguration hinter der Variablen steht. Auch dies ergibt keinen Sinn für im Skript definierte Variablen.

Will man ein „\$“, „@“ oder „%“ im Text haben, schreibt man „\$\$“, „@@“ bzw. „%%“.

**Hinweis:** Elemente von Array-Variablen können auf diese Weise *nicht* in Zeichenketten integriert werden, weil es keine Möglichkeit gibt, einen Index anzugeben.

Generell unterliegen nur *Konstanten* der Variablenersetzung; Zeichenketten, die über eine Variable hereinkommen, bleiben unverändert. Ein Beispiel soll dies verdeutlichen - es sei die folgende Konfiguration gegeben:

```
HOSTNAME='fli41'
TEST='${HOSTNAME}'
```

Dann führt der Code:

```
warning "${TEST}"
```

zur Ausgabe von:

```
Warning: ${HOSTNAME}
```

und *nicht* zur Ausgabe von:

```
Warning: fli41
```

In den folgenden Abschnitten wird explizit darauf hingewiesen, unter welchen Umständen Zeichenketten der Variablenersetzung unterliegen.

### Definition eines Dienstes mit einer dazugehörenden Versionsnummer: **provides**

Damit kann z. B. ein OPT deklarieren, dass es einen Drucker-Dienst oder einen Webserver-Dienst bereitstellt. Es kann jeweils nur ein einziges Paket geben, dass einen Dienst bereitstellt. Damit kann man verhindern, dass z. B. zwei Webserver parallel installiert werden, was nahe-liegenderweise nicht gehen würde, da sich die beiden Server um den Port 80 streiten würden. Zusätzlich wird die aktuelle Version des Dienstes angegeben, so dass Weiterentwicklungen Rechnung getragen werden kann. Die Versionsnummer besteht aus zwei- oder drei Zahlen, die durch Punkte voneinander getrennt sind, etwa „4.0“ oder „2.1.23“.

Typischerweise werden Dienste auf OPTs, nicht auf ganze Pakete abgebildet. So besitzt etwa das Paket „tools“ eine ganze Reihe von Programmen, die jeweils ihre eigene **provides**-Anweisung definieren, so sie denn via `OPT_...='yes'` aktiviert sind.

Die Syntax lautet:

```
provides <Name> version <Version>
```

Beispiel aus dem Paket „easycron“:

```
provides cron version 3.10.0
```

Die Versionsnummer sollte vom OPT-Entwickler in der dritten Komponente angehoben werden, wenn lediglich Funktionserweiterungen vorgenommen wurden und die Schnittstelle zum OPT kompatibel geblieben ist. Die Versionsnummer sollte in der ersten oder zweiten Komponente angehoben werden, wenn sich die Schnittstelle in irgendeiner Weise inkompatibel verändert hat (z. B. auf Grund von Variablenumbenennungen, Pfad-Änderungen, verschwundenen oder umbenannten Dienstprogrammen etc.).

**Definition einer Abhängigkeit zu einem Dienst mit einer bestimmten Version: depends**

Benötigt man zur Erbringung der eigenen Funktionalität einen anderen Dienst (z. B. einen Webserver), kann man hiermit diese Abhängigkeit zu einem Dienst mit einer bestimmten Version spezifizieren. Die Version kann zweistellig (z. B. „2.1“) oder dreistellig (z. B. „2.1.11“) angegeben werden, wobei die zweistellige Variante alle Versionen akzeptiert, die ebenfalls so beginnen, während die dreistellige Version nur genau diese angegebene Version akzeptiert. Des Weiteren kann eine Liste von solchen Versionsnummern angegeben werden, falls mehrere Versionen des Dienstes kompatibel mit dem Paket sind.

Die Syntax lautet:

```
depends on <Name> version <Version>+
```

Ein Beispiel: Paket „server“ enthalte:

```
provides server version 1.0.1
```

Sei ein Paket „client“ gegeben. Darin seien folgende `depends`-Anweisungen beispielhaft enthalten:<sup>5</sup>

```
depends on server version 1.0           # OK, '1.0' passt zu '1.0.1'
depends on server version 1.0.1        # OK, '1.0.1' passt zu '1.0.1'
depends on server version 1.0.2        # Fehler, '1.0.2' passt nicht zu '1.0.1'
depends on server version 1.1          # Fehler, '1.1' passt nicht zu '1.0.1'
depends on server version 1.0 1.1      # OK, '1.0' passt zu '1.0.1'
depends on server version 1.0.2 1.1    # Fehler, weder '1.0.2' noch '1.1' passen
                                     # zu '1.0.1'
```

**Kommunikation mit dem Nutzer: warning, error, fatal\_error**

Mit Hilfe dieser drei Funktionen kann man Nutzer warnen, einen Fehler signalisieren oder die Prüfung sofort abbrechen. Die Syntax sieht wie folgt aus:

- `warning "text"`
- `error "text"`
- `fatal_error "text"`

Alle an diese Funktionen übergebenen Zeichenketten-Konstanten unterliegen der [Variablen-ersetzung](#) (Seite 22).

**Zuweisungen**

Benötigt man aus irgendeinem Grund eine temporäre Variable, kann man diese einfach mit „`set var [= value]`“ anlegen. *Die Variable darf kein Konfigurationsvariable sein!*<sup>6</sup> Lässt man den „`= value`“ Teil weg, wird die Variable einfach auf „yes“ gesetzt, so dass man sie hinterher einfach in einer `if`-Anweisung testen kann. Wird ein Zuweisungsteil angegeben, kann hinter

<sup>5</sup>Natürlich nur jeweils eine zur selben Zeit!

<sup>6</sup>Dies ist eine bewusste Entscheidung: Durch `check`-Skripte lässt sich die Benutzerkonfiguration *nicht* verändern.



dem Gleichheitszeichen alles stehen: normale Variablen, indizierte Variablen, Zahlen, Zeichenketten, Versionsnummern.

Zu beachten ist, dass durch diese Zuweisung gleichzeitig der *Typ* der temporären Variablen festgelegt wird. Wird eine Zahl zugewiesen, „merkt“ mkfli4l sich, dass diese Variable eine Zahl enthält, und erlaubt später das Rechnen damit. Versucht man, mit einer anders getypten Variable zu rechnen, wird dies fehlschlagen. Beispiel:

```
set i=1    # OK, i ist eine numerische Variable
set j=i+1  # OK, j ist eine numerische Variable und enthält den Wert 2
set i="1"  # OK, i ist nun eine Zeichenketten-Variable
set j=i+1  # Fehler "You can't use a non-numeric ID in a numeric
           #          context. Check type of operand."
           # --> mit Zeichenketten kann man nicht rechnen!
```

Man kann auch temporäre Arrays (siehe unten) anlegen. Beispiel:

```
set prim_%[1]=2
set prim_%[2]=3
set prim_%[3]=5
warning "${prim_n}"
```

Dabei wird die Anzahl der Elemente in dem Array in der Variable `prim_n` von mkfli4l verwaltet. Der obige Code führt somit zu folgender Ausgabe:

```
Warning: 3
```

Wenn auf der rechten Seite einer Zuweisung eine Zeichenketten-Konstante steht, unterliegt sie zum Zeitpunkt der Zuweisung der [Variablenersetzung](#) (Seite 22). Dies wird im folgenden Beispiel demonstriert. Der Code:

```
set s="a"
set v1="$s" # v1="a"
set s="b"
set v2="$s" # v2="b"
if (v1 == v2)
then
  warning "gleich"
else
  warning "ungleich"
fi
```

produziert die Ausgabe „ungleich“, weil die Variablen `v1` und `v2` bereits während der Zuweisung den aktuellen Inhalt der Variablen `s` ersetzen.

**Hinweis:** Eine in einem Skript gesetzte Variable ist bei der Abarbeitung weiterer Skripte sichtbar – es existiert zur Zeit kein Lokalisierungsprinzip für derart eingeführte Variablen. Da die Reihenfolge, in der die Skripte verschiedener Pakete abgearbeitet wird, nicht definiert ist, sollte man sich nie darauf verlassen, dass Variablen irgendwelche Werte besitzen bzw. von einem anderen Paket übernommen haben.

## Arrays

Will man auf einzelne Elemente einer %-Variablen (eines Arrays) zugreifen, muss man den Original-Namen der Variable, wie er in der `check/<PAKET>.txt`-Datei steht, verwenden, und dabei für jedes „%-Zeichen einen Index mit Hilfe von „*Index*“ anhängen.

Beispiel: Will man auf die Elemente der Variable `PF_USR_CHAIN_%_RULE_%` zugreifen, benötigt man zwei Indizes, weil die Variable zwei „%-Zeichen besitzt. Alle Elemente ausgeben kann man z. B. mit Hilfe des folgenden Codes (die `foreach`-Schleife wird [weiter unten](#) (Seite 31) erläutert):

```
foreach i in pf_usr_chain_n
do
    # nur ein Index nötig, da nur ein '%' im Variablennamen
    set j_n=pf_usr_chain_%_rule_n[i]
    # Achtung: ein
    # foreach j in pf_usr_chain_%_rule_n[i]
    # ist leider nicht möglich, deshalb der Umweg über j_n!
    foreach j in j_n
    do
        # zwei Indizes nötig, da zwei '%' im Variablennamen
        set rule=pf_usr_chain_%_rule_%[i][j]
        warning "Rule $i/$j: ${rule}"
    done
done
```

Mit der folgenden Beispiel-Konfiguration

```
PF_USR_CHAIN_N='2'
PF_USR_CHAIN_1_NAME='usr-chain_a'
PF_USR_CHAIN_1_RULE_N='2'
PF_USR_CHAIN_1_RULE_1='ACCEPT'
PF_USR_CHAIN_1_RULE_2='REJECT'
PF_USR_CHAIN_2_NAME='usr-chain_b'
PF_USR_CHAIN_2_RULE_N='1'
PF_USR_CHAIN_2_RULE_1='DROP'
```

gibt es dann die folgenden Ausgaben:

```
Warning: Rule 1/1: ACCEPT
Warning: Rule 1/2: REJECT
Warning: Rule 2/1: DROP
```

Alternativ kann man direkt über alle Werte des Arrays iterieren, kennt dann allerdings nicht die exakten Indizes der Einträge (was auch nicht immer erforderlich ist):

```
foreach rule in pf_usr_chain_%_rule_%
do
    warning "Rule %{rule}='${rule}'"
done
```

Das produziert mit der Beispiel-Konfiguration von oben die folgenden Ausgaben:

```
Warning: Rule PF_USR_CHAIN_1_RULE_1='ACCEPT'
Warning: Rule PF_USR_CHAIN_1_RULE_2='REJECT'
Warning: Rule PF_USR_CHAIN_2_RULE_1='DROP'
```

An dem zweiten Beispiel sieht man auch schön die Bedeutung der %<Name>-Syntax: Innerhalb der Zeichenkette wird %rule durch den *Namen* der betrachteten Variable ersetzt (also z. B. PF\_USR\_CHAIN\_1\_RULE\_1), während \$rule durch dessen *Inhalt* (also z. B. ACCEPT) ersetzt wird.

### Verschlüsseln eines Passwortes: crypt

Einige Variablen enthalten Passwörter, die nicht im Klartext in der rc.cfg stehen sollen. Diese Variablen können mittels **crypt** verschlüsselt werden und werden damit in das Format überführt, dass auch auf dem Router benötigt wird. Verwendet wird das wie folgt:

```
crypt (<Variable>)
```

Die crypt-Funktion ist die *einzige* Stelle, an der eine Konfigurationsvariable verändert werden kann.

### Abfragen von Eigenschaften einer Datei: stat

stat ermöglicht es, Eigenschaften einer Datei abzufragen. Zur Verfügung gestellt wird im Augenblick lediglich die Größe einer Datei. Wenn man auf Dateien unterhalb des aktuellen Konfigurationsverzeichnis testen will, kann man die interne Variable config\_dir benutzen. Die Syntax lautet:

```
stat (<Dateiname>, <Schlüssel>)
```

Der Aufruf sieht wie folgt aus (wobei die verwendeten Parameter nur Beispiele sind):

```
foreach i in openvpn_%_secret
do
    stat("${config_dir}/etc/openvpn/$i.secret", keyfile)
    if (keyfile_res != "OK")
    then
        error "OpenVPN: missing secretfile <config>/etc/openvpn/$i.secret"
    fi
done
```

In dem Beispiel wird geprüft, ob eine Datei im aktuellen Konfigurationsverzeichnis vorhanden ist. Wenn also OPENVPN\_1\_SECRET='test' in der Konfigurationsdatei gesetzt wird, prüft die Schleife im ersten Durchlauf, ob im aktuellen Konfigurationsverzeichnis die Datei etc/openvpn/test.secret vorhanden ist.

Nach dem Aufruf sind zwei Variablen definiert:

- <Schlüssel>\_res: Resultat des Systemaufrufs stat() („OK“, wenn Systemruf erfolgreich, sonst Fehlermeldung des Systemaufrufs)
- <Schlüssel>\_size: Größe der Datei

Das könnte dann z. B. so aussehen:

```

stat ("unix/Makefile", test)
if ("${test_res}" == "OK")
then
    warning "test_size = ${test_size}"
else
    error "Error '${test_res}' while trying to get size of 'unix/Makefile'"
fi

```

Ein als Zeichenketten-Konstante übergebener Dateiname unterliegt der [Variablenersetzung](#) (Seite 22).

### Durchsuchen von Dateien: **fgrep**

Wenn Sie in einer Datei per „grep“<sup>7</sup> suchen wollen, steht Ihnen das **fgrep**-Kommando zur Verfügung. Die Syntax lautet:

```
fgrep (<Dateiname>, <RegEx>)
```

Wenn die Datei <Dateiname> nicht existiert wird **mkfli41** mit einem fatalen Fehler beendet! Wenn Sie also nicht sicher sind, ob die Datei immer vorhanden ist, testen Sie die Existenz von <Dateiname> vorher mit **stat** ab. Nach dem Aufruf von **fgrep** steht Ihnen das Suchresultat in dem Array **FGREP\_MATCH\_%** zur Verfügung, wobei der Index *x* wie üblich von eins bis **FGREP\_MATCH\_N** reicht. **FGREP\_MATCH\_1** verweist dabei auf den gesamten Bereich der Zeile, auf den der reguläre Ausdruck gepasst hat, während **FGREP\_MATCH\_2** bis **FGREP\_MATCH\_N** den jeweils *n-1*-ten geklammerten Teil beinhalten.

Ein erstes einfaches Beispiel soll die Verwendung demonstrieren. Die Datei **opt/etc/shells** enthält die Zeile:

```
/bin/sh
```

Der folgende Code

```

fgrep("opt/etc/shells", "~/(.)(.*)/")
foreach v in FGREP_MATCH_%
do
    warning "%v='${v}'"
done

```

produziert die folgende Ausgabe:

```

Warning: FGREP_MATCH_1='/bin/'
Warning: FGREP_MATCH_2='b'
Warning: FGREP_MATCH_3='in'

```

Der reguläre Ausdruck hat (nur) auf „/bin/“ gepasst, deshalb steht auch (nur) dieser Teil der Zeile in der Variable **FGREP\_MATCH\_1**. Der erste geklammerte Teil im Ausdruck passt auf das erste Zeichen hinter dem ersten „/“, deshalb steht auch nur „b“ in **FGREP\_MATCH\_2**. Der zweite geklammerte Teil umfasst den Rest hinter den „b“ bis zum letzten „/“, somit steht „in“ in der Variable **FGREP\_MATCH\_3**.

Das folgende zweite Beispiel demonstriert eine praxisnahe Verwendung von **fgrep** an einem Beispiel aus der **check/base.ext**. Hier wird getestet, ob alle in der **PF\_FORWARD\_x** angegebenen **tmpl:-**Referenzen vorhanden sind:

---

<sup>7</sup> „grep“ ist ein auf Unix-Betriebssystemen verbreitetes Kommando zum Filtern von Textströmen.

```

foreach n in pf_forward_n
do
  set rule=pf_forward_%[n]
  if (rule =~ "tpl:([[:space:]]+)" )
  then
    foreach m in match_%
    do
      stat("$config_dir/etc/fwrules.tpl/$m", tplfile)
      if(tplfile_res == "OK")
      then
        add_to_opt "etc/fwrules.tpl/$m"
      else
        stat("opt/etc/fwrules.tpl/$m", tplfile)
        if(tplfile_res == "OK")
        then
          add_to_opt "etc/fwrules.tpl/$m"
        else
          fgrep("opt/etc/fwrules.tpl/templates", "^$m[[:space:]]+")
          if (fgrep_match_n == 0)
          then
            error "Can't find tpl:$m for PF_FORWARD_${n}='$rule'!"
          fi
        fi
      fi
    done
  fi
done

```

Sowohl ein als Zeichenketten-Konstante übergebener Dateiname als auch als Zeichenketten-Konstante übergebener regulärer Ausdruck unterliegen der [Variablenersetzung](#) (Seite 22).

### Auseinandernehmen von Parametern: `split`

Oftmals werden Variablen mit mehreren Parametern belegt, die dann in Startup-Skripten erst wieder auseinandergenommen werden. Will man diese bereits vorher auseinandernehmen und Tests auf ihnen durchführen, nimmt man `split`. Die Syntax lautet:

```
split (<Zeichenkette>, <Array>, <Trennzeichen>)
```

Die Zeichenkette kann durch eine Variable oder direkt als Konstante angegeben werden. `mkfli4l` zerlegt ihn an den Stellen, an denen das Trennzeichen auftaucht, und erzeugt pro Teil ein Element des Arrays. Über diese Elemente kann man dann hinterher iterieren und Prüfungen vornehmen. Steht zwischen zwei Trennzeichen nichts, wird ein Array-Element mit einer leeren Zeichenkette als Wert erzeugt. Ausnahme ist „`:`“: Hier werden alle Leerzeichen konsumiert und keine leeren Variablen erzeugt.

Sollen die bei der Zerlegung entstandenen Elemente in einem numerischen Kontext verwendet werden (z.B. als Indizes), muss das beim Aufruf von `split` spezifiziert werden. Das geschieht durch das zusätzliche Attribut „`numeric`“. Der Aufruf sieht dann wie folgt aus:

```
split (<Zeichenkette>, <Array>, <Trennzeichen>, numeric)
```

Ein Beispiel:

```

set bar="1.2.3.4"
split (bar, tmp_%, '.', numeric)
foreach i in tmp_%
do
    warning "%i = $i"
done

```

Die produzierte Ausgabe ist:

```

Warning: TMP_1 = 1
Warning: TMP_2 = 2
Warning: TMP_3 = 3
Warning: TMP_4 = 4

```

**Hinweis:** Wenn die „numeric“-Variante verwendet wird, dann prüft `mkfli4l` zum Zeitpunkt der Zerlegung *nicht*, ob die Teil-Zeichenketten auch wirklich numerisch sind! Bei einer späteren Verwendung in einem numerischen Kontext (etwa beim Addieren) löst `mkfli4l` jedoch einen fatalen Fehler aus, wenn eine solche Variable doch nicht numerisch ist. Beispiel:

```

set bar="a.b.c.d"
split (bar, tmp_%, '.', numeric)
# Fehler: invalid number 'a'
set i=tmp_%[1]+1

```

Eine an `split` im ersten Parameter übergebene Zeichenketten-Konstante unterliegt der [Variablenersetzung](#) (Seite 22).

### Hinzufügen von Dateien zum Archiv: `add_to_opt`

Mit der Funktion `add_to_opt` können zusätzliche Dateien ans `opt`- oder `rootfs`-Archiv angehängt werden. Es können dabei *alle* Dateien unterhalb von `opt/` oder aus dem Konfigurationsverzeichnis ausgewählt werden. Eine Beschränkung nur auf die Dateien, die mit einem bestimmten Paket geliefert werden, gibt es nicht. Liegt eine Datei sowohl unter `opt/` als auch im Konfigurationsverzeichnis im gleichen Pfad, bevorzugt `add_to_opt` die Dateien aus dem Konfigurationsverzeichnis. Die Funktion `add_to_opt` wird in der Regel dann eingesetzt, wenn komplexe logische Regeln darüber entscheiden, ob und welche Dateien in das Archiv aufgenommen werden müssen.

Die Syntax sieht wie folgt aus:

```
add_to_opt <Datei> [<Flags>]
```

Die Flags sind optional. Es gelten die in Tabelle 1.2 aufgeführten Standard-Werte, falls keine Flags angegeben sind.

Es folgt ein Beispiel aus dem Paket „sshd“:

```

if (opt_sshd)
then
    foreach pkf in sshd_public_keyfile_%
    do
        stat("$config_dir/etc/ssh/$pkf", publickeyfile)
        if(publickeyfile_res == "OK")

```

```

then
    add_to_opt "etc/ssh/$pkf" "mode=400 flags=utxt"
else
    error "sshd: missing public keyfile %pkf=$pkf"
fi
done
fi

```

Mit **stat** (Seite 27) wird zunächst geprüft, ob die Datei im Konfigurationsverzeichnis existiert. Ist die Datei vorhanden, wird sie ans Archiv angehängt, andernfalls bricht **mkfli4l** mit einer entsprechenden Fehlermeldung ab.

**Hinweis:** Auch bei **add\_to\_opt** **prüft** (Seite 11) **mkfli4l** zuerst, ob die zu kopierende Datei im Konfigurationsverzeichnis zu finden ist.

Sowohl ein als Zeichenketten-Konstante übergebener Dateiname als auch als Zeichenketten-Konstante übergebene Flags unterliegen der **Variablenersetzung** (Seite 22).

## Kontrollfluss

```

if (expr)
then
    statement
else
    statement
fi

```

Eine klassische Fallunterscheidung, wie man sie kennt. Ist die Bedingung wahr, wird der **then**-Teil ausgeführt, ist die Bedingung falsch, wird der **else**-Teil ausgeführt.

Will man Tests über Array-Variablen durchführen, muss man jede einzelne Variable testen. Dazu gibt es die **foreach**-Schleife in zwei Varianten.

1. Iterieren über Array-Variablen:

```

foreach <Laufvariable> in <Array-Variable>
do
    <Anweisung>
done

foreach <Laufvariable> in <Array-Variable-1> <Array-Variable-2> ...
do
    <Anweisung>
done

```

Diese Schleife iteriert über alle angegebenen Array-Variablen, jeweils angefangen beim ersten Element bis hin zum letzten; die Anzahl der Elemente im Array wird dabei der dem Array zugeordneten N-Variable entnommen. Die Laufvariable nimmt dabei die jeweiligen Werte der Array-Variablen an. Zu beachten ist dabei, dass bei optionalen Array-Variablen, die in der Konfiguration nicht vorhanden sind, ein leeres Element generiert wird. Unter Umständen muss das im Skript berücksichtigt werden, was man z.B. wie folgt tun kann:

```
foreach i in template_var_opt_%
do
  if (i != "")
  then
    warning "%i is present (%i='$i')"
```

```
  else
    warning "%i is undefined (empty)"
  fi
done
```

Wie man auch am Beispiel erkennen kann, lässt sich der *Name* der jeweiligen Array-Variablen durch die %<Laufvariable>-Konstruktion ermitteln.

Die Anweisung in der Schleife kann eine der oben beschriebenen Kontrollelemente oder Funktionen (`if`, `foreach`, `provides`, `depends`, ...) sein.

Will man auf genau ein Element eines Arrays zugreifen, kann man dieses mittels der Syntax <Array>[<Index>] ansprechen. Der Index kann dabei eine normale Variable, eine Zahlenkonstante oder wiederum ein indiziertes Array sein.

## 2. Iterieren über N-Variablen:

```
foreach <Laufvariable> in <N-Variable>
do
  <Anweisung>
done
```

Diese Schleife läuft von 1 bis zum Wert, der in der N-Variable steht. Man kann die Laufvariable dazu benutzen, um Array-Variablen zu indizieren. Will man also nicht nur über eine Array-Variable iterieren, sondern über mehrere gleichzeitig, die alle durch *dieselben* N-Variable kontrolliert werden, nimmt man diese Variante der Schleife und verwendet die Laufvariable zum Indizieren mehrerer Array-Variablen. Beispiel:

```
foreach i in host_n
do
  set name=host_%_name[i]
  set ip4=host_%_ip4[i]
  warning "$i: name=$name ip4=$ip4"
```

```
done
```

Das ergibt bei entsprechend gefüllten HOST\_%\_NAME- und HOST\_%\_IP4-Arrays beispielsweise:

```
Warning: 1: name=berry ip4=192.168.11.226
Warning: 2: name=fence ip4=192.168.11.254
Warning: 3: name=sandbox ip4=192.168.12.254
```

## Ausdrücke

Ausdrücke verknüpfen Werte und Operatoren zu einem neuen Wert. Ein Wert kann dabei eine gewöhnliche Variable, ein Array-Element oder eine Konstante (Zahl, Zeichenkette oder Versionsnummer) sein. Alle Zeichenketten-Konstanten, die in Ausdrücken auftreten, unterliegen der [Variablenersetzung](#) (Seite 22).



Operatoren erlauben so gut wie alles, was man von einer Programmiersprache gewöhnt ist. Ein Test auf die Gleichheit zweier Variablen könnte also so aussehen:

```
var1 == var2
"$var1" == "$var2"
```

Zu beachten ist dabei, dass der Vergleich in Abhängigkeit vom Typ der Variable erfolgt, der in `check/<PAKET>.txt` festgelegt wurde. Ist eine der beiden Variablen **numerisch** (Seite 21), erfolgt der Vergleich auf numerischer Basis, d. h. die Zeichenketten werden in Zahlen umgewandelt und dann verglichen. Sonst erfolgt der Vergleich auf Zeichenketten-Basis; ein Vergleich `"05" == "5"` ergibt „falsch“, ein Vergleich `"18" < "9"` ergibt „wahr“ auf Grund der lexikographischen Ordnung auf Zeichenketten: die Ziffer „1“ liegt vor der Ziffer „9“ im zugrunde liegenden ASCII-Zeichensatz.

Für den Vergleich von Versionen wird das Hilfskonstrukt `numeric(version)` eingeführt, welches den numerischen Wert für eine Versionsnummer für Vergleichszwecke bestimmt. Dabei gilt:

```
numeric(version) := major * 10000 + minor * 1000 + sub
```

wobei „major“ die erste Komponente der Versionsnummer darstellt, „minor“ die zweite und „sub“ die dritte; fehlt „sub“, entfällt der Term in der obigen Summe einfach (oder anders ausgedrückt, für „sub“ wird null angenommen).

Eine vollständige Auflistung aller Ausdrücke ist in Tabelle 1.3 zu finden. Dabei steht „val“ für einen beliebig getypten Wert, „number“ für einen numerischen Wert und „string“ für eine Zeichenkette.

Tabelle 1.3: Logische Ausdrücke

Ausdruck	wahr wenn
<code>id</code>	<code>id == „yes“</code>
<code>val == val</code>	identisch getypte Werte sind gleich
<code>val != val</code>	identisch getypte Werte sind ungleich
<code>val == number</code>	numerischer Wert von <code>val == number</code>
<code>val != number</code>	numerischer Wert von <code>val != number</code>
<code>val &lt; number</code>	numerischer Wert von <code>val &lt; number</code>
<code>val &gt; number</code>	numerischer Wert von <code>val &gt; number</code>
<code>val == version</code>	<code>numeric(val) == numeric(version)</code>
<code>val &lt; version</code>	<code>numeric(val) &lt; numeric(version)</code>
<code>val &gt; version</code>	<code>numeric(val) &gt; numeric(version)</code>
<code>val =~ string</code>	regulärer Ausdruck in <code>string</code> auf <code>val</code> passt
<code>( expr )</code>	Ausdruck in Klammern ist wahr
<code>expr &amp;&amp; expr</code>	beide Ausdrücke sind wahr
<code>expr    expr</code>	mind. einer der beiden Audrücke ist wahr
<code>copy_pending(id)</code>	siehe Beschreibung
<code>samenet (string1, string2)</code>	<code>string1</code> das gleiche netz wie <code>string2</code> beschreibt
<code>subnet (string1, string2)</code>	<code>string1</code> ein Subnetz von <code>string2</code> beschreibt

## Match-Operator

Mit dem Match-Operator `=~` kann man prüfen, ob ein regulärer Ausdruck auf den Wert einer Variable passt. Weiterhin kann man den Operator auch nutzen, um Teilausdrücke aus einer Variablen zu extrahieren. Nach erfolgreichem Anwenden eines regulären Ausdrucks auf eine Variable enthält das Array `MATCH_%` die gefundenen Teile. Das könnte z. B. wie folgt aussehen:

```
set foo="foobar12"
if ( foo =~ "(foo)(bar)([0-9]*)" )
then
    foreach i in match_%
    do
        warning "match %i: $i"
    done
fi
```

Ein `mkfli41`-Aufruf führt dann zu folgender Ausgabe:

```
Warning: match MATCH_1: foo
Warning: match MATCH_2: bar
Warning: match MATCH_3: 12
```

Bei Verwendung von `=~` kann Bezug auf alle existierenden regulären Ausdrücke genommen werden. Will man z. B. prüfen, ob ein PCMCIA-Ethernet-Treiber ausgewählt wurde, ohne dass `OPT_PCMCIA` auf „yes“ gesetzt wurde, könnte das wie folgt aussehen:

```
if (!opt_pcmcia)
then
    foreach i in net_drv_%
    do
        if (i =~ "(RE:PCMCIA_NET_DRV)$")
        then
            error "If you want to use ..."
        fi
    done
fi
```

Wie in dem Beispiel demonstriert wird, ist es wichtig, den regulären Ausdruck mit Hilfe von `^` und `$` zu *verankern*, wenn man den Ausdruck auf die *gesamte* Variable anwenden will. Ansonsten liefert der Match-Ausdruck schon „wahr“, wenn nur ein *Teil* der Variable vom regulären Ausdruck abgedeckt wird, was in diesem Fall sicherlich nicht erwünscht ist.

## Prüfen, ob in Abhängigkeit vom Wert einer Variable eine Datei kopiert wurde:

### `copy_pending`

Mit den im Check-Prozess gewonnenen Informationen prüft die Funktion `copy_pending`, ob in Abhängigkeit vom Wert einer Variable eine Datei kopiert wurde oder nicht. Das kann man verwenden, um z. B. zu testen, ob der vom Nutzer angegebene Treiber auch wirklich existiert und kopiert wurde. `copy_pending` akzeptiert den zu prüfenden Namen in Form einer Variablen oder einer Zeichenkette.<sup>8</sup> `copy_pending` prüft dazu, ob

<sup>8</sup>Wie eingangs beschrieben unterliegt die Zeichenkette der Variablenersetzung, so dass man z. B. mittels einer [foreach-Schleife](#) (Seite 31) und der [%<Name>-Ersetzung](#) (Seite 22) alle Elemente eines Arrays prüfen kann.

- die Variable aktiv ist (wenn sie von einem OPT abhängt, muss dieses auf „yes“ gesetzt sein),
- die Variable in einer `opt/<PAKET>.txt`-Datei referenziert wurde, und
- ob in Abhängigkeit vom aktuellen Wert eine Datei kopiert wurde.

Dabei liefert `copy_pending` „wahr“ zurück, wenn im letzten Schritt festgestellt wurde, dass *keine* Datei kopiert wurde, das Kopieren also somit noch aussteht (also „pending“ ist).

Ein kleines Beispiel für die Anwendung all dieser Funktionen findet man in `check/base.ext`:

```
foreach i in net_drv_%
do
    if (copy_pending("%i"))
    then
        error "No network driver found for %i='$i', check config/base.txt"
    fi
done
```

Hier werden alle Elemente des Arrays `NET_DRV_%` angemackert, für die keine Kopieraktion vorgenommen wurde, für die also in der `opt/base.txt` kein entsprechender Eintrag existiert.

### Vergleich von Netzwerkadressen: `samenet` und `subnet`

Zum Prüfen von Routen benötigt man ab und zu einen Test, ob zwei Netzwerke identisch sind oder eines ein Subnetz eines anderen ist. Dazu gibt es die beiden Funktionen `samenet` und `subnet`. Dabei liefert

```
samenet (netz1, netz2)
```

„wahr“, wenn beide Netze identisch sind, und

```
subnet (netz1, netz2)
```

gibt „wahr“ zurück, wenn „netz1“ ein Subnetz von „netz2“ ist.

### Erweitern der Kernel-Kommandozeile

Ist ein OPT gezwungen, dem Kernel andere Boot-Parameter zu übergeben, so musste früher die Variable `KERNEL_BOOT_OPTION` geprüft werden, ob der nötige Parameter enthalten war, und ggf. eine Warnung oder eine Fehlermeldung ausgegeben werden. Mit der internen Variable `KERNEL_BOOT_OPTION_EXT` kann man nötige, aber fehlende Optionen direkt im ext-Skript ergänzen. Ein Beispiel aus der `check/base.ext`:

```
if (powermanagement =~ "apm.*|none")
then
    if ( ! kernel_boot_option =~ "acpi=off")
    then
        set kernel_boot_option_ext="${kernel_boot_option_ext} acpi=off"
    fi
fi
```

Damit wird „acpi=off“ an den Kernel übergeben, falls keine Energieverwaltung oder welche vom Typ „APM“ gewünscht ist.

### 1.3.8 Unterstützung verschiedener Kernelversionslinien

Verschiedene Kernelversionslinien unterscheiden sich häufig in einigen Details:

- es stehen andere Treiber zur Verfügung, einige sind weggefallen, andere hinzugekommen
- die Module heißen teilweise einfach anders
- die Modul-Abhängigkeiten sehen anders aus
- die Module liegen woanders

Diese Unterschiede werden zum großen Teil durch `mkfli4l` automatisch behandelt. Um die zur Verfügung stehenden Module zu beschreiben, kann man zum einen die zur Prüfung verwendeten Prüfungen in Abhängigkeit von der Version erweitern ([bedingte reguläre Ausdrücke](#) (Seite 17)), und zum anderen erlaubt `mkfli4l` *versionsabhängige* `opt/<PAKET>.txt`-Dateien. Dies heißen dann `opt/<PAKET>_<Kernel-Version>.txt`, wobei die Komponenten der Kernel-Version durch Unterstriche voneinander getrennt werden. Ein Beispiel: Das Paket „base“ enthält die folgenden Dateien im `opt`-Verzeichnis:

- `base.txt`
- `base_3_18.txt`
- `base_3_19.txt`

Die erste Datei (`base.txt`) wird *immer* verarbeitet. Die anderen beiden Dateien werden nur verarbeitet, wenn die Kernelversion „3.18(.)“ bzw. „3.19(.)“ lautet. Wie man sieht, können Versionskomponenten im Dateinamen weggelassen werden, wenn man eine ganze Gruppe von Kernen „erschlagen“ möchte. Unter Annahme von `KERNEL_VERSION='3.18.9'` werden für ein Paket `<PAKET>` die folgenden Dateien (sofern vorhanden) eingelesen und verarbeitet:

- `<PAKET>.txt`
- `<PAKET>_3.txt`
- `<PAKET>_3_18.txt`
- `<PAKET>_3_18_9.txt`

### 1.3.9 Dokumentation

Die Dokumentation wird in den Dateien

- `doc/<SPRACHE>/opt/<PAKET>.txt`
- `doc/<SPRACHE>/opt/<PAKET>.html`

abgelegt. Die HTML-Dateien können auch aufgeteilt werden, d. h. für jedes enthaltene OPT eine. Dann muss trotzdem eine `<PAKET>.html` angelegt werden, die auf die anderen Dateien verweist. Änderungen sollten in folgenden Dateien dokumentiert werden:

- `changes/<PAKET>.txt`

Die gesamte Text-Dokumentation darf keine Tabulatoren enthalten und muss nach spätestens 79 Zeichen einen harten Zeilenumbruch haben. Die stellt sicher, dass die Dokumentation auch mit einem Editor ohne automatischen Zeilenumbruch richtig gelesen werden kann.

Wer mag kann auch eine Dokumentation im  $\text{\LaTeX}$ -Format erstellen und daraus dann HTML- und PDF-Fassungen erzeugen. Als Beispiel kann die Dokumentation von fli4l dienen. Einen Rahmen für die Dokumentation und die minimal benötigten  $\text{\LaTeX}$ -Macros kann man im Paket „template“ finden. Eine kurze Beschreibung ist in den folgenden Unterabschnitten zu finden.

Die fli4l-Dokumentation steht zur Zeit in den folgenden Sprachen zur Verfügung: deutsch, englisch ( $\langle\text{SPRACHE}\rangle = \text{„english“}$ ) und französisch ( $\langle\text{SPRACHE}\rangle = \text{„french“}$ ). Es steht einem Paket-Entwickler jedoch frei, sein Paket in beliebigen Sprachen zu dokumentieren. Im Sinne der Verständlichkeit wird jedoch empfohlen, eine Dokumentation in deutsch und/oder englisch (idealerweise in beiden Sprachen) anzufertigen.

### Voraussetzungen für die Erstellung einer $\text{\LaTeX}$ -Dokumentation

Zum Erstellen der Dokumentation aus  $\text{\LaTeX}$ -Quellen gibt es folgende Anforderungen an die Umgebung:

- Linux/OS X-Umgebung: Zur einfachen Erzeugung gibt es ein Makefile, mit dem alle weiteren Aufrufe automatisiert sind (Cygwin müsste auch funktionieren, wird aber nicht vom fli4l-Team getestet)
- LaTeX2HTML für die HTML-Version
- natürlich  $\text{\LaTeX}$  (empfohlen wird „TeX Live“ für Linux/OS X und „MiKTeX“ für Microsoft Windows) mit dem „pdf $\text{\TeX}$ “-Programm und folgenden  $\text{\TeX}$ -Paketen:
  - aktuelles KOMA-Skript (mindestens Version 2)
  - alle notwendigen Pakete für pdf $\text{\TeX}$
  - ausgepacktes Dokumentationspaket für fli4l, welches die benötigten Makefiles und  $\text{\TeX}$ -Stile bereitstellt

### Dateinamen

Die Dateien der Dokumentation werden nach folgendem Schema benannt:

**$\langle\text{PAKET}\rangle\_main.tex$ :** Diese Datei enthält den Hauptteil der Dokumentation.  $\langle\text{PAKET}\rangle$  steht hier für den Namen des Pakets, das beschrieben werden soll (in Kleinbuchstaben).

**$\langle\text{PAKET}\rangle\_appendix.tex$ :** Sollen zu diesem Paket noch weitere Anmerkungen im Anhang hinzugefügt werden, so werden diese hier abgelegt.

Diese Dateien werden im Verzeichnis `fli4l/ $\langle\text{PAKET}\rangle$ /doc/ $\langle\text{SPRACHE}\rangle$ /tex/ $\langle\text{PAKET}\rangle$`  abgelegt. Für das Paket „sshd“ sieht das z. B. wie folgt aus:

```
$ ls fli4l/doc/deutsch/tex/sshd/  
Makefile sshd_appendix.tex  sshd_main.tex  sshd.tex
```

Das Makefile ist für die Generierung der Dokumentation verantwortlich, die `sshd.tex`-Datei stellt einen Rahmen für die eigentliche Dokumentation und den Anhang bereit, der sich in den anderen beiden Dateien befindet. Ansehen kann man sich das am Beispiel der Dokumentation des „template“-Pakets.

## L<sup>A</sup>T<sub>E</sub>X-Grundlagen

L<sup>A</sup>T<sub>E</sub>X arbeitet ähnlich wie HTML „Tag-orientiert“, nur dass die Tags hier „Kommandos“ heißen und folgendes Format aufweisen: `\kommando` bzw. `\begin{umgebung} ... \end{umgebung}`

Nach Möglichkeit sollte man mit Hilfe von Kommandos eher die *Bedeutung* des jeweiligen Textes auszeichnen und weniger dessen *Darstellung*. Es ist also vorteilhaft, z. B.

```
\warning{Bitte_nicht_..._tun}
statt
\emph{Bitte_nicht_..._tun}
zu verwenden.
```

Jedes Kommando bzw. jede Umgebung kann noch weitere Parameter aufnehmen, die mit `\kommando{parameter1}{parameter2}{parameterN}` geschrieben werden.

Manche Kommandos haben optionale Parameter, die in eckigen (statt geschweiften) Klammern stehen: `\kommando[optionalerParameter]{parameter1} ...`. Dabei kommt im Normalfall nur ein optionaler Parameter vor, in seltenen Fällen aber auch mehrere.

Einzelne Absätze werden im Dokument durch Leerzeilen getrennt. Innerhalb dieser Absätze nimmt L<sup>A</sup>T<sub>E</sub>X selbst den Zeilenumbruch und die Worttrennung vor.

Folgende Buchstaben haben eine spezielle Bedeutung in L<sup>A</sup>T<sub>E</sub>X und müssen, sollten sie in normalem Text vorkommen, mit einem vorangestellten `\` maskiert werden: `# $ & _ % { }`. „~“ und „^“ müssen wie folgt geschrieben werden: `\verb?~? \verb?^?`

Die wichtigsten L<sup>A</sup>T<sub>E</sub>X-Kommandos werden in der Dokumentation des „template“-Pakets verwendet und erklärt.

### 1.3.10 Dateiformate

Alle Textdateien (sowohl Dokumentation als auch Skripte, die später auf dem Router liegen) müssen im DOS-Dateiformat, also mit CR/LF statt nur LF am Zeilenende in das Paket gelegt werden. Dadurch wird erreicht, dass Windows-Nutzer die Dokumentation auch mit „notepad“ lesen können und durch eine Änderung eines Skripts unter Windows das Ganze später auf dem Router trotzdem lauffähig bleibt. Die Skripte werden beim Bauen der Archive in das auf dem Router benötigte Format konvertiert (siehe die Beschreibung der Flags in Tabelle 1.2).

### 1.3.11 Entwickler-Dokumentation

Sollte ein Programm aus dem Paket eine neue Schnittstelle definieren, die andere Programme nutzen können, so ist die Dokumentation dieser Schnittstelle in einer separaten Dokumentation unter `doc/dev/<PAKET>.txt` abzulegen.

### 1.3.12 Client-Programme

Sollte ein Paket zusätzliche Client-Programme mitliefern, so sind diese im Verzeichnis `windows/` für Windows-Clients und im Verzeichnis `unix/` für Unix- und Linux-Clients abzulegen.

### 1.3.13 Quellcode

Angepasste Programme und Quellcodes können im Verzeichnis `src/<PAKET>/` beigelegt werden. Sollen die Programme genauso wie die restlichen fli4l-Programme gebaut werden, bitte einen Blick in die Dokumentation des „src“-Pakets (Seite ??) werfen.

### 1.3.14 Weitere Dateien

Alle Dateien, die nachher auf dem Router liegen, werden unter `opt/` abgelegt. Dabei liegen unter:

- `opt/etc/boot.d/` und `opt/etc/rc.d/` Skripte, die beim Starten des Systems ausgeführt werden sollen
- `opt/etc/rc0.d/` Skripte, die beim Herunterfahren des Systems ausgeführt werden
- `opt/etc/ppp/` Skripte, die beim Einwählen und Auflegen ausgeführt werden
- `opt/` die ausführbaren Programme und sonstige Dateien entsprechend ihrer Positionen im Dateisystem (d. h. die Datei `opt/bin/busybox` wird später auf dem Router im Verzeichnis `/bin` liegen)

Die Skripte in `opt/etc/boot.d/`, `opt/etc/rc.d/` und `opt/etc/rc0.d/` werden wie folgt benannt:

```
rc<nummer>.<name>
```

Die Nummer entscheidet über die Reihenfolge der Ausführung, der Name gibt einen Hinweis darauf, welches Programm/Paket von diesem Skript behandelt wird.

## 1.4 Allgemeine Skript-Erstellung auf fli4l

Hier folgt jetzt *keine* allgemeine Einführung in Shell-Skripte, das kann jeder im Internet selber nachlesen, es wird nur auf die spezielle Gegebenheiten bei fli4l eingegangen. Informationen dazu gibt es in den diversen Unix-/Linux-Hilfeseiten. Folgende Links können als Einstiegspunkte zu diesem Thema dienen:

- Einführung in Shell-Skripte:
  - <http://cip.physik.uni-freiburg.de/main/howtos/sh.php>
- Hilfeseiten online:
  - <http://linux.die.net/>
  - <http://heapsort.de/man2web>
  - <http://man.he.net/>
  - [http://www.linuxcommand.org/superman\\_pages.php](http://www.linuxcommand.org/superman_pages.php)

### 1.4.1 Aufbau

In der Unix-Welt ist es nötig, ein Skript mit dem Namen des Interpreters zu beginnen, daher steht in der ersten Zeile:

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

Damit man später leichter erkennen kann, was ein Skript macht und wer es geschrieben hat, sollte jetzt ein kurzer Header folgen, in etwa so:

```
#-----
# /etc/rc.d/rc500.dummy - start my cool dummy server
#
# Creation:      19.07.2001  Toller Hecht <toller-hecht@example.net>
# Last Update:  11.11.2001  Süße Maus <suesse-maus@example.net>
#-----
```

Nun kann das eigentliche Skript folgen...

### 1.4.2 Umgang mit Konfigurationsvariablen

Pakete werden über die Datei `config/<PACKAGE>.txt` konfiguriert. Die darin enthaltenen und [aktiven Variablen](#) (Seite 12) werden beim Erzeugen des Boot-Mediums in die Datei `rc.cfg` übernommen. Beim Booten des Routers wird diese Datei eingelesen, bevor irgend ein rc-Skript (Skripte unter `/etc/rc.d/`) gestartet wird. Diese Skripte können dadurch auf alle Konfigurationsvariablen einfach durch `$<Variablenname>` zugreifen.

Benötigt man Werte von Konfigurationsvariablen auch noch nach dem Booten, dann kann man sie aus der `/etc/rc.cfg` extrahieren, in welche während des Bootens die Konfiguration des Boot-Mediums geschrieben wurde. Möchte man beispielsweise den Wert der Variable `OPT_DNS` in einem Skript auslesen, so kann man dies folgendermaßen tun:

```
eval $(grep "^OPT_DNS=" /etc/rc.cfg)
```

Das funktioniert auch mit mehreren Variablen effizient (d. h. mit nur einem Aufruf des `grep`-Programms):

```
eval $(grep "^\(HOSTNAME\|DOMAIN_NAME\|OPT_DNS\|DNS_LISTEN_N\)=" /etc/rc.cfg)
```

### 1.4.3 Persistente Speicherung von Daten

Gelegentlich benötigt ein Paket die Möglichkeit, Daten persistent abzulegen, die also einen Neustart des Routers überleben. Dazu existiert die Funktion `map2persistent`, die von einem Skript in `/etc/rc.d/` aufgerufen werden kann. Sie erwartet eine Variable, die einen Pfad enthält, und ein Unterverzeichnis. Die Idee ist, dass die Variable entweder einen tatsächlichen Pfad beschreibt – dann wird dieser Pfad auch genommen, denn der Nutzer hat ihn so gewünscht, oder die Zeichenkette „auto“ – dann wird unterhalb eines Verzeichnisses auf einem persistenten Medium ein entsprechendes Unterverzeichnis gemäß dem zweiten Parameter erzeugt. Die Funktion liefert das Resultat in eben der Variable zurück, deren Name im ersten Parameter übergeben wurde.

Ein Beispiel soll dies verdeutlichen. Sei `VBOX_SPOOLPATH` eine Variable, die einen Pfad oder die Zeichenkette „auto“ enthält. Dann führt der Aufruf

```
begin_script VBOX "Configuring vbox ..."
[...]
map2persistent VBOX_SPOOLPATH /spool
[...]
end_script
```

dazu, dass die Variable `VBOX_SPOOLPATH` entweder gar nicht verändert wird (falls sie einen Pfad enthält), oder dass sie durch den Pfad `/var/lib/persistent/vbox/spool` ersetzt wird (falls sie die Zeichenkette „auto“ enthält). Dabei verweist<sup>9</sup> `/var/lib/persistent` auf ein Ver-

<sup>9</sup>mit Hilfe eines so genannten „bind“-Mounts



zeichnis auf einem beschreibbaren und nicht flüchtigen Speichermedium, und <SCRIPT> stellt das aufrufende Skript in Kleinbuchstaben dar (dieser Name wird vom ersten Argument des `begin_script-Aufrufs` (Seite 41) abgeleitet). Falls kein geeignetes Medium existieren sollte (was durchaus sein kann), ist `/var/lib/persistent` ein Verzeichnis in der RAM-Disk.

Zu beachten ist, dass der Pfad, der von `map2persistent` zurückgegeben wird, *nicht* automatisch erzeugt wird – das muss der Aufrufer selbst erledigen (etwa durch einen Aufruf von `mkdir -p <Pfad>`).

In der Datei `/var/run/persistent.conf` kann nachgeschaut werden, ob die persistente Speicherung von Daten möglich ist. Beispiel:

```
. /var/run/persistent.conf
case $SAVETYPE in
persistent)
    echo "Persistente Speicherung möglich!"
    ;;
transient)
    echo "Persistente Speicherung NICHT möglich!"
    ;;
esac
```

#### 1.4.4 Fehlersuche

Bei Start-Skripten ist es oft sinnvoll, diese bei Bedarf im Debug-Modus der Shell laufen zu lassen, um festzustellen, wo „der Wurm drin ist“. Dazu wird am Anfang und am Ende folgendes eingefügt:

```
begin_script <OPT-Name> "start message"
<script code>
end_script
```

Im normalen Betrieb erscheint jetzt beim Start des Skriptes der angegebene Text und am Ende der gleiche Text mit einem vorangestellten „finished“.

Will man die Skripte debuggen, muss man zwei Dinge tun:

1. Man muss `DEBUG_STARTUP` (Seite ??) auf „yes“ setzen.
2. Man muss das Debuggen für dieses OPT aktivieren. Das tut man in der Regel durch den Eintrag

```
<OPT-Name>_DO_DEBUG='yes'
```

in der Konfigurationsdatei.<sup>10</sup> Jetzt wird während der Ausführung am Bildschirm genau dargestellt, was passiert.

---

<sup>10</sup>Manchmal werden mehrere Start-Skripte verwendet, die dann auch verschiedene Namen für ihre Debug-Variablen haben. Hier hilft ein kurzer Blick in die Skripte.

**Weitere beim Debuggen hilfreiche Variablen**

**DEBUG\_ENABLE\_CORE** Diese Variable gestattet das Erzeugen von „Core-Dumps“ (Speicherausgüssen). Stürzt ein Programm aufgrund eines Fehlers ab, wird ein Abbild des aktuellen Zustandes im Dateisystem abgelegt, der hinterher zur Analyse des Problems verwendet werden kann. Die Core-Dumps werden unter `/var/log/dumps/` abgelegt.

**DEBUG\_IP** Wird diese Variable gesetzt, werden alle Aufrufe des Programms `ip` protokolliert.

**DEBUG\_IPUP** Wird diese Variable auf „yes“ gesetzt, werden während der Ausführung der `ip-up/ip-down`-Skripte die ausgeführten Anweisungen mitgezeichnet und im System-Protokoll gespeichert.

**LOG\_BOOT\_SEQ** Wird diese Variable auf „yes“ gesetzt, protokolliert der `bootlogd` während des Bootens alle auf der Konsole getätigten Ausgaben in der Datei `/var/tmp/boot.log`. Diese Variable hat standardmäßig den Wert „yes“.

**DEBUG\_KEEP\_BOOTLOGD** Normalerweise wird der `bootlogd` am Ende des Bootvorganges beendet. Ein Setzen dieser Variable unterbindet das und erlaubt ein Protokollieren der Konsolenausgaben über den Bootvorgang hinaus.

**DEBUG\_MDEV** Ein Setzen dieser Variable generiert ein Protokoll des `mdev`-Daemons, der für das Anlegen der Geräte-Dateien unter `/dev` zuständig ist.

**1.4.5 Hinweise**

- Es ist *immer* besser, geschweifte Klammern „`{...}`“ an Stelle von runden Klammern „`(...)`“ zu benutzen. Allerdings muss dabei darauf geachtet werden, dass nach der öffnenden Klammer ein Leerzeichen oder eine neue Zeile vor dem nächsten Befehl kommt und vor der schließenden Klammer ein Semikolon oder auch eine neue Zeile kommt. Beispielsweise ist

```
{ echo "cpu"; echo "quit"; } | ...
```

gleichbedeutend mit:

```
{
    echo "cpu"
    echo "quit"
} | ...
```

- Ein Skript kann mit „`exit`“ vorzeitig beendet werden. Dies ist aber bei den Start-Skripten (`opt/etc/boot.d/...`, `opt/etc/rc.d/...`), den Stopp-Skripten (`opt/etc/rc0.d/...`) und den `ip-up/ip-down`-Skripten (`opt/etc/ppp/...`) geradezu tödlich, da auch nachfolgende Skripte nicht mehr ausgeführt werden. Im Zweifelsfall immer weglassen.
- KISS – Keep it small and simple. Du willst Perl als Skript-Sprache verwenden? Dir reichen die Skripting-Möglichkeiten von `fl4l` nicht? Überdenke deine Einstellung! Ist dein `OPT` wirklich nötig? `fl4l` ist immer noch „nur“ ein Router, ein Router sollte eigentlich keine Serverdienste anbieten.

- Die Fehlermeldung „: not found“ heißt meistens, dass das Skript noch im DOS-Format vorliegt. Weitere Fehlerquelle: Das Skript ist nicht ausführbar. In beiden Fällen sollte die `opt/<PACKAGE>.txt`-Datei daraufhin geprüft werden, ob sie die korrekten Optionen (in Bezug auf „mode“, „gid“, „uid“ und Flags) enthält. Wenn das Skript erst beim Booten erzeugt wird, sollte ein `„chmod +x <Skriptname>“` ausgeführt werden.
- Für temporäre Dateien sollte der Pfad `/tmp` genutzt werden. Es ist aber unbedingt darauf zu achten, dass hier nur wenig Platz ist, weil dies in der RootFS-RAM-Disk liegt! Wenn mehr Platz benötigt wird, muss man sich eine eigene RAM-Disk erstellen und mounten. Details dazu verrät der Abschnitt „RAM-Disks“ in dieser Dokumentation.
- Damit temporäre Dateien eindeutige Namen erhalten, sollte man grundsätzlich die aktuelle Prozess-ID, die in der Shell-Variablen `„$“` gespeichert ist, an den Dateinamen anhängen. `/tmp/<OPT-Name>.$$` stellt somit einen guten Dateinamen dar, `/tmp/<OPT-Name>` eher weniger, wobei `<OPT-Name>` natürlich nicht so stehen bleiben soll, sondern geeignet ersetzt werden muss.

## 1.5 Arbeit mit dem Paketfilter

### 1.5.1 Hinzufügen von eigenen Ketten und Regeln

Zur Manipulation des Paketfilters steht eine Reihe von Routinen zur Verfügung, mit deren Hilfe man Ketten (engl. „Chains“) und Regeln hinzufügen und wieder löschen kann. Eine Kette ist eine benannte und geordnete Liste von Regeln. Es gibt einen Satz vordefinierter Ketten (`PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, `POSTROUTING`); mit Hilfe dieser Funktionen können weitere Ketten nach Bedarf erstellt werden.

**add\_chain/add\_nat\_chain <chain>:** Fügt eine Kette zur „filter“- oder „nat“-Tabelle hinzu.

**flush\_chain/flush\_nat\_chain <chain>:** Entfernt alle Regeln aus einer Kette der „filter“- oder „nat“-Tabelle.

**del\_chain/del\_nat\_chain <chain>:** Entfernt eine Kette aus der „filter“- oder „nat“-Tabelle. Ketten müssen leer sein, bevor sie gelöscht werden können, und es darf auch keine Referenz mehr auf sie geben. Eine solche Referenz ist z. B. eine JUMP-Aktion, deren Ziel eben diese Kette ist.

**add\_rule/ins\_rule/del\_rule:** Fügt Regeln am Ende einer Kette (`add_rule`) bzw. an beliebiger Stelle in einer Kette (`ins_rule`) ein bzw. löscht Regeln aus einer Kette (`del_rule`). Ein Aufruf sieht wie folgt aus:

```
add_rule <table> <chain> <rule> <comment>
ins_rule <table> <chain> <rule> <position> <comment>
del_rule <table> <chain> <rule> <comment>
```

wobei die Parameter folgende Bedeutung haben:

**table** Die Tabelle, in der sich die Kette befindet

**chain** Die Kette, in welche die Regel eingefügt werden soll

**rule** Die Regel, die eingefügt werden soll; das Format entspricht dem in der Konfigurationsdatei verwendeten

**position** Die Position, an der die Regel eingefügt werden soll (nur bei `ins_rule`)

**comment** Ein Kommentar, der zusammen mit der Regel angezeigt werden soll, wenn sich jemand den Paketfilter ansieht.

### 1.5.2 Einordnen in bestehende Regeln

fi4l konfiguriert den Paketfilter mit einem gewissen Standardregelsatz. Will man eigene Regeln einfügen, wird man diese in der Regel nach dem Standardregelsatz einfügen wollen. Ebenfalls wird man wissen wollen, was denn die vom Nutzer gewünschte Aktion beim Verwerfen eines Paketes ist. Diese Informationen bekommt man für die FORWARD- und INPUT-Ketten durch Aufruf zweier Funktionen, `get_defaults` und `get_count`. Nach Aufruf von

```
get_defaults <chain>
```

erhält man die folgenden Ergebnisse:

**drop:** Diese Variable enthält die Kette, in die verzweigt wird, wenn ein Paket verworfen wird.

**reject:** Diese Variable enthält die Kette, in die verzweigt wird, wenn ein Paket abgelehnt wird.

Nach Aufruf von

```
get_count <chain>
```

erhält man in der Variable `res` die Anzahl der Regeln in der Kette `<chain>`. Diese Position ist insofern wichtig, als man *nicht* einfach `add_rule` verwenden kann, um eine Regel am Ende der vordefinierten „filter“-Ketten INPUT, FORWARD und OUTPUT einzufügen. Dies liegt daran, dass diese Ketten mit einer Standardregel abgeschlossen werden, welche alle verbliebenen Pakete behandelt, je nach Belegung der `PF_<Kette>_POLICY`-Variablen. Ein Einfügen *hinter* dieser letzten Regel hat also keine Auswirkungen. Die Funktion `get_count` erlaubt es nun hingegen, die Stelle direkt *vor* dieser letzten Regel zu ermitteln und die Position dann an die `ins_rule`-Funktion im Parameter `<position>` zu übergeben, um die Regel wie gewünscht am Ende der jeweiligen Kette, aber vor der letzten Auffang-Regel einzubauen.

Ein Beispiel aus dem Skript `opt/etc/rc.d/rc390.dns_dhcp` des Pakets „dns\_dhcp“ soll dies verdeutlichen:

```
case $OPT_DHCPRELAY in
    yes)
        begin_script DHCRELAY "starting dhcprelay ..."

        idx=1
        interfaces=""
        while [ $idx -le $DHCPRELAY_IF_N ]
        do
            eval iface='$DHCPRELAY_IF_'$idx

            get_count INPUT
            ins_rule filter INPUT "prot:udp  if:$iface:any 68 67 ACCEPT" \
```

```

        $res "dhcrelay access"

        interfaces=$interfaces' -i '$iface
        idx=`expr $idx + 1`
    done
    dhcrelay $interfaces $DHCPRELAY_SERVER

    end_script
;;
esac

```

Hier sieht man inmitten der Schleife einen Aufruf von `get_count`, gefolgt von einem Aufruf der `ins_rule`-Funktion, der unter anderem die `res`-Variable als `position`-Parameter übergeben wird.

### 1.5.3 Erweiterung der Paketfilter-Tests

fi4l verwendet in den Paketfilterregeln die Syntax `match:params`, um zusätzliche Bedingungen an die Pakete zu stellen (siehe `mac:`, `limit:`, `length:`, `prot:`, ...). Will man zusätzliche Tests hinzufügen, wird das folgendermaßen gemacht:

1. Festlegen eines passenden Namens. Dieser Name muss mit einem Kleinbuchstaben im Bereich a-z beginnen und ansonsten aus beliebigen Buchstaben und Ziffern bestehen.

**Wenn der Paketfilter-Test in IPv6-Regeln verwendet werden soll, dann muss darauf geachtet werden, dass der Name keine gültige IPv6-Adresskomponente ist!**

2. Anlegen einer Datei `opt/etc/rc.d/fwrules-<name>.ext`. In dieser Datei steht in etwa Folgendes:

```

# IPv4 extension is available
foo_p=yes

# the actual IPv4 extension, adding matches to match_opt
do_foo()
{
    param=$1
    get_negation $param
    match_opt="$match_opt -m foo $neg_opt --fooval $param"
}

# IPv6 extension is available
foo6_p=yes

# the actual IPv6 extension, adding matches to match_opt
do6_foo()
{
    param=$1
    get_negation6 $param
    match_opt="$match_opt -m foo $neg_opt --fooval $param"
}

```

Der Paketfilter-Test muss nicht zwingend sowohl für IPv4 als auch für IPv6 implementiert sein (obwohl dies zu bevorzugen ist, falls er für beide Layer-3-Protokolle sinnvoll ist).

### 3. Testen der Erweiterung:

```
$ cd opt/etc/rc.d
$ sh test-rules.sh 'foo:bar ACCEPT'
add_rule filter FORWARD 'foo:bar ACCEPT'
iptables -t filter -A FORWARD -m foo --fooval bar -s 0.0.0.0/0 \
    -d 0.0.0.0/0 -m comment --comment foo:bar ACCEPT -j ACCEPT
```

### 4. Aufnahme der Erweiterung und aller noch benötigten Dateien (iptables-Komponenten) ins Archiv über einen der bekannten Mechanismen.

### 5. Zulassen der Erweiterung in der Konfiguration durch Erweitern von FW\_GENERIC\_MATCH und/oder FW\_GENERIC\_MATCH6 in einer exp-Datei, z. B. wie folgt:

```
+FW_GENERIC_MATCH(OPT_FOO) = 'foo:bar' : ''
+FW_GENERIC_MATCH6(OPT_FOO) = 'foo:bar' : ''
```

## 1.6 CGI-Erstellung für das *httpd*-Paket

### 1.6.1 Allgemeines zum Webserver

Der Webserver, der bei fli4l verwendet wird, ist der `mini_httpd` von ACME Labs. Die Quellen können unter [http://www.acme.com/software/mini\\_httpd/](http://www.acme.com/software/mini_httpd/) heruntergeladen werden. Allerdings wurden für fli4l ein paar Änderungen vorgenommen. Die Anpassungen befinden sich im *src*-Paket im Verzeichnis `src/fbr/buildroot/package/mini_httpd`.

### 1.6.2 Skriptnamen

Der Skriptname sollte möglichst vielsagend sein, damit er von anderen Skripten leichter zu unterscheiden ist und es keine Namensüberschneidungen bei verschiedenen OPTs gibt.

Damit die Skripte ausführbar gemacht werden und DOS-Zeilenumbrüche in Unix-Zeilenumbrüche umgewandelt werden, muss in der `opt/<PAKET>.txt` ein entsprechender Eintrag gemacht werden, siehe Tabelle 1.2 (Seite 10).

### 1.6.3 Menü-Einträge

Um einen Eintrag im Menü vorzunehmen, muss eine Eintragung in der Datei `/etc/httpd/menu` vorgenommen werden. Dieser Mechanismus erlaubt es OPTs, auch im laufendem Betrieb Änderungen am Menü vorzunehmen. Dies sollte nur mit dem Skript `httpd-menu.sh` gemacht werden, da dieses darauf achtet, dass das Dateiformat dieser Datei immer konsistent ist. Um neue Menüpunkte einzufügen, wird es folgendermaßen aufgerufen:

```
httpd-menu.sh add [-p <priority>] <link> <name> [section] [realm]
```

So wird ein Eintrag mit dem Namen `<name>` in den Abschnitt `[section]` eingetragen. Wenn `[section]` weggelassen wird, wird es standardmäßig in den Abschnitt „OPT-Pakete“ eingetragen. `<link>` gibt das Ziel des neuen Links an. `<priority>` spezifiziert die Priorität eines Menüeintrags in seinem Abschnitt. Wird sie nicht angegeben, wird die Standardpriorität 500 benutzt. Die Priorität sollte eine dreistellige Nummer sein. Je kleiner die Priorität, desto weiter oben steht der Link in dem Abschnitt. Soll ein Eintrag möglichst weit nach unten, so ist z.B. die Priorität 900 zu wählen. Bei gleicher Priorität werden die Einträge nach dem Ziel des Links sortiert. Bei `[realm]` wird der Bereich angegeben, für den ein angemeldeter Benutzer mindestens die Berechtigung *view* haben muss, damit der Menüpunkt angezeigt wird. Wird `[realm]` nicht angegeben, wird der Menüpunkt immer angezeigt. Siehe hierzu auch den Abschnitt „Benutzerrechte“ (Seite 51).

Beispiel:

```
httpd-menu.sh add "neuedatei.cgi" "Hier klicken" "Tools" "tools"
```

Dieses Beispiel erzeugt im Abschnitt „Tools“ einen Link mit dem Titel „Hier klicken“ und dem Link-Ziel „neuedatei.cgi“ und legt den Abschnitt, falls dieser nicht vorhanden ist, an.

Das Skript kann auch Einträge aus dem Menü wieder entfernen:

```
httpd-menu.sh rem <link>
```

Mit diesem Aufruf wird der Eintrag mit dem Link `<link>` wieder entfernt.

**Wichtig:** Wenn mehrere Menüeinträge auf die gleiche Datei verweisen, werden alle diese Einträge aus dem Menü entfernt.

Da Abschnitte auch Prioritäten haben können, können diese auch manuell angelegt werden. Wird ein Abschnitt automatisch beim Hinzufügen eines Eintrages angelegt, erhält er automatisch die Priorität 500. Der Syntax zum Anlegen von Abschnitten lautet:

```
httpd-menu.sh addsec <priority> <name>
```

Auch hier sollte `<priority>` nur dreistellige numerische Werte annehmen.

Um sinnvolle Prioritäten in Erfahrung zu bringen, lohnt es sich, auf einem laufenden fl4l in die Datei `/etc/httpd/menu` zu schauen, die Prioritäten stehen in der zweiten Spalte.

Der Vollständigkeit halber wird hier kurz auf das Dateiformat der Menüdatei eingegangen. Wem die Funktion von `httpd-menu.sh` reicht, der kann diesen Abschnitt überspringen. Die Datei `/etc/httpd/menu` hat den folgenden Aufbau: Sie ist in vier Spalten eingeteilt. In der ersten Spalte steht ein Kennbuchstabe, der Überschriften und Einträge unterscheidet. In der zweiten Spalte steht die Sortierungspriorität. Die dritte Spalte enthält bei Einträgen das Ziel des Links und bei Überschriften einen Bindestrich, da dieses Feld bei Überschriften keine Bedeutung hat. Im Rest der Zeile steht der Text, der nachher im Menü erscheint.

Überschriften benutzen den Kennbuchstaben „t“, dann wird ein neuer Menüabschnitt begonnen. Normale Menüeinträge benutzen den Kennbuchstaben „e“. Ein Beispiel:

```
t 300 - Mein tolles OPT
e 200 meinopt.cgi Mach etwas Tolles
e 500 meinopt.cgi?mehr=ja Mach mehr Tolles
```

Beim Bearbeiten dieser Datei muss man darauf achten, dass das `httpd-menu.sh`-Skript die Datei immer sortiert abspeichert. Die einzelnen Abschnitte sind sortiert und die Einträge pro Abschnitt sind in diesem Abschnitt sortiert. Der genaue Sortieralgorithmus kann aus `httpd-menu.sh` übernommen werden – besser wäre allerdings, dieses Skript um mögliche neue Funktionen zu erweitern, damit alle Menü-Bearbeitungen an zentraler Stelle passieren.

### 1.6.4 Aufbau eines CGI-Skriptes

#### Die Kopfzeilen

Alle Skripte des Webservers sind einfache Shell-Skripte (Interpreter wie z.B. Perl, PHP, etc. sind viel zu groß für fli4l). Sie sollten mit dem obligatorischen Skript-Header anfangen (Verweis auf den Interpreter, Name, Sinn des Skriptes, Autor, Lizenz).

#### Das Hilfsskript “cgi-helper“

Gleich nach den Kopfzeilen sollte dann schon das Hilfsskript `cgi-helper` mit folgendem Aufruf eingebunden werden:

```
. /srv/www/include/cgi-helper
```

Wichtig ist ein Leerzeichen zwischen Punkt und Schrägstrich!

Dieses Skript stellt diverse Hilfsfunktionen bereit, die das Erstellen von CGIs für fli4l wesentlich vereinfachen sollen. Außerdem werden mit dem Einbinden des `cgi-helper` auch noch Standardaufgaben ausgeführt, wie beispielsweise das Parsen von Variablen, die mit Formularen oder über die URL übergebenen wurden, oder das Laden von Sprach- und CSS-Dateien.

Tabelle 1.4 gibt einen Überblick über die Funktionen des `cgi-helper`-Skriptes.

Tabelle 1.4: Funktionen des `cgi-helper`-Skriptes

Name	Funktion
<code>check_rights</code>	Überprüfung der Benutzerrechte
<code>http_header</code>	Ausgabe eines Standard-HTTP-Headers oder eines speziellen HTTP-Headers, beispielsweise zum Download von Dateien
<code>show_html_header</code>	Ausgabe des kompletten Seitenheaders (inkl. HTTP-Header, Kopfzeile und Menü)
<code>show_html_footer</code>	Ausgabe des Abschlusses der HTML-Seite
<code>show_tab_header</code>	Ausgabe eines Inhalt-Fensters mit Reitern
<code>show_tab_footer</code>	Ausgabe des Abschlusses des Inhaltsfensters
<code>show_error</code>	Ausgabe einer Box für Fehlermeldungen (Hintergrundfarbe: rot)
<code>show_warn</code>	Ausgabe einer Box für Warnmeldungen (Hintergrundfarbe: gelb)
<code>show_info</code>	Ausgabe einer Box für Informationen oder Erfolgsmeldungen (Hintergrundfarbe: grün)

#### Der Inhalt eines CGI-Skriptes

Um ein einheitliches Design und vor allem die Kompatibilität mit zukünftigen fli4l-Versionen zu gewährleisten, ist es sehr zu empfehlen, die Funktionen des Hilfsskriptes `cgi-helper` zu benutzen, auch wenn man in einem CGI theoretisch alle Ausgaben selbst generieren kann.

Eine einfaches CGI-Skript könnte wie folgt aussehen:



```
#!/bin/sh
# -----
# Header (c) Autor Datum
# -----
# get main helper functions
. /srv/www/include/cgi-helper

show_html_header "Mein erstes CGI"
echo '    <h2>Willkommen</h2>'
echo '    <h3>Dies ist ein Beispiel-CGI-Skript</h3>'
show_html_footer
```

### Die Funktion `show_html_header`

Die Funktion `show_html_header` erwartet eine Zeichenkette als Parameter. Diese Zeichenkette stellt den Titel der zu generierenden Seite dar. Die Funktion generiert automatisch das Menü und bindet ebenso automatisch zum Skript gehörende CSS- und Sprachdateien ein. Voraussetzung dafür ist, dass diese sich in den Verzeichnissen `/srv/www/css` bzw. `/srv/www/lang` befinden und denselben Namen (aber natürlich eine andere Endung) wie das Skript haben. Ein Beispiel:

```
/srv/www/admin/OpenVPN.cgi
/srv/www/css/OpenVPN.css
/srv/www/lang/OpenVPN.de
```

Sowohl das Benutzen von Sprachdateien als auch von CSS-Dateien ist optional. Immer eingebunden werden `css/main.css` und `lang/main.<lang>`, wobei `<lang>` der gewählten Sprache entspricht.

Der Funktion `show_html_header` können aber neben dem Titel noch weitere Parameter übergeben werden. Ein Aufruf mit allen möglichen Parametern könnte so aussehen:

```
show_html_header "Titel" "refresh=$time;url=$url;cssfile=$cssfile;showmenu=no"
```

Alle zusätzlichen Parameter müssen, wie im Beispiel gezeigt, mit Anführungszeichen umschlossen und durch ein Semikolon getrennt werden. Eine Überprüfung der Syntax erfolgt *nicht*! Es ist also notwendig, sehr genau auf die Parameterübergabe zu achten.

Hier eine kurze Übersicht über die Funktion der Parameter:

- `refresh=time`: Zeit in Sekunden in der die Seite vom Browser neu geladen werden soll
- `url=url`: die URL, die bei einem Refresh neu geladen wird
- `cssfile=cssfile`: Name einer CSS-Datei, wenn diese vom Namen des CGIs abweicht
- `showmenu=no`: unterdrückt die Anzeige des Menüs und des Headers

Sonstige Richtlinien zum Inhalt:

- Fasst euch kurz :-)
- Schreibt sauberes HTML (SelfHTML<sup>11</sup> ist da ein guter Ansatzpunkt).
- Verzichtet auf hochmodernen Schnick-Schnack (JavaScript ist i. O., wenn es nicht stört, sondern den Benutzer unterstützt, das Ganze muss auch ohne JavaScript funktionieren).

<sup>11</sup>siehe <http://de.selfhtml.org/>

**Die Funktion `show_html_footer`**

Die Funktion `show_html_footer` schließt den Block im CGI-Skript ab, der durch die Funktion `show_html_header` eingeleitet wurde.

**Die Funktion `show_tab_header`**

Damit der Inhalt eurer mit Hilfe des CGIs erzeugten Webseite auch hübsch geordnet aussieht, könnt ihr die `cgi-helper`-Funktion `show_tab_header` nutzen. Sie erzeugt dann anklickbare Reiter („Tabs“ genannt), so dass ihr eure Seite in mehrere logisch voneinander getrennte Bereiche unterteilen könnt.

Der `show_tab_header`-Funktion werden Parameter immer in Paaren übergeben. Der erste Wert entspricht dem Titel eines Reiters, der zweite dem Link. Wird als Link die Zeichenkette „no“ übergeben, wird lediglich der Titel ausgegeben und der Reiter ist nicht anklickbar (und blau).

Im folgenden Beispiel wird ein „Fenster“ mit dem Titel „Ein tolles Fenster“ erzeugt. Im Fenster steht „foo bar“:

```
show_tab_header "Ein tolles Fenster" "no"
echo "foo"
echo "bar"
show_tab_footer
```

Im nächsten Beispiel werden zwei anklickbare Reiter generiert, die dem Skript die Variable `action` mit verschiedenen Werten übergibt.

```
show_tab_header "1. Auswahltab" "$myname?action=machdies" \
                "2. Auswahltab" "$myname?action=machjenes"
echo "foo"
echo "bar"
show_tab_footer
```

Nun kann das Skript den Inhalt der Variablen `FORM_action` (siehe weiter unten zur Variablenauswertung) auswerten und je nachdem unterschiedliche Inhalte bereitstellen. Damit der angeklickte Reiter auch ausgewählt erscheint und nicht mehr angeklickt werden kann, müsste der Funktion statt dem Link wie schon erwähnt ein „no“ übergeben werden. Das geht aber auch einfacher, wenn man sich an die Konvention in folgendem Beispiel hält:

```
_opt_machdies="1. Auswahltab"
_opt_machjenes="2. Auswahltab"
show_tab_header "$_opt_machdies" "$myname?action=opt_machdies" \
                "$_opt_machjenes" "$myname?action=opt_machjenes"
case $FORM_action in
    opt_machdies) echo "foo" ;;
    opt_machjenes) echo "bar" ;;
esac
show_tab_footer
```

Wird also für den Titel eine Variable verwendet, deren Namen dem Inhalt der Variablen `action` mit führendem Unterstrich (`_`) entspricht, wird der entsprechende Reiter ausgewählt dargestellt.

### Die Funktion `show_tab_footer`

Die Funktion `show_tab_footer` schließt den Block im CGI-Skript ab, der durch die Funktion `show_tab_header` eingeleitet wurde.

### Mehrsprachfähigkeit

Das Hilfsskript `cgi-helper` enthält weiterhin Funktionen, um CGI-Skripte mehrsprachig zu machen. Dazu müssen „nur“ für alle Textausgaben Variablen mit führenden Unterstrichen (`_`) verwendet und diese Variablen in den entsprechenden Sprachdateien definiert werden.

Beispiel:

`lang/opt.de` enthalte:

```
_opt_machdies="Eine Ausgabe"
```

`lang/opt.en` enthalte:

```
_opt_machdies="An Output"
```

`admin/opt.cgi` enthalte:

```
...  
echo $_opt_machdies  
...
```

### Formular-Auswertung

Um Formulare zu verarbeiten, muss man noch einige Dinge wissen. Egal ob die Formular-Methode GET oder POST verwendet wird, die Parameter finden sich nach dem Einbinden des `cgi-helper`-Skripts (welches wiederum das Hilfsprogramm `proccgi` aufruft) in den Variablen `FORM_<Parameter>` wieder. Wenn das Formularfeld also den Namen „eingabe“ hatte, kann im CGI-Skript mit `$FORM_eingabe` darauf zugegriffen werden.

Weitere Informationen zum Programm `proccgi` finden sich unter <http://www.fpx.de/fp/Software/ProcCGI.html>.

### Benutzerrechte: Die Funktion `check_rights`

Um zu prüfen, ob der Benutzer ausreichende Rechte zur Nutzung eines CGI-Skripts besitzt, muss am Anfang des CGI-Skripts die Funktion `check_rights` wie folgt aufgerufen werden:

```
check_rights <Bereich> <Aktion>
```

Das CGI-Skript wird dann nur ausgeführt, wenn der aktuell angemeldete Benutzer

- alle Rechte hat (`HTTPD_RIGHTS_x='all'`), oder
- alle Rechte für den angegebenen Bereich hat (`HTTPD_RIGHTS_x='<Bereich>:all'`), oder
- das Recht hat, die angegebene Aktion im angegebenen Bereich auszuführen (`HTTPD_RIGHTS_x='<Bereich>:<Aktion>'`).

### Die Funktion `show_error`

Diese Funktion gibt eine Fehlermeldung in einer roten Box aus. Sie erwartet zwei Parameter: einen Titel sowie eine Meldung. Beispiel:

```
show_error "Error: No key" "No key was specified!"
```

### Die Funktion `show_warn`

Diese Funktion gibt eine Warnmeldung in einer gelben Box aus. Sie erwartet zwei Parameter: einen Titel sowie eine Meldung. Beispiel:

```
show_info "Warnung" "Derzeit besteht keine Verbindung!"
```

### Die Funktion `show_info`

Diese Funktion gibt eine Informations- oder Erfolgsmeldung in einer grünen Box aus. Sie erwartet zwei Parameter: einen Titel sowie eine Meldung. Beispiel:

```
show_info "Info" "Aktion wurde erfolgreich ausgeführt!"
```

### Das Hilfsskript “`cgi-helper-ip4`”

Gleich nach dem Hilfsskript “`cgi-helper`” kann dann das Hilfsskript `cgi-helper-ip4` mit folgendem Aufruf eingebunden werden:

```
. /srv/www/include/cgi-helper-ip4
```

Wichtig ist ein Leerzeichen zwischen Punkt und Schrägstrich!

Dieses Skript stellt Hilfsfunktionen bereit, um Prüfungen von IPv4-Adressen vornehmen zu können.

### Die Funktion `ip4_isvalidaddr`

Diese Funktion überprüft, ob eine gültige IPv4-Adresse übergeben wurde. Beispiel:

```
if ip4_isvalidaddr ${FORM_inputip}
then
    ...
fi
```

### Die Funktion `ipv4_normalize`

Diese Funktion entfernt aus der übergebenen IPv4-Adresse führende Nullen. Beispiel:

```
ip4_normalize ${FORM_inputip}
IP=$res
if [ -n "$IP" ]
then
    ...
fi
```

**Die Funktion `ipv4_isindhcprange`**

Diese Funktion prüft, ob die übergebene IPv4-Adresse sich im Bereich der übergebenen Start- und Zieladresse befindet. Beispiel:

```
if ipv4_isindhcprange $FORM_inputip $ip_start $ip_end
then
    ...
fi
```

**1.6.5 Sonstiges**

Dies und das (ja, das ist auch noch wichtig!):

- Der `mini_httpd` schützt Unterverzeichnisse nicht mit einem Passwort. Es muss für jedes Verzeichnis eine eigene `.htaccess`-Datei oder ein Link auf eine andere `.htaccess`-Datei angelegt werden.
- KISS - Keep it simple, stupid!
- Diese Angaben können sich jederzeit ohne Vorankündigung ändern!

**1.6.6 Fehlersuche**

Um die Fehlersuche innerhalb eines CGI-Skripts zu erleichtern, kann man vor der Einbindung des `cgi-helper`-Skripts den Debugging-Modus aktivieren. Dazu muss die Variable `set_debug` auf den Wert „yes“ gesetzt werden. Dies führt zur Erstellung der Datei `debug.log`, die über die URL `http://<fli4l-Host>/admin/debug.log` heruntergeladen werden kann. Diese enthält alle Aufrufe des CGIs. Die `set_debug`-Variable ist nicht global, muss also in jedem Problem-CGI erneut gesetzt werden. Beispiel:

```
set_debug="yes"
. /srv/www/include/cgi-helper
```

Alternativ kann auch die jeweilige CGI-URL um den Parameter `debug=yes` ergänzt werden, etwa `http://<fli4l-Host>/admin/meinopt.cgi?debug=yes`.

Des Weiteren eignet sich `cURL`<sup>12</sup> hervorragend zur Fehlersuche, insbesondere wenn die HTTP-Kopfzeilen nicht korrekt zusammengesetzt werden oder der Browser nur weiße Seiten anzeigt. Auch ist das Cachingverhalten moderner Webbrowser bei der Fehlersuche hinderlich.

Beispiel: Mit dem folgenden Aufruf wird der HTTP-Header („dump“, `-D`) sowie die normale Ausgabe des CGIs `admin/mein.cgi` ausgegeben. Es soll der Benutzername („user“, `-u`) „admin“ verwendet werden.

```
curl -D - http://fli4l/admin/mein.cgi -u admin
```

<sup>12</sup>siehe <http://de.wikipedia.org/wiki/CURL>

## 1.7 Hochfahren, Herunterfahren, Einwählen und Auflegen unter fli4l

### 1.7.1 Bootkonzept

fli4l 2.0 sollte eine saubere Installation auf eine Festplatte oder ein CompactFlash(TM)-Medium bieten, aber auch eine Installation auf ein Zip-Medium oder die Erstellung einer bootfähigen CD-ROM sollte möglich sein. Zusätzlich sollte die Festplattenversion sich nicht grundlegend von einer Installation auf Diskette<sup>13</sup> unterscheiden.

Diese Anforderungen wurden realisiert, indem die Dateien des `opt.img`-Archivs aus der bisherigen RAM-Disk auf ein anderes Medium verlagert werden können. Dies kann eine Partition auf einer Festplatte bzw. einem CF-Medium sein. Dieses zweite Volume wird unter `/opt` eingehängt, und dort liegende Programme werden nur noch über symbolische Links in das RootFS integriert. Das entstehende Layout im RootFS-Dateisystem entspricht dann dem im `opt`-Verzeichnis der ausgepackten fli4l-Distribution mit einer Ausnahme – das `files`-Präfix entfällt. Die Datei `opt/etc/rc` ist dann also direkt unter `/etc/rc` zu finden, `opt/bin/busybox` unter `/bin/busybox`. Dass diese Dateien unter Umständen nur symbolische Verknüpfungen auf ein im Nur-Lese-Modus eingehängtes Volume sind, kann man ignorieren, solange man die Dateien nicht modifizieren möchte. Will man dies tun, muss man die Dateien vorher mit `mk_writable` (s. u.) schreibbar machen.

### 1.7.2 Start- und Stopp-Skripte

Skripte, die beim Hochfahren des Systems ausgeführt werden sollen, befinden sich in den Verzeichnissen `opt/etc/boot.d/` und `opt/etc/rc.d/` und werden auch in dieser Reihenfolge ausgeführt. Des Weiteren befinden sich in `opt/etc/rc0.d/` Skripte, die beim Herunterfahren des Systems ausgeführt werden.

**Wichtig:** *Da zum Ausführen dieser Skripte kein separater Prozess erzeugt wird, dürfen sie nicht mit „exit“ abgeschlossen werden. Ein solcher Befehl führt zum vorzeitigen Abbruch des Bootvorgangs!*

#### Start-Skripte in `opt/etc/boot.d/`

Skripte, die in diesem Verzeichnis liegen, werden als erstes ausgeführt. Ihre Aufgabe ist es, das Boot-Volume einzuhängen, die auf dem Boot-Medium liegende Konfigurationsdatei `rc.cfg` einzulesen und das Opt-Archiv zu entpacken. Je nach Boot-Typ (Seite ??) sind diese Skripte mehr oder weniger kompliziert und tun die folgenden Dinge:

- Laden von Hardware-Treibern (optional)
- Boot-Volume einhängen (optional)
- Konfigurationsdatei `rc.cfg` vom Boot-Volume einlesen und in die Datei `/etc/rc.cfg` schreiben
- Einhängen des Opt-Volumes (optional)
- Extrahieren des Opt-Archivs (optional)

<sup>13</sup>Ursprünglich konnte fli4l auch von einer Diskette betrieben werden. Da fli4l inzwischen dafür zu groß geworden ist, wird dies nicht mehr unterstützt.

Damit die Skripte überhaupt eine Chance haben, etwas über die fli4l-Konfiguration zu erfahren, wird die Konfigurationsdatei auch ins RootFS-Archiv unter `/etc/rc.cfg` eingebunden; die Konfigurationsvariablen in dieser Datei stehen den Start-Skripten in `opt/etc/boot.d/` direkt zur Verfügung. Nach dem Einhängen des Boot-Volumes wird die `/etc/rc.cfg` durch die Konfigurationsdatei vom Boot-Volume ersetzt, so dass den Start-Skripten in `opt/etc/rc.d/` (s. u.) die aktuelle Konfiguration vom Boot-Volume zur Verfügung steht. <sup>14</sup>

### Start-Skripte in `opt/etc/rc.d/`

Befehle, die immer beim Starten des Routers ausgeführt werden müssen, können in Skripten im Verzeichnis `opt/etc/rc.d/` abgelegt werden. Hierbei gelten folgende Konventionen:

1. Es gilt folgende Namenskonvention:

```
rc<dreistellige Zahl>.<Name des OPTs>
```

Die Skripte werden in aufsteigender Reihenfolge der Zahlen gestartet. Ist mehreren Skripten dieselbe Zahl zugeordnet, entscheidet die alphabetische Sortierung nach dem Punkt. Falls der Start eines Pakets erst nach einem anderen erfolgen darf, wird das durch die Zahl festgelegt.

Hier eine ungefähre Richtlinie, welche Nummern für welche Aufgaben verwendet werden sollten:

Nummer	Aufgabe
000-099	Grundsystem (Hardware, Zeitzone, Dateisystem)
100-199	Kernel-Module (Treiber)
200-299	externe Verbindungen (PPPoE, ISDN4Linux, PPTP)
300-399	Netzwerk (Routing, Interfaces, Paketfilter)
400-497	Server (DHCP, HTTPD, Proxy, etc.)
498-499	reserviert (Aktivierung des Circuit-Systems)
500-997	nach Belieben (ab hier können Skripte von Circuits kontrollierte Verbindungen nutzen und laufen möglicherweise parallel zu einer Einwahl)
998-999	reserviert (bitte nicht benutzen!)

2. In diesen Skripten *müssen* alle Funktionen, die das RootFS verändern, hinterlegt werden, etwa das Anlegen eines Verzeichnisses `/var/log/lpd`.
3. In diesen Skripten dürfen *keine* Schreibzugriffe auf Dateien erfolgen, die Teil des Opt-Archivs sein können, da diese Dateien auf einem im Nur-Lese-Modus eingehängten Volume liegen können. Muss man eine solche Datei modifizieren, muss man sie vorher mit Hilfe der Funktion `mk_writable` (s. u.) beschreibbar machen. Durch den Aufruf der Funktion wird die Datei (wenn nötig) als beschreibbare Kopie im RootFS abgelegt. Ist die Datei bereits beschreibbar, bewirkt der `mk_writable`-Aufruf nichts.

<sup>14</sup>Normalerweise sind diese beiden Dateien identisch. Eine Abweichung entsteht nur, wenn die Konfigurationsdatei auf dem Boot-Volume händisch editiert wird, etwa um die Konfiguration nachträglich abzuändern, ohne die fli4l-Archive neu zu bauen.

**Wichtig:** *mk\_writable* muss direkt auf Dateien im RootFS angewandt werden, nicht über den Umweg des *opt*-Verzeichnisses. Will man also */usr/local/bin/foo* modifizieren, ruft man *mk\_writable* mit dem Argument */usr/local/bin/foo* auf.

4. Diese Skripte müssen vor der Ausführung der eigentlichen Befehle prüfen, ob das dazugehörige OPT auch aktiv ist. Das ist normalerweise durch eine einfache Fallunterscheidung erledigt:

```
if [ "$OPT_<OPT-Name>" = "yes" ]
then
    ...
    # Hier OPT starten!
    ...
fi
```

5. Um die Fehlersuche zu erleichtern, sollten die Skripte mit *begin\_script* und *end\_script* geklammert werden:

```
if [ "$OPT_<OPT-Name>" = "yes" ]
then
    begin_script F00 "configuring foo ..."
    ...
    end_script
fi
```

Die Fehlersuche einzelner Start-Skripte kann man dann einfach via *F00\_D0\_DEBUG='yes'* aktivieren.

6. Den Skripten stehen alle Konfigurationsvariablen direkt zur Verfügung. Im Abschnitt „[Umgang mit Konfigurationsvariablen](#)“ (Seite 40) wird erklärt, wie man von anderen Skripten aus auf die Konfigurationsvariablen zugreifen kann.
7. Der Pfad */opt* darf auch nicht als Speicherplatz für Datenbestände der OPTs benutzt werden. Falls zusätzlicher Speicherplatz benötigt wird, sollte dem Benutzer über eine Konfigurationsvariable die Möglichkeit gegeben werden, einen geeigneten Pfad auszuwählen. Je nach Art der zu speichernden Daten (persistente oder transiente Daten) sind verschiedene Standard-Belegungen sinnvoll. So bieten sich für transiente Daten etwa Pfade unterhalb von */var/run/* an; für persistente Daten sollte die Funktion [map2persistent](#) (Seite 40) in Kombination mit einer geeigneten Konfigurationsvariable verwendet werden.

### Stopp-Skripte in *opt/etc/rc0.d/*

Jeder Rechner muss mal heruntergefahren oder neu gestartet werden. Nun kann es vorkommen, dass man Vorgänge ausführen muss, bevor der Rechner heruntergefahren oder neu gestartet wird. Zum Herunterfahren und Neustarten sind die Befehle „halt“ bzw. „reboot“ zuständig. Diese Befehle werden auch aufgerufen, wenn man im IMONC oder in der Web-GUI auf die entsprechenden Schaltflächen klickt.

Alle Stopp-Skripte liegen im Verzeichnis *opt/etc/rc0.d/*. Ihre Dateinamen werden analog zu den Start-Skripten gebildet. Sie werden ebenfalls in *aufsteigender* Reihenfolge der Zahlen ausgeführt.



### 1.7.3 Hilfsfunktionen

In `/etc/boot.d/base-helper` werden verschiedene Funktionen bereitgestellt, die von den Start- und anderen Skripten verwendet werden können. Das betrifft Dinge wie Unterstützung zur Fehlersuche, Laden von Kernel-Modulen oder Ausgabe von Meldungen. Die einzelnen Funktionen werden im Folgenden aufgelistet und kurz beschrieben.

#### Skript-Steuerung

**begin\_script <Symbol> <Meldung>:** Gibt eine Meldung aus und aktiviert die Fehlersuche im Skript mittels `set -x`, falls `<Symbol>_DO_DEBUG` auf „yes“ steht.

**end\_script:** Gibt eine Abschluss-Meldung aus und deaktiviert die Fehlersuche, falls sie mit `begin_script` aktiviert wurde. Für jeden `begin_script`-Aufruf muss es einen zugehörigen `end_script`-Aufruf geben (und umgekehrt).

#### Laden von Kernel-Modulen

**do\_modprobe [-q] <Modul> <Parameter>\***: Lädt ein Kernel-Modul inkl. eventueller Parameter bei gleichzeitiger Auflösung der Modulabhängigkeiten. Der Parameter „-q“ verhindert, dass im Fehlerfall eine Meldung auf der Konsole ausgegeben und ins Boot-Protokoll geschrieben wird. Die Funktion liefert im Erfolgsfall den Rückgabewert null zurück und im Fehlerfall einen Wert ungleich null. Damit lässt sich Code schreiben, der ein Fehlschlagen des Ladens eines Kernel-Moduls geeignet behandelt:

```
if do_modprobe -q acpi-cpufreq
then
    # kein CPU-Frequenzsteuerung via ACPI
    log_error "CPU-Frequenzsteuerung via ACPI nicht verfügbar!"
    # [...]
else
    log_info "CPU-Frequenzsteuerung via ACPI aktiviert."
    # [...]
fi
```

**do\_modprobe\_if\_exists [-q] <Modulpfad> <Modul> <Parameter>\***: Prüft, ob das Modul `/lib/modules/<Kernel-Version>/<Modulpfad>/<Modul>` existiert und ruft bei Vorhandensein die Funktion `do_modprobe` auf.

**Wichtig:** *Das Modul muss tatsächlich unter dem angegebenen Modulnamen existieren, der Modulname darf kein Alias sein. Bei einem Alias wird direkt `do_modprobe` aufgerufen.*

#### Meldungen und Fehlerbehandlung

**log\_info <Meldung>:** Schreibt eine Meldung auf die Konsole und nach `/bootmsg.txt`. Wird keine Meldung als Parameter übergeben, liest `log_info` von der Standard-Eingabe. Die Funktion liefert als Rückgabewert immer null zurück.

**log\_warn** <Meldung>: Schreibt eine Warnmeldung auf die Konsole und nach `/bootmsg.txt`, wobei vor die Meldung die Zeichenkette **WARN:** gesetzt wird. Wird keine Meldung als Parameter übergeben, liest **log\_warn** von der Standard-Eingabe. Die Funktion liefert als Rückgabewert immer null zurück.

**log\_error** <Meldung>: Schreibt eine Fehlermeldung auf die Konsole und nach `/bootmsg.txt`, wobei vor die Meldung die Zeichenkette **ERR:** gesetzt wird. Wird keine Meldung als Parameter übergeben, liest **log\_error** von der Standard-Eingabe. Die Funktion liefert als Rückgabewert immer einen Wert ungleich null zurück.

**set\_error** <Meldung>: Gibt die Fehlermeldung aus und setzt eine interne Fehlervariable, das später via **is\_error** geprüft werden kann.

**is\_error**: Setzt die interne Fehlervariable zurück und liefert wahr zurück, falls sie vorher durch **set\_error** gesetzt wurde.

## Netzwerk-Funktionen

**translate\_ip\_net** <Wert> <Variablenname> [<Ergebnisvariable>]: Ersetzt symbolische Referenzen in Parametern. Momentan finden folgende Übersetzungen statt:

**Host- oder Netzwerk-Adressen** werden nicht übersetzt

**any** wird durch `0.0.0.0/0` ersetzt

**dynamic** wird durch die dynamische IPv4-Adresse des Routers ersetzt; das ist die IPv4-Adresse, die der Router beim Einwählen über einen Circuit mit einer Default-Route zugewiesen bekommt

**IP\_NET\_x** wird durch das in der Konfiguration stehende Netzwerk ersetzt; referenziert die Variable einen Circuit, so wird das dem Circuit zugewiesene IPv4-Netz zurückgegeben

**IP\_NET\_x\_IPADDR** wird durch die in der Konfiguration stehende IP-Adresse ersetzt; referenziert die Variable einen Circuit, so wird die dem Circuit zugewiesene IPv4-Adresse zurückgegeben

**IP\_ROUTE\_x** wird durch das in der Konfiguration stehende geroutete Netzwerk ersetzt

**@<Hostname>** wird durch die in der Konfiguration für den angegebenen Host spezifizierte IP-Adresse ersetzt

**{<circuit>}** wird durch das dem Circuit zugewiesene IPv4-Netz ersetzt

Das Ergebnis der Übersetzung wird in der Variable gespeichert, deren Name im dritten Parameter übergeben wird; fehlt dieser Parameter, wird das Ergebnis in der Variable **res** gespeichert. Der Variablenname, der im zweiten Parameter übergeben wird, wird nur für Fehlermeldungen benutzt, falls die Übersetzung fehlschlägt; hier kann also vom Aufrufer die Quelle des zu übersetzenden Wertes angegeben werden. Im Fehlerfall wird dann eine Meldung wie

Unable to translate value '<Wert>' contained in <Variablenname>.

ausgegeben.

Der Rückgabewert ist null, falls die Übersetzung erfolgreich war, und ungleich null, falls ein Fehler aufgetreten ist.

## Diverses

**mk\_writable <Datei>:** Stellt sicher, dass die übergebene Datei beschreibbar ist. Befindet sich die Datei auf einem im Nur-Lese-Modus eingehängten Volume und ist lediglich über eine symbolische Verknüpfung ins Dateisystem eingebunden, wird eine lokale Kopie angelegt, die dann beschreibbar ist.

**list\_unique <Liste>:** Entfernt Duplikate aus der übergebenen Liste. Das Resultat wird in die Standard-Ausgabe geschrieben.

### 1.7.4 mdev-Regeln

Für OPTs ist es möglich, zusätzliche mdev-Regeln zu etablieren, die spezielle Aktionen beim Erscheinen oder Verschwinden bestimmter Geräte vornehmen. Das `OPT_AUTOMOUNT` im `hd`-Paket verwendet beispielsweise eine solche Regel, um auftauchende Speichermedien automatisch einzuhängen. Will man eine zusätzliche mdev-Regel integrieren, muss man ein Skript der Form

```
/etc/mdev.d/mdev<Nummer>.<Name>
```

ins RootFS einbauen, wobei die Nummer ähnlich den Start- und Stopp-Skripten aus drei Ziffern bestehen muss und der Name beliebig gewählt werden kann. Innerhalb dieses Skriptes werden sämtliche Ausgaben an die Standardausgabe in die resultierende `/etc/mdev.conf` integriert. Ein Beispiel aus dem oben erwähnten `OPT_AUTOMOUNT`:

```
#!/bin/sh
#-----
# /etc/mdev.d/mdev500.automount - mdev HD automounting rules      __FLI4LVER__
#
#
# Last Update:  $Id: dev_main_boot_dial.tex 51959 2018-03-11 22:18:24Z kristov $
#-----

cat <<"EOF"
#
# mdev500.automount
#

-SUBSYSTEM=block;DEVTYPE=partition;.+      0:0 660 */lib/mdev/automount

EOF
```

Zu der Syntax der Regeln sei auf den Dateikopf der `/etc/mdev.conf` sowie auf die mdev-Dokumentation unter <http://git.busybox.net/busybox/plain/docs/mdev.txt> verwiesen. Falls eine Regel ein Skript aufruft (wie `/lib/mdev/automount` im obigen Beispiel), dann hat es Zugriff auf alle Variablen des auslösenden Kernel-„uevent“s, insbesondere auf:

- ACTION (typischerweise `add` oder `remove`, seltener `change`)
- DEVPATH (sysfs-Pfad zu der betroffenen Komponente)
- SUBSYSTEM (das betroffene Kernel-Subsystem, siehe unten)

- DEVNAME (die betroffene Gerätedatei unter /dev; fehlt, wenn es nicht um zu erstellende oder löschende Geräte geht, sondern z.B. um Module)
- MDEV (wird von mdev auf den Namen der letztlich erzeugten Gerätedatei gesetzt)

Beispiele für Kernel-Subsysteme:

**block** Blockgeräte (Speichermedien) wie **sda** (erste Festplatte), **sr0** (erstes CD-Laufwerk) oder **ram1** (zweite RAM-Disk)

**input** Geräte für Eingaben von Tastatur, Maus etc. wie **input/event0**; welche Gerätedateien welchen Geräten zugeordnet sind, ist nicht festgelegt und muss im sysfs ermittelt werden

**mem** Geräte zum Zugriff auf den Speicher und Hardware-Ports wie **mem** und **ports**; hier fallen auch Pseudo-Geräte wie **zero** (liefert ununterbrochen das ASCII-Zeichen mit Wert null) und **null** (liefert nichts, verschluckt alles) darunter

**sound** diverse Geräte für die Tonausgabe, Benennung uneinheitlich

**tty** Geräte zum Zugriff auf physische und virtuelle Konsolen wie **tty1** (erste virtuelle Konsole) oder **ttyS0** (erste serielle Konsole)

Ein Beispiel für die ersten beiden seriellen Schnittstellen:

```
mdev[42]: 30.050644 add@/devices/pnp0/00:04/tty/ttyS0
mdev[42]: ACTION=add
mdev[42]: DEVPATH=/devices/pnp0/00:04/tty/ttyS0
mdev[42]: SUBSYSTEM=tty
mdev[42]: MAJOR=4
mdev[42]: MINOR=64
mdev[42]: DEVNAME=ttyS0
mdev[42]: SEQNUM=613

mdev[42]: 30.051477 add@/devices/platform/serial8250/tty/ttyS1
mdev[42]: ACTION=add
mdev[42]: DEVPATH=/devices/platform/serial8250/tty/ttyS1
mdev[42]: SUBSYSTEM=tty
mdev[42]: MAJOR=4
mdev[42]: MINOR=65
mdev[42]: DEVNAME=ttyS1
mdev[42]: SEQNUM=614
```

Ein Beispiel für eine angeschlossene MF II-Tastatur:

```
mdev[41]: 4.030653 add@/devices/platform/i8042/serio0/input/input0
mdev[41]: ACTION=add
mdev[41]: DEVPATH=/devices/platform/i8042/serio0/input/input0
mdev[41]: SUBSYSTEM=input
mdev[41]: PRODUCT=11/1/1/ab41
mdev[41]: NAME="AT Translated Set 2 keyboard"
mdev[41]: PHYS="isa0060/serio0/input0"
mdev[41]: PROP=0
mdev[41]: EV=120013
mdev[41]: KEY=4 2000000 3803078 f800d001 feffffdf ffefffff ffffffff fffffffe
mdev[41]: MSC=10
mdev[41]: LED=7
mdev[41]: MODALIAS=input:b0011v0001p0001eAB41-e0,1,4,11,14,k71,72,73,74,75,76,77,79,
7A,7B,7C,7D,7E,7F,80,8C,8E,8F,9B,9C,9D,9E,9F,A3,A4,A5,A6,AC,AD,B7,B8,B9,D9,E2,ram4,l0,
1,2,sfw
mdev[41]: SEQNUM=604
```

Ein Beispiel für ein geladenes USB-Kernelmodul (`uhci_hcd`):

```
mdev[41]: 6.537506 add@/module/uhci_hcd
mdev[41]: ACTION=add
mdev[41]: DEVPATH=/module/uhci_hcd
mdev[41]: SUBSYSTEM=module
mdev[41]: SEQNUM=633
```

Ein Beispiel für eine angeschlossene Festplatte:

```
mdev[41]: 7.267527 add@/devices/pci0000:00/0000:00:07.1/ata1/host0/target0:0:0/0:0:0/block/sda
mdev[41]: ACTION=add
mdev[41]: DEVPATH=/devices/pci0000:00/0000:00:07.1/ata1/host0/target0:0:0/0:0:0/block/sda
mdev[41]: SUBSYSTEM=block
mdev[41]: MAJOR=8
mdev[41]: MINOR=0
mdev[41]: DEVNAME=sda
mdev[41]: DEVTYPE=disk
mdev[41]: SEQNUM=688
```

Dies ist eine ATA/IDE-Festplatte (`ata1`), die über den Gerätenamen `sda` angesprochen werden sollte.

Ein Beispiel für ein entferntes Blockgerät (die Zuordnung einer Image-Datei zu einer `fl4l`-VM wurde via `virsh detach-device` gelöst):

```
mdev[42]: 52.600646 remove@/devices/pci0000:00/0000:00:0a.0/virtio5/block/vdb/vdb1
mdev[42]: ACTION=remove
mdev[42]: DEVPATH=/devices/pci0000:00/0000:00:0a.0/virtio5/block/vdb/vdb1
mdev[42]: SUBSYSTEM=block
mdev[42]: MAJOR=254
mdev[42]: MINOR=17
mdev[42]: DEVNAME=vdb1
mdev[42]: DEVTYPE=partition
mdev[42]: SEQNUM=776

mdev[42]: 52.644642 remove@/devices/virtual/bdi/254:16
mdev[42]: ACTION=remove
mdev[42]: DEVPATH=/devices/virtual/bdi/254:16
mdev[42]: SUBSYSTEM=bdi
mdev[42]: SEQNUM=777

mdev[42]: 52.644718 remove@/devices/pci0000:00/0000:00:0a.0/virtio5/block/vdb
mdev[42]: ACTION=remove
mdev[42]: DEVPATH=/devices/pci0000:00/0000:00:0a.0/virtio5/block/vdb
mdev[42]: SUBSYSTEM=block
mdev[42]: MAJOR=254
mdev[42]: MINOR=16
mdev[42]: DEVNAME=vdb
mdev[42]: DEVTYPE=disk
mdev[42]: SEQNUM=778

mdev[42]: 52.644777 remove@/devices/pci0000:00/0000:00:0a.0/virtio5
mdev[42]: ACTION=remove
mdev[42]: DEVPATH=/devices/pci0000:00/0000:00:0a.0/virtio5
mdev[42]: SUBSYSTEM=virtio
mdev[42]: MODALIAS=virtio:d00000002v00001AF4
mdev[42]: SEQNUM=779

mdev[42]: 52.644973 remove@/devices/pci0000:00/0000:00:0a.0
mdev[42]: ACTION=remove
mdev[42]: DEVPATH=/devices/pci0000:00/0000:00:0a.0
mdev[42]: SUBSYSTEM=pci
mdev[42]: PCI_CLASS=10000
```

```
mdev[42]: PCI_ID=1AF4:1001
mdev[42]: PCI_SUBSYS_ID=1AF4:0002
mdev[42]: PCI_SLOT_NAME=0000:00:0a.0
mdev[42]: MODALIAS=pci:v00001AF4d00001001sv00001AF4sd00000002bc01sc00i00
mdev[42]: SEQNUM=780
```

Wie man sehen kann, sind bei einer solchen Entfernung diverse Kernel-Subsysteme involviert (hier block, bdi, virtio und pci).

### 1.7.5 ttyI-Geräte

Für die ttyI-Geräte (/dev/ttyI0 ... /dev/ttyI15), über welche die „Modem-Emulation“ der ISDN-Karte genutzt werden kann, existiert ein Zähler, um Konflikte zwischen verschiedenen Paketen, die diese Geräte nutzen, zu vermeiden. Hierzu wird beim Start des Routers die Datei /var/run/next\_ttyI angelegt, die von den verschiedenen OPTs abgefragt und fortgezählt werden kann. Mit dem folgenden Beispielskript kann dieser Wert abgefragt, um eins erhöht und wieder für das nächste OPT exportiert werden.

```
ttydev_error=
ttydev=$(cat /var/run/next_ttyI)
if [ $ttydev -le 16 ]
then
    ttydev=$((ttydev + 1))          # ttyI device available? yes
    echo $ttydev >/var/run/next_ttyI # ttyI device + 1
    # save it
else
    # ttyI device available? no
    log_error "No ttyI device for <Name deines OPTs> available!"
    ttydev_error=true              # set error for later use
fi

if [ -z "$ttydev_error" ]
then
    # start OPT only if next tty device
    # was available to minimize error
    ...                            # messages and minimize the
    # risk of uncomplete boot
fi
```

### 1.7.6 Skripte beim Einwählen und Auflegen

#### Allgemeines

Nach dem Herstellen bzw. Trennen einer Circuit-Verbindung werden die Skripte in /etc/ppp/ abgearbeitet. Hier können OPTs Aktionen hinterlegen, die nach dem Herstellen bzw. Auflegen der Verbindung nötig sind. Benannt werden die Dateien wie folgt:

```
ip-up<dreistellige Zahl>.<OPT-Name>
ip-down<dreistellige Zahl>.<OPT-Name>
```

Dabei werden die ip-up-Skripte nach dem *Aufbau* und die ip-down-Skripte nach dem *Abbau* der Verbindung ausgeführt.

**Wichtig:** In den *ip-down*-Skripten dürfen keine Aktionen ausgeführt werden, die zu einer erneuten Einwahl führen, da dadurch nur ein Dauer-Online-Zustand erreicht wird, was für Nicht-Flatrate-Benutzer ein teures Unterfangen ist.

**Wichtig:** Da für die einzelnen Skripte kein eigener Prozess erzeugt wird, dürfen auch diese Skripte nicht mit „exit“ abgeschlossen werden!

## Variablen

Durch das spezielle Aufrufkonzept stehen die folgenden Variablen den `ip-up`- und `ip-down`-Skripten zur Verfügung:

<code>real_interface</code>	die aktuelle Schnittstelle, also z. B. <code>ppp0</code> , <code>ippp0</code> , ...
<code>interface</code>	das IMOND-Interface, also <code>pppoe</code> , <code>ippp0</code> , ...
<code>tty</code>	verbundenes Terminal, möglicherweise leer!
<code>speed</code>	die Verbindungsgeschwindigkeit, bei ISDN z. B. 64000
<code>local</code>	die eigene IP-Adresse
<code>remote</code>	die IP-Adresse des Point-To-Point-Partners
<code>is_default_route</code>	gibt an, ob das aktuelle <code>ip-up</code> / <code>ip-down</code> für die Schnittstelle durchgeführt wird, über welche die Default-Route geht (kann „yes“ oder „no“ sein)

## Default-Route

Seit Version 2.1.0 werden die `ip-up`/`ip-down`-Skripte nicht nur für die Schnittstelle ausgeführt, über welche die Default-Route geht, sondern für alle Verbindungen, welche die `ip-up`- und `ip-down`-Skripte aufrufen. Um das alte Verhalten zu simulieren, muss in `ip-up`- und `ip-down`-Skripten die folgende Abfrage eingefügt werden:

```
# is a default-route-interface going up?
if [ "$is_default_route" = "yes" ]
then
    # die eigentlichen Aktionen
fi
```

Natürlich darf das neue Verhalten auch für spezielle Aktionen ausgenutzt werden.

## 1.8 Paket „template“

Um einiges von dem hier Beschriebenen etwas zu veranschaulichen, liegt der `fli4l`-Distribution das Paket „template“ bei. Dieses erklärt an kleinen Beispielen, wie:

- eine Konfigurationsdatei auszusehen hat (`config/template.txt`)
- eine Check-Datei geschrieben wird (`check/template.txt`)
- die erweiterten Prüfmöglichkeiten verwendet werden (`check/template.ext`)
- Konfigurationsvariablen für spätere Verwendung abgelegt werden können (`opt/etc/rc.d/rc999.template`)
- abgelegte Konfigurationsvariablen wieder ausgelesen werden (`opt/usr/bin/template_show_config`)

## 1.9 Aufbau des Boot-Datenträgers

Seit Version 1.5 wird das Programm `syslinux` zum Booten verwendet. Dieses hat den Vorteil, dass ein DOS-kompatibles Dateisystem auf dem Datenträger zur Verfügung steht.

Der Boot-Datenträger enthält folgende Dateien:

<code>ldlinux.sys</code>	der Urlader („Boot loader“) <code>syslinux</code>
<code>syslinux.cfg</code>	Konfigurationsdatei für <code>syslinux</code>
<code>kernel</code>	Linux-Kernel
<code>rootfs.img</code>	RootFS: enthält zum Booten nötige Programme
<code>opt.img</code>	Optionale Dateien: Treiber und Pakete
<code>rc.cfg</code>	Konfigurationsdatei mit den benutzten Variablen aus den Dateien des Konfigurationsverzeichnisses
<code>boot.msg</code>	Texte für das <code>syslinux</code> -Bootmenü
<code>boot_s.msg</code>	Texte für das <code>syslinux</code> -Bootmenü
<code>boot_z.msg</code>	Texte für das <code>syslinux</code> -Bootmenü
<code>hd.cfg</code>	Konfigurationsdatei zur Zuordnung der Partitionen

Durch das Skript `mkfli4l.sh` (bzw. `mkfli4l.bat`) werden zunächst die Dateien `opt.img`, `syslinux.cfg` und `rc.cfg` sowie das `rootfs.img` erzeugt. Die dafür nötigen Dateien ermittelt das Programm `mkfli4l` (im `unix-` bzw. `windows-`Unterverzeichnis). In den beiden Archiven sind die benötigten Kernel- und andere Pakete enthalten. Die Datei `rc.cfg` befindet sich sowohl im Opt-Archiv als auch auf dem Boot-Datenträger.<sup>15</sup>

Anschließend werden die Dateien `kernel`, `rootfs.img`, `opt.img` und `rc.cfg` zusammen mit den `syslinux`-Dateien auf den Datenträger kopiert.

Beim Booten von `fli4l` wird über das Skript `/etc/rc` die `rc.cfg` ausgewertet und das komprimierte `opt.img`-Archiv in die RootFS-RAM-Disk integriert (je nach Installationstyp werden dabei die Dateien direkt in die RootFS-RAM-Disk entpackt oder über symbolische Verknüpfungen eingebunden). Danach werden die Skripte in `/etc/rc.d/` in alphanumerischer Reihenfolge ausgeführt und somit die Treiber geladen und die Dienste gestartet.

## 1.10 Konfigurationsdateien

Hier werden die Dateien kurz aufgeführt, die vom `fli4l`-Router on-the-fly beim Booten erzeugt werden.

### 1. Konfiguration Provider

- `etc/ppp/pap-secrets`
- `etc/ppp/chap-secrets`

### 2. Konfiguration DNS

- `etc/resolv.conf`
- `etc/dnsmasq.conf`
- `etc/dnsmasq_dhcp.conf`

<sup>15</sup>Die Fassung im Opt-Archiv ist während der frühen Boot-Phase nötig, weil zu diesem Zeitpunkt das Boot-Volume noch nicht eingehängt ist.



- `etc/resolv.dnsmasq`
3. Hosts-Datei
    - `etc/hosts`
  4. imond-Konfiguration
    - `etc/imond.conf`

### 1.10.1 Konfiguration Provider

Für den ausgesuchten Provider wird in `etc/ppp/pap-secrets` die User-ID und das Passwort angepasst.

Beispiel für Provider Planet-Interkom:

```
# Secrets for authentication using PAP
# client      server  secret                IP addresses
"anonymer"    *        "surfer"              *
```

Dabei ist im Beispiel „anonymer“ die USER-ID. Als Remote-Server wird prinzipiell jeder erlaubt (deshalb „\*“). „surfer“ ist das Passwort für den Provider Planet-Interkom.

### 1.10.2 Konfiguration DNS

Man kann den fli4l-Router als DNS-Server einsetzen. Warum dies sinnvoll und bei Windows-Rechnern im LAN sogar zwingend notwendig ist, wird in der Dokumentation des „base“-Pakets erläutert.

Die Resolver-Datei `etc/resolv.conf` enthält den Domainnamen und den zu verwendenden Nameserver. Sie hat folgenden Inhalt (wobei „domain.de“ nur ein Platzhalter für den Wert der Konfigurationsvariable `DOMAIN_NAME` ist):

```
search domain.de
nameserver 127.0.0.1
```

Der DNS-Server `dnsmasq` wird über die Datei `etc/dnsmasq.conf` konfiguriert. Sie wird beim Booten vom Skript `rc040.dns-local` sowie `rc370.dnsmasq` automatisch erzeugt und könnte wie folgt aussehen:

```
user=dns
group=dns
resolv-file=/etc/resolv.dnsmasq
no-poll
no-negcache
bogus-priv
log-queries
domain-suffix=lan.fli4l
local=/lan.fli4l/
domain-needed
expand-hosts
filterwin2k
conf-file=/etc/dnsmasq_dhcp.conf
```

### 1.10.3 Hosts-Datei

Diese Datei enthält eine Zuordnung von Host-Namen zu IP-Adressen. Diese Zuordnung ist jedoch nur lokal auf dem fli4l verwendbar, für andere Rechner im LAN ist sie nicht sichtbar. Diese Datei ist eigentlich überflüssig, wenn zusätzlich ein lokaler DNS-Server gestartet wird.

### 1.10.4 imond-Konfiguration

Die Datei `etc/imond.conf` wird unter anderem aus den Konfigurationsvariablen `CIRC_x_NAME`, `CIRC_x_ROUTE`, `CIRC_x_CHARGEINT` und `CIRC_x_TIMES` konstruiert. Sie kann aus bis zu 32 Zeilen (ohne die Kommentarzeilen) bestehen. Jede Zeile besteht aus acht Spalten:

1. Bereich Wochentag bis Wochentag
2. Bereich Stunde bis Stunde
3. Device (`ipppX` oder `isdnX`)
4. Circuit mit Default-Route: „yes“/„no“
5. Telefonnummer
6. Name des Circuits
7. Telefonkosten pro Minute in Euro
8. Zeittakt („Charge interval“) in Sekunden

Hier ein Beispiel:

#day	hour	device	defroute	phone	name	charge	ch-int
Mo-Fr	18-09	ippp0	yes	010280192306	Addcom	0.0248	60
Sa-Su	00-24	ippp0	yes	010280192306	Addcom	0.0248	60
Mo-Fr	09-18	ippp1	yes	019160	Compuserve	0.019	180
Mo-Fr	09-18	isdn2	no	0221xxxxxxx	Firma	0.08	90
Mo-Fr	18-09	isdn2	no	0221xxxxxxx	Firma	0.03	90
Sa-Su	00-24	isdn2	no	0221xxxxxxx	Firma	0.03	90

Weitere Erklärungen zum Least-Cost-Routing findet man in der Dokumentation des „base“-Pakets.

### 1.10.5 Die `/etc/.profile`-Datei

Die Datei `/etc/.profile` enthält benutzerdefinierte Einstellungen für die Shell. Um die Standard-Einstellungen zu überschreiben, ist es nötig, unterhalb seines Konfigurationsverzeichnis eine Datei `etc/.profile` zu erstellen. Dort können dann Einstellungen zum Prompt oder Abkürzungen (so genannte „Aliase“) eingetragen werden.

**Wichtig:** Diese Datei darf kein *exit* enthalten!

Beispiel:

```
alias ll='ls -al'
```

### 1.10.6 Skripte in `/etc/profile.d/`

In dem Verzeichnis `/etc/profile.d/` können Skripte abgelegt werden, die beim Starten einer Shell ausgeführt und damit die Umgebung für die Shell beeinflussen können. Typischerweise platzieren OPTs dort Skripte, welche spezielle Umgebungsvariablen setzen, die für die Programme des OPTs notwendig sind.

Falls sich sowohl Skripte in `/etc/profile.d/` befinden als auch die Datei `/etc/.profile` existiert, werden die Skripte in `/etc/profile.d/` *nach* dem Skript `/etc/.profile` ausgeführt.

## 1.11 Inkompatibilitäten zwischen 3.x und 4.x

Beim Umstellen von Paketen von Version 3.x auf 4.x sind die folgenden Dinge zu beachten:

- Die zugrunde liegende `µClibc`-Bibliothek ist aktualisiert worden. Deswegen sollten alle Binärprogramme, die gegen eine ältere `µClibc` gebunden wurden, neu übersetzt werden. Das dafür nötige FBR (*fli4l Buildroot*) ist im *src*-Paket zu finden.
- Das Circuit-Konzept ist stark ausgebaut worden, so dass die Skripte in `/etc/ppp` häufiger aufgerufen werden als vorher (z.B. auch für alle konfigurierten Routen). Des Weiteren können jetzt dieselben Skripte für verschiedene Circuits parallel ausgeführt werden. Greift ein `ip-up`- oder ein `ip-down`-Skript auf eine globale Ressource zu (z.B. eine Datei, die von allen Circuits gleichermaßen genutzt wird), muss diese Ressource für die Dauer der Bearbeitung gesperrt und hinterher wieder freigegeben werden. Die dafür nötigen Funktionen `sync_lock_resource` und `sync_unlock_resource` werden in Abschnitt ?? beschrieben.
- Der Circuit und das Gerät `pppoe` existiert nur noch aus Kompatibilitätsgründen und sollte nicht mehr benutzt werden. Es repräsentiert den ersten konfigurierten PPPoE-Circuit bzw. die ihm zugrunde liegende Netzwerkschnittstelle. PPP-Circuits, die *keine* PPPoE-Circuits sind (dies trifft z.B. auf PPTP-, Fritz!DSL- oder UMTS-Circuits zu), werden über diesen Alias *nicht* mehr repräsentiert. Des Weiteren sollte in den Circuit-Skripten unterhalb von `/etc/ppp` die Variable `real_interface` nicht mehr genutzt werden. Statt dessen sollte die Variable `interface` verwendet werden. Eine Abbildung von der tatsächlichen PPP-Schnittstelle auf `pppoe` findet nicht mehr statt, da nicht mehr davon ausgegangen werden darf, dass es höchstens einen DSL-Circuit gibt. Es ist somit auch keine Prüfung mehr vorzunehmen, ob die Variable `interface` den Wert `pppoe` beinhaltet.

# Abbildungsverzeichnis

# Tabellenverzeichnis

1.1	Parameter für <code>mkfli4l</code> . . . . .	6
1.2	Optionen für Dateien . . . . .	10
1.3	Logische Ausdrücke . . . . .	33
1.4	Funktionen des <code>cgi-helper</code> -Skriptes . . . . .	48

# Index

DEBUG\_ENABLE\_CORE, [42](#)  
DEBUG\_IP, [42](#)  
DEBUG\_IPUP, [42](#)  
DEBUG\_KEEP\_BOOTLOGD, [42](#)  
DEBUG\_MDEV, [42](#)  
  
LOG\_BOOT\_SEQ, [42](#)