

fli4l – Developer documentation

Version 4.0.0-testing-x86-r60732

Frank Meyer
email: frank@fli4l.de

the fli4l-Team
email: team@fli4l.de

August 31, 2022

Contents

| | | |
|----------|--|----------|
| 1 | Documentation for Developers | 4 |
| 1.1 | Common Rules | 4 |
| 1.2 | Compiling Programs | 4 |
| 1.3 | Module Concept | 5 |
| 1.3.1 | mkfli4l | 5 |
| 1.3.2 | Structure | 5 |
| 1.3.3 | Configuration of Packages | 7 |
| 1.3.4 | List of Files to Copy | 7 |
| 1.3.5 | Checking Configuration Variables | 11 |
| 1.3.6 | Own Definitions for Checking the Configuration Variables | 13 |
| 1.3.7 | Extended Checks of the Configuration | 19 |
| 1.3.8 | Support for Different Kernel Version Lines | 34 |
| 1.3.9 | Documentation | 34 |
| 1.3.10 | File Formats | 36 |
| 1.3.11 | Developer Documentation | 36 |
| 1.3.12 | Client Programs | 36 |
| 1.3.13 | Source Code | 36 |
| 1.3.14 | More Files | 37 |
| 1.4 | Creating Scripts for fli4l | 37 |
| 1.4.1 | Structure | 37 |
| 1.4.2 | Handling of Configuration Variables | 38 |
| 1.4.3 | Persistent Data Storage | 38 |
| 1.4.4 | Debugging | 39 |
| 1.4.5 | Hints | 40 |
| 1.5 | Using The Packet Filter | 41 |
| 1.5.1 | Adding Own Chains And Rules | 41 |
| 1.5.2 | Integrating Into Existing Rules | 41 |
| 1.5.3 | Extending The Packet Filter Tests | 42 |
| 1.6 | CGI-Creation for Package <i>httpd</i> | 44 |
| 1.6.1 | General information about the web server | 44 |
| 1.6.2 | Script Names | 44 |
| 1.6.3 | Menu Entries | 44 |
| 1.6.4 | Construction of a CGI script | 45 |
| 1.6.5 | Miscellaneous | 50 |
| 1.6.6 | Debugging | 50 |
| 1.7 | Boot, Reboot, Dialin And Hangup Under fli4l | 51 |
| 1.7.1 | Boot Concept | 51 |
| 1.7.2 | Start And Stop Scripts | 51 |
| 1.7.3 | Helper Functions | 53 |
| 1.7.4 | mdev-Rules | 55 |

Contents

| | | |
|------------------------|---|-----------|
| 1.7.5 | ttyI Devices | 58 |
| 1.7.6 | Dialin And Hangup Scripts | 59 |
| 1.8 | Package “template” | 60 |
| 1.9 | Structure of the Boot Medium | 60 |
| 1.10 | Configuration Files | 61 |
| 1.10.1 | Provider Configuration | 61 |
| 1.10.2 | DNS Configuration | 61 |
| 1.10.3 | Hosts File | 62 |
| 1.10.4 | imond Configuration | 62 |
| 1.10.5 | The File <code>/etc/.profile</code> | 63 |
| 1.10.6 | Scripts in <code>/etc/profile.d/</code> | 63 |
| List of Figures | | 64 |
| List of Tables | | 65 |
| Index | | 66 |

1 Documentation for Developers

1.1 Common Rules

In order to include a new package in the OPT database on the fli4l homepage some rules must be obeyed. Packages that do not comply with these rules may be removed from the database without warning.

1. *No* file copy actions by the user! fli4l provides a sophisticated system to include the fli4l packages into the installation archives. All files that should go to the router are located in `opt/`.
2. Pack and compress packets properly: packages must be constructed in a way to allow effortless unpacking into the fli4l directory.
3. Packages must be *completely* configurable via the config file. Further editing of configuration files shall not be required by the user. Keep difficult decisions away from the users and move them to another part at the end of the configuration file. *Note: ONLY MODIFY IF YOU KNOW WHAT YOU DO.*
4. Another hint for config files: by its name it has to be seen clearly which OPT it belongs to. For example `OPT_HTTPD` contains the variables `OPT_HTTPD`, `HTTPD_USER_N`, a.s.o.
5. Please built binaries as small as possible! If you compile them yourself in FBR please remember to deactivate any unneeded feature.
6. Check for Copyrights! If using template files please keep in mind to change the Copyright accordingly. This applies also for config-, check- and Opt text files. Replace the Copyright with your own name. A documentation that was only copied of course has to keep the Copyright of the original author!
7. Please use only free formats as archive types. These are:
 - ZIP (`.zip`)
 - GZIP (`.tgz` or `.tar.gz`)

Please don't use other formats like RAR, ACE, Blackhole, LHA ... Windows-Installer files (`.msi`) or self-extracting archives and installers (`.exe`) should *not* be used.

1.2 Compiling Programs

The environment needed for compiling programs is available in the separate package "src". There it is also documented how to compile own programs for fli4l.

1.3 Module Concept

As of version 2.0 fli4l is split into modules (packages), i.e.

- fli4l-4.0.0-testing-x86-r60732 <— The Base Package
- dns-dhcp
- dsl
- isdn
- sshd
- and much more...

With the base package fli4l acts as a pure Ethernet router. For ISDN and/or DSL the packages isdn and/or dsl have to be unpacked to the fli4l directory. The same applies for the other packages.

1.3.1 mkfli4l

Depending on the current configuration a file called `rc.cfg` and two archives `rootfs.img` and `opt.img` will be generated which contain all required configuration informations and files. These files are generated using `mkfli4l` which reads the individual package files and checks for configuration errors.

`mkfli4l` will accept the parameters listed in table 1.1. If omitted the default values noted in brackets are used. A complete list of all options (Table 1.1) is displayed when executing

```
mkfli4l -h
```

.

1.3.2 Structure

A package can contain multiple OPTs, if it contains only one, however, it is appropriate to name the package like the OPT. Below `<PACKAGE>` is to be replaced by the respective package name. A package consists of the following parts:

- Administrative Files
- Documentation
- Developer Documentation
- Client Programs
- Source Code
- More Files

The individual parts are described in more detail below.

Table 1.1: Parameters for mkfli41

| Option | Meaning | |
|---------------|--|--------------------------------------|
| -c, --config | Declaration of the directory mkfli41 will scan for package config files (default: config) | |
| -x, --check | Declaration of the directory mkfli41 will scan for files needed for package error checking (<package>.txt, <package>.exp and <package>.ext; default: check) | |
| -l, --log | Declaration of the log file to which mkfli41 will log error messages and warnings (default: img/mkfli41.log) | |
| -p, --package | Declaration of the packages to be checked, this option may be used more than once in case of a desired check for several packages in conjunction. If using -p, however, the file <check_dir>/base.exp will always be read first to provide the common regular expressions provided by the base package. Hence, this file must exist. | |
| -i, --info | Provides information on the check (which files are read, which tests are run, which uncommon things happened during the review process) | |
| -v, --verbose | More verbose variant of option -i | |
| -h, --help | Displays the help text | |
| -d, --debug | Allows for debugging the review process. This is meant to be a help for package developers wishing to know in detail how the process of package checking is working. | |
| | Debug Option | Meaning |
| | 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 opt/<package>.txt |
| | ext-trace | show trace of extended checks |

1.3.3 Configuration of Packages

The user's changes to the package's configuration are made in the file `config/<PACKAGE>.txt`. All the OPT's variables should begin with the name of the OPT, for example:

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

An OPT should be prefixed by a header in the configuration file (see above). This increases readability, especially as a package indeed can contain multiple OPTs. Variables associated to the OPT should — again in the interest of readability — not be indented further. Comments and blank lines are allowed, with comments always starting in column 33. If a variable including its content has more than 32 characters, the comment should be inserted with a row offset, starting in column 33. Longer comments are spread over multiple lines, each starting at column 33. All this increases easy review of the configuration file.

All values following the equal sign must be enclosed in quotes¹ not doing so can lead to problems during system boot.

Activated variables (see below), will be transferred to `rc.cfg`, everything else will be ignored. The only exceptions are variables by the name of `<PACKAGE>_DO_DEBUG`. These are used for debugging and are transferred as is.

1.3.4 List of Files to Copy

The file `opt/<PACKAGE>.txt` contains instructions that describe

- which files are owned by the OPT,
- the preconditions for inclusion in the `opt.img` resp. `rootfs.img`,
- what User ID (uid), Group ID (gid) and rights will be applied to files,
- which conversions have to be made before inclusion in the archive.

Based on this information `mkfli41` will generate the archives needed.

Blank lines and lines beginning with “#” are ignored. In one of the first lines the version of the package file format should be noted as follows:

```
<first column>      <second column> <third column>
opt_format_version  1                      -
```

The remaining lines have the following syntax:

```
<first column> <second column> <third column> <columns following>
Variable      Value           File           Options
```

¹Both single and double quotes are valid. You can hence write `F00='bar'` as well as `F00="bar"`. The use of double quotes should be an exception and you should previously inform about how an *nix shell uses single and double quotes.

1. The first column contains the name of a variable which triggers inclusion of the file referenced in the third column depending on its value in the package's config file. The name of a variable may appear in the first column as often as needed if multiple files depend on it. Any variable that appears in the file `opt/<PACKAGE>.txt` is marked by `mkfli41`.

If multiple variables should be tested for the same value a list of variables (separated by commas) can be used instead. It is sufficient in this case if at least *one* variable contains the value required in the second column. It is important *not* to use spaces between the individual variables!

In OPT variables (ie variables that begin with `OPT_` and typically have the type `YESNO`), the prefix "`OPT_`" can be omitted. It does not matter whether variables are noted in upper- or lowercase (or mixed).

2. The second column contains a value. If the variable in the first column is identical with this value and is activated too (see below), the file referenced in the third column will be included. If the first column contains a %-variable it will be iterated over all indices and checked whether the respective variable matches the value. If this is the case copying will be executed. In addition, the copy process based on the current value of the variable will be logged.

It is possible to write a "!" in front of the value. In this case, the test is negated, meaning the file is only copied if the variable does *not* contain the value.

3. In the third column a file name is referenced. The path must be given relative to the `opt` directory. The file must exist and be readable, otherwise an error is raised while generating the boot medium and the build process is aborted.

If the file name is prefixed with a "`rootfs:`" the file is included in the list of files to be copied to the RootFS. The prefix will be stripped before.

If the file is located below the current configuration directory it is added to the list of files to be copied from there, otherwise the file found below `opt` is taken. Those files are not allowed to have a `rootfs:` prefix.

If the file to copy is a kernel module the actual kernel version may be substituted by `${KERNEL_VERSION}`. `mkfli41` will then pick the version from the configuration and place it there. Using this you may provide modules for several kernel versions for the package and the module matching the current kernel version will be copied to the router. For kernel modules the path may be omitted, `mkfli41` will find the module using `modules.dep` and `modules.alias`, see the section "[Automatically Resolving Dependencies for Kernel Modules](#)" (Page 11).

4. the other columns may contain the options for owner, group, rights for files and conversion listed in table 1.2.

Some examples:

- copy file if `OPT_TELNETD='yes'`, set its uid/gid to root and the rights to 755 (`rwxr-xr-x`)

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

Table 1.2: Options for Files

| Option | Meaning | Default Value |
|-------------------------------|--|--|
| type= | Type of the Entry: <i>local</i> Filesystem Object <i>file</i> File <i>dir</i> Directory <i>node</i> Device <i>symlink</i> (symbolic) Link This option has to be placed in front when given. The type “local” represents the type of an object existing in the file system and hence matches “file”, “dir”, “node” or “symlink” (depends). All other types except for “file” can create entries in the archive that do not have to exist in the local file system. This can i.e. be used to create devices files in the RootFS archive. | local |
| uid= | The file owner, either numeric or as a name from passwd | root |
| gid= | File group, either numeric or as a name from group | root |
| mode= | Access rights | Files and Devices: rw-r--r-- (644) Directories: rwxr-xr-x (755) Links: rwxrwxrwx (777) |
| flags= (type=file) | Conversions before inclusion in the archive: <i>utxt</i> Conversion to *nix format <i>dtxt</i> Conversion to DOS format <i>sh</i> Shell script: Conversion to *nix format, stripping of superfluous chars <i>luac</i> Lua script: Translation into byte code of the Lua VM | |
| name= | Alternative name for inclusion of the entry in the archive | |
| devtype= (type=node) | Describes the type of the device (“c” for character and “b” für block oriented devices). Has to be placed in second position. | |
| major= (type=node) | Describes the so-called “Major” number of the device file. Has to be placed in third position. | |
| minor= (type=node) | Describes the so-called “Minor” number of the device file. Has to be placed in fourth position. | |
| linktarget= (type=symlink) | Describes the target of the symbolic link. Has to be placed in second position. | |

- copy file, set uid/gid to root, the rights to 555 (**r-xr-xr-x**) and convert the file to *nix format while stripping all superfluous chars

1 Documentation for Developers

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

- copy file if PCMCIA_PCIC='i82365', set uid/gid to root and the rights to 644 (rw-r--r--)

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

- copy file if one of the NET_DRV_% variables matches the second field, set uid/gid to root and the rights to 644 (rw-r--r--)

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

- copy file if the variable POWERMANAGEMENT does *not* contain the value “none”:

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

- copy file if any of the OPT variables OPT_MYOPTA or OPT_MYOPTB contains the value “yes”:

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

This example is only an abbreviation for:

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

And the latter is a shorthand notation for:

```
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
```

- copy file opt/usr/bin/beep.sh to the RootFS archive, but rename it to bin/beep before:

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

The files will be copied only if the above conditions are met and OPT_PACKAGE='yes' of the corresponding package is set. What OPT variable is referenced is described in the file check/<PACKAGE>.txt.

If a variable is referenced in a package that is not defined by the package itself, it may happen that the corresponding package is not installed. This would result in an error message from mkfli41, as it expects that all of the variables referenced by opt/<PACKAGE>.txt are defined.

To handle this situation correctly the “weak” declaration has been introduced. It has the following format:

```
weak      <Variable>    -
```

By this the variable it is defined (if not already existing) and its value is set to “undefined”. Please note that the prefix “OPT_” *must* be provided (if existing) because else a variable *without* this prefix will be created.

An example taken from `opt/rrdtool.txt`:

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

Without the **weak** definition `mkfli4l` would display an error message when using the package “rrdtool” while the “openvpn” package is not activated. By using the **weak** definition no error message is raised even in the case that the “openvpn” package does not exist.

Files adapted by Configuration

In some situations it is desired to replace original files with configuration-specific files for inclusion in the archive, i.e. host keys, own firewall scripts, ... `mkfli4l` supports this scenario by checking whether a file can be found in the configuration directory and, if so, including this one instead in the file list for `opt.img` resp. `rootfs.img`.

Another option to add configuration-specific files to an archive is described in the section [Extended Checks of the Configuration](#) (Page 28).

Automatically Resolving Dependencies for Kernel Modules

Kernel modules may depend on other kernel modules. Those must be loaded before and therefore also have to be added to the archive. `mkfli4l` resolves this dependencies based on `modules.dep` and `modules.alias` (two files generated during the kernel build), automatically including all required modules in the archives. Thus, for example the following entry

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

triggers that both `8390.ko` and `crc32.ko` are included in the archive because `ne2k_pci` depends on both of them.

The necessary entries from `modules.dep` and `modules.alias` are included in the RootFS and can be used by `modprobe` for loading the drivers.

1.3.5 Checking Configuration Variables

By the help of `check/<PACKAGE>.txt` the content of variables can be checked for validity. In earlier version of the program `mkfli4l` this check was hard coded there but it was outsourced to the check files in the course of modularizing `fli4l`. This file contains a line for each variable in the config files. These lines consist of four to five columns which have the following functions:

1. Variable: this column specifies the name of the configuration file variable to check. If this is an *array variable*, it can appear multiple times with different indices, so instead of the index number a percent sign (%) is added to the variable name. It is always used as “_” in the middle of a name resp. “_” at the end of a name. The name may contain more than one percent sign allowing the use of multidimensional arrays. It is recommended (but not mandatory) to add some text between the percent signs to avoid weird names like “FOO_%%”.

Often the problem occurs that certain variables describe options that are needed only in some situations. Therefore variables may be marked as optional. Optional variables are identified by the prefix “+”. They may then exist, but do not have to. Arrays can also use a “++” prefix. Prefixed with a “+” the array can exist or be entirely absent. Prefixed with a “++” in addition some elements of the array may be missing.

2. **OPT_VARIABLE:** This column assigns the variable to a specific OPT. The variable is checked for validity only if the OPT variable is set to “yes”. If there is no OPT variable a “-” indicates this. In this case, the variable must be defined in the configuration file, unless a default value is defined (see below). The name of the OPT variable may be arbitrary but should start with the prefix “OPT_”.

If a variable does not depend on any OPT variables, it is considered *active*. If it is depending on an OPT variable, it is precisely active if

- its OPT variable is active and
- its OPT variable contains the value “yes”.

In all other cases the variable is inactive.

Hint: Inactive OPT variables will be set back to “no” by `mkfli4l` if set to “yes” in the configuration file, an appropriate warning will be generated then (i.e. `OPT_Y='yes'` is ignored, because `OPT_X='no'`). For transitive dependency chains (`OPT_Z` depends on `OPT_Y` which in turn depends on `OPT_X`) this will only work reliable, if the names of all OPT-variables start with “OPT_”.

3. **VARIABLE_N:** If the first column contains a variable with a “%” in its name, it indicates the number of occurrences of the variable (the so-called *N-variable*). In case of a multi-dimensional variable, the occurrences of the last index are specified. If the variable depends on a certain OPT, the N-variable must be dependant on the same or no OPT. If the variable does not depend on any OPT, the N-variable also shouldn’t. If no N-variable exists, specify “-” to indicate that.

For compatibility with future versions of `fli4l` the variable specified here *must* be identical with the variable in **OPT_VARIABLE** where the last “%” is replaced by an “N” and everything following is removed. An array `HOST_%_IP4` must have the N-Variable `HOST_N` assigned and an array `PF_USR_CHAIN_%_RULE_%` hence the N-variable `PF_USR_CHAIN_%_RULE_N`, and this N-variable itself is an array variable with the corresponding N-variable `PF_USR_CHAIN_N`. *All other namings of the N variables will be incompatible with future versions of fli4l!*

4. **VALUE:** This column provides the values a variable can hold. For example the following settings are possible:

| Name | Meaning |
|----------|--|
| NONE | No error checking will be done |
| YESNO | The variable must be “yes” or “no” |
| NOTEMPTY | The variable can’t be empty |
| NOBLANK | The variable can’t contain spaces |
| NUMERIC | The variable must be numeric |
| IPADDR | The variable must be an IP address |
| DIALMODE | The variable must be “on”, “off” or “auto” |

If values are prefixed by “WARN_” an illegal content will not raise an error message and abort the build by `mkfli41`, but only display a warning.

The possible checks are defined by regular expressions in `check/base.exp`. This file may be extended and now contains some new checking routines, for example: `HEX`, `NUMHEX`, `IP_ROUTE`, `DISK` and `PARTITION`.

The number of expressions may be extended at any time for the future needs of package developers. Provide feedback!

In addition, regular expressions can also be directly defined in the check-files, even relations to existing expressions can be made. Instead of `YESNO` you could, for example also write

```
RE:yes|no.
```

This is useful if a test is performed only once and is relatively easy. For more details see the next chapter.

5. Default Setting: In this column, an optional default value for the variables can be defined in the case that the variable is not specified in the configuration file.

Hint: At the moment this does not work for array variables. Additionally, the variable can't be optional (no “+” in front of the variable name).

Example:

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

If `OPT_TELNETD` is missing in the config file, “no” will be assumed and written as a value to `rc.cfg`.

The percent sign thingie is best described with an example. Let's assume `check/base.txt` amongst others has the following content:

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

This means that depending on the value of `NET_DRV_N` the variables `NET_DRV_N`, `NET_DRV_1_OPTION`, `NET_DRV_2_OPTION`, `NET_DRV_3_OPTION`, a.s.o. will be checked.

1.3.6 Own Definitions for Checking the Configuration Variables

Introduction of Regular Expressions

In version 2.0 only the above mentioned value ranges for variable checks existed: `NONE`, `NOTEMPTY`, `NUMERIC`, `IPADDR`, `YESNO`, `NOBLANK`, `DIALMODE`. Checking was hard-coded to `mkfli41`, not expandable and restricted to essential “data types” which could be evaluated with reasonable efforts.

As of version 2.1 this checking has been reimplemented. The aim of the new implementation is a more flexible testing of variables, that is also able to examine more complex expressions. Therefore, regular expressions are used that can be stored in one or more separate files. This on one hand makes it possible to examine variables that are not checked for the moment and on the other hand, developers of optional packages can now define own terms in order to check the configuration of their packages.

A description of regular expressions can be found via “man 7 regex” or i.e. here: <http://unixhelp.ed.ac.uk/CGI/man-cgi?regex+7>.

Specification of Regular Expressions

Specification of regular expressions can be accomplished in two ways:

1. Package specific exp files `check/<PACKAGE>.exp`

This file can be found in the `check` directory and has the same name as the package containing it, i.e. `check/base.exp`. It contains definitions for expressions that can be referenced in the file `check/<PACKAGE>.txt`. `check/base.exp` for example at the moment contains definitions for the known tests and `check/isdn.exp` a definition for the variable `ISDN_CIRC_x_ROUTE` (the absence of this check was the trigger for the changes).

The syntax is as follows (again, double quotes can be used if needed):

```
<Name> = '<Regular Expression>' : '<Error Message>'
```

as an example `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 CIDR netmask,
            for instance 192.168.6.0/24'
```

The regular expressions can also include already existing definitions by a reference. These are then pasted to substitute the reference. This makes it easier to construct regular expressions. The references are inserted by '(RE: Reference)'. (See the definition of the term `NETWORKS` above for an appropriate example.)

The error messages tend to be too long. Therefore, they may be displayed on multiple lines. The lines afterwards always have to start with a space or tab then. When reading the file `check/<PACKAGE>.exp` superfluous whitespaces are reduced to one and tabs are replaced by spaces. An entry in `check/<PACKAGE>.exp` could look like this:

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

2. Regular expressions directly in the check file `check/<PACKAGE>.txt`

Some expressions occur but once and are not worth defining a regular expression in a `check/<PACKAGE>.exp` file. You can simply write this expression to the check file for example:

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

MOUNT_BOOT can only take the value “ro”, “rw” or “no”, everything else will be denied.

If you want to refer to existing regular expressions, simply add a reference via ‘(RE:...)’.

Example:

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

Expansion of Existing Regular Expressions

If an optional package adds an additional value for a variable which will be examined by a regular expression, then the regular expression has to be expanded. This is done simply by defining the new possible values by a regular expression (as described above) and complement the existing regular expression in a separate `check/<PACKAGE>.exp` file. That an existing expression is modified is indicated by a leading “+”. The new expression complements the existing expression by appending the new value to the existing value(s) as an alternative. If another expression makes use of the complemented expression, the supplement is also there. The specified error message is simply appended to the end of the existing one.

Using the Ethernet driver as an example this could look like here:

- The base packages provides a lot of Ethernet drivers and checks the variable `NET_DRV_x` using the regular expression `NET_DRV`, which is defined as follows:

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

- The package “pcmcia” provides additional device drivers, and hence has to complement `NET_DRV`. This is done as follows:

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

Now PCMCIA drivers can be chosen in addition.

Extend Regular Expressions in Relation to YESNO Variables

If you have extended `NET_DRV` with the PCMCIA drivers as shown above, but the package “pcmcia” has been deactivated, you still could select a PCMCIA driver in `config/base.txt` without an error message generated when creating the archives. To prevent this, you may let the regular expression depend on a `YESNO` variable in the configuration. For this purpose, the name of the variable that determines whether the expression is extended is added with brackets immediately after the name of the expression. If the variable is active and has the value “yes”, the term is extended, otherwise not.

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

If specifying `OPT_PCMCIA='no'` and using i.e. the PCMCIA driver `xirc2ps_cs` in `config/base.txt`, an error message will be generated during archive build.

Hint: This does *not* work if the variable is not set explicitly in the configuration file but gets its value by a default setting in `check/<PACKAGE>.txt`. In this case the variable hence has to be set explicitly and the default setting has to be avoided if necessary.

Extending Regular Expressions Depending on other Variables

Alternatively, you may also use arbitrary values of variables as conditions, the syntax looks like this:

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

If `KERNEL_VERSION` matches the given regular expression (if any of the kernels of the 3.18 line is used) then the list of network driver allowed is extended with the drivers mentioned.

Hint: This does *not* work if the variable is not set explicitly in the configuration file but gets its value by a default setting in `check/<PACKAGE>.txt`. In this case the variable hence has to be set explicitly and the default setting has to be avoided if necessary.

Error Messages

If the checking process detects an error, an error message of the following kind is displayed:

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

For the first error, the term was defined in a `check/<PACKAGE>.exp` file and an explanation of the error is displayed. In the second case the term was specified directly in a `check/<PACKAGE>.txt` file, so there is no additional explanation of the error cause.

Definition of Regular Expressions

Regular expressions are defined as follows:

Regular expression: One or more alternatives, separated by '|', i.e. “ro|rw|no”. If one option matches, the whole term matches (in this case “ro”, “rw” and “no” are valid expressions).

An alternative is a concatenation of several sections that are simply added.

A section is an “atom”, followed by a single “*”, “+”, “?” or “{min, max}”. The meaning is as follows:

- “a*” — as many “a”s as wished (allows also no “a” is existing at all)
- “a+” — at least one “a”
- “a?” — none or one “a”
- “a{2,5}” — two to five “a”s

- “a{5}” — exactly five “a”s
- “a{2,}” — at least two “a”s
- “a{,5}” — a maximum of five “a”s

An “atom” is a

- regular expression enclosed in brackets, for example “(a|b)+” matches any string containing at least one “a” or “b”, up to an arbitrary number and in any order
- an empty pair of brackets stands for an “empty” expression
- an expression in square brackets “[]” (see below)
- a dot “.”, matching an arbitrary character, for example a “.+” matches any string containing at least one char
- a “^” represents the beginning of a line, for example a “^a.*” matches a string beginning with an “a” followed by any char like in “a” or “adkadhashdkash”
- a “\$” represents the end of a line
- a “\” followed by one of the special characters ^ . [\$ () | * + ? { \ stands for the second char without its special meaning
- a normal char matches exactly this char, for example “a” matches exactly an “a”.

An expression in square brackets indicates the following:

- “x-y” — matches any char inbetween “x” and “y”, for example “[0-9]” matches all chars between “0” and “9”; “[a-zA-Z]” symbolizes all chars, either upper- or lowercase.
- “^ x-y” — matches any char *not* contained in the given interval, for example “[^0-9]” matches all chars *except* for digits.
- “[*character-class*:]” — matches a char from *character-class*. Relevant standard character classes are: `alnum`, `alpha`, `blank`, `digit`, `lower`, `print`, `punct`, `space`, `upper` and `xdigit`. I.e. “[[:alpha:]]” stands for all upper- or lowercase chars and hence is identical with “[[:lower:]][[:upper:]]”.

Examples for regular Expressions

Let’s have a look at some examples!

NUMERIC: A numeric value consists of at least one, but otherwise any number of digits. “At least one” is expressed with a “+”, one digit was already in an example above. So this results in:

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

or alternatively

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

NOBLANK: A value that does not contain spaces, is any char (except for the char “space”) and any number of them:

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

or, if the value is not allowed to be empty:

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

IPADDR: Let’s have a look at an example with an IP4-address. An ipv4 address consists of four “Octets”, divided by dots (“.”). An octet is a number between 0 and 255. Let’s define an octet at first. It may be

```
a number between 0 and 9:      [0-9]
a number between 10 and 99:    [1-9][0-9]
a number between 100 and 199:  1[0-9][0-9]
a number between 200 and 249:  2[0-4][0-9]
a number between 250 and 255:  25[0-5]
```

All are alternatives hence we concatenate them with “|” forming one expression: “[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]” and get an octet. Now we compose an IP4 address, four octets divided by dots (the dot must be masked with a *backslash*, because else it would represent an arbitrary char). Based on the syntax of an exp-file it would look like this:

```
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) '
```

Assistance for the Design of Regular Expressions

If you want to design and test regular expressions, you can use the “regex” program located in the `unix` or `windows` directory of the package “base”. It accepts the following syntax:

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

The parameters explained in short:

- `<check dir>` is the directory containing check and exp files. These are read by “regex” to use expressions already defined there.
- `<regex>` is the regular expression (enclosed in ‘...’ or “...” if in doubt, with double quotes needed only if single quotes are used in the expression itself)
- `<string>` is the string to be checked

This may for example look like here:

```
./i586-linux-regex -c ../check '[0-9]' 0
adding user defined regular expression='[0-9]' ('^([0-9])$')
checking '0' against regex '[0-9]' ('^([0-9])$')
'[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 Extended Checks of the Configuration

Sometimes it is necessary to perform more complex checks. Examples of such complex things would be i.e. dependencies between packages or conditions that must be satisfied only when variables take certain values. For example if a PCMCIA ISDN adapter is used the package “pcmcia” has to be installed, too.

In order to perform these checks you may write small tests to `check/<PACKAGE>.ext` (also called ext-script). The language consists of the following elements:

1. Keywords:

- Control Flow:
 - 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
- Dependencies:
 - provides *package* version *x.y.z*
 - depends on *package* version *x1.y1 x2.y2.z2 x3.y3* ...
- Actions:
 - 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. Data Types: strings, positive integers, version numbers
3. Logical Operations: `<`, `==`, `>`, `!=`, `!`, `&&`, `||`, `==~`, `copy_pending`, `samenet`, `subnet`

Data Types

Concerning data types please note that variables, based on the associated regular expression are permanently assigned to a data type:

- Variables, starting with type “NUM” are numeric and contain positive integers
- Variables representing an N-variable for any kind of array are numeric as well
- all other variables are treated as strings

This means, among other things, that a variable of type `ENUMERIC` can *not* be used as an index when accessing an array variable, even if you have checked at first that it is not empty. The following code thus does not work as expected:

```
# TEST should be a variable of type ENUMERIC
if (test != "")
then
  # Error: You can't use a non-numeric ID in a numeric
  #       context. Check type of operand.
  set i=my_array[test]
  # Error: You can't use a non-numeric ID in a numeric
  #       context. Check type of operand.
  set j=test+2
fi
```

A solution for this problem is offered by `split` (Page 27):

```
if (test != "")
then
  # all elements of test_% are numeric
  split(test, test_%, ' ', numeric)
  # OK
  set i=my_array[test_%[1]]
  # OK
  set j=test_%[1]+2
fi
```

Substitution of Strings and Variables

At various points strings are needed, such as when a [Warning](#) (Page 23) should be issued. In some cases described in this documentation, such a string is scanned for variables. If found, these are *replaced* by their contents or other attributes. This replacement is called *variable substitution*.

This will be illustrated by an example. Assume this configuration:

```
# config/base.txt
HOSTNAME='fli41'
# config/dns_dhcp.txt
HOST_N='1' # Number of hosts
HOST_1_NAME='client'
HOST_1_IP4='192.168.1.1'
```

Then the character strings are rewritten as follows, if variable substitution is active in this context:

```
"My router is called $HOSTNAME"
# --> "My router is called fli41"
"HOSTNAME is part of the package ${HOSTNAME}"
# --> "HOSTNAME is part of the package base"
"@HOST_N is $HOST_N"
# --> " # Number of hosts is 1"
```

As you can see, there are basically three options for replacement:

- `$<Name>` resp. `${<Name>}`: Replaces the variable name with the contents of the variable. This is the most common form of replacement. The name must be enclosed in `{...}` if in the string it is directly followed by a char that may be a valid part of a variable name (a letter, a digit, or an underscore). In all other cases, the use of curly brackets is possible, but not mandatory.
- `%<Name>` resp. `%${<Name>}`: Replaces the variable name with the name of the package in which the variable is defined. This does *not* work with variables assigned in the script via `set` (Page 23) or with counting variables of a `foreach-loop` (Page 29) since such variables do not have a package and their syntax is different.
- `@<Name>` resp. `@${<Name>}`: Replaces the variable name with the comment noted in the configuration after the variable. Again, this does not make sense for variables defined by the script.

Hint: Elements of array variables can *not* be integrated into strings this way, because there is no possibility to provide an index.

In general, only *constants* can be used for variable substitution, strings that come from a variable remain unchanged. An example will make this clear - assume the following configuration:

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

This code:

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

leads to the following output:

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

It will *not* display:

```
Warning: fli41
```

In the following sections it will be explicitly noted under which conditions strings are subject of variable substitution.

Definition of a Service with an associated Version Number: provides

For instance, an OPT may declare that it provides a Printer service or a Webserver service. Only one package can provide a certain service. This prevents i.e. that two web servers are installed in parallel, which is not possible for obvious reasons, since the two servers would both register port 80. In addition, the current version of the service is provided so that updates can be triggered. The version number consists of two or three numbers separated by dots, such as “4.0” or “2.1.23”.

Services typically originate from OPTs, not from packages. For example the package “tools” has a number of programs that each have their own **provides** statement defined if activated by `OPT_...='yes'`.

The syntax is as follows:

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

Example from package “easycron”:

```
provides cron version 3.10.0
```

The version number should be incremented by the OPT-developer in the third component, if only functional enhancements have been made and the OPT’s interface is still. The version number should be increased in the first or second component when the interface has changed in any incompatible way (eg. due to variable renaming, path changes, missing or renamed utilities, etc.).

Definition of a Dependency to a Service with a specific Version: depends

If another service is needed to provide the own function (eg. a web server) this dependency to a specific version may be defined here. The version can be given with two (i.e. “2.1”) or three numbers (i.e. “2.1.11”) while the two-number version accepts all versions starting with this number and the three-number version only accepting just the specified one. A list of version numbers may also be specified if multiple versions of the service are compatible with the package.

The syntax is as follows:

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

An example: Package “server” contains:

```
provides server version 1.0.1
```

A Package “client” with the following **depends**-instruction is given:²

```
depends on server version 1.0          # OK, '1.0' matches '1.0.1'
depends on server version 1.0.1        # OK, '1.0.1' matches '1.0.1'
depends on server version 1.0.2        # Error, '1.0.2' does not match with '1.0.1'
depends on server version 1.1          # Error, '1.1' does not match with '1.0.1'
depends on server version 1.0 1.1      # OK, '1.0' matches '1.0.1'
depends on server version 1.0.2 1.1    # Error, neither '1.0.2' nor '1.1' are matching
# '1.0.1'
```

²Of course only one at a time!

Communication with the User: warning, error, fatal_error

Using these three functions users may be warned, signaled an errors or stop the test immediately. The syntax is as follows:

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

All strings passed to these funtions are subject of [variable substitution](#) (Page 20).

Assignments

If for any reason a temporary variable is required it can be created by “`set var [= value]`”. *The variable can not be a configuration variable!* ³ If you omit the “`= value`” part the variable is simply set to “yes” so it may be tested in an `if`-statement. If an assignment part is given, anything may be specified after the equal sign: normal variables, indexed variables, numbers, strings and version numbers.

Please note that by the assignment also the *type* of the temporary variable is defined. If a number is assigned `mkfli4l` “remembers” that the variable contains a number and later on allows calculations with it. Trying to do calculations with variables of other types will fail.

Example:

```
set i=1    # OK, i is a numeric variable
set j=i+1 # OK, j is a numeric variable and contains the value 2
set i="1"  # OK, i now is a string variable
set j=i+1 # Error "You can't use a non-numeric ID in a numeric
          #          context. Check type of operand."
          # --> no calculations with strings!
```

You may also define temporary arrays (see below). Example:

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

The number of array elements is kept by `mkfli4l` in the variable `prim_n`. The code above hence leads to the following output:

```
Warning: 3
```

If the right side of an assignment is a string constant, it is subject of [variable substitution](#) (Page 20) at the time of assignment. The following example demonstrates this. The code:

³This is a desired restriction: Check scripts are *not* able to change the user configuration.

```

set s="a"
set v1="$s" # v1="a"
set s="b"
set v2="$s" # v2="b"
if (v1 == v2)
then
    warning "equal"
else
    warning "not equal"
fi

```

will output “not equal”, because the variables `v1` and `v2` replace the content of the variable `s` already at the time of assignment.

Hint: A variable set in a script is visible while processing further scripts – currently there exists no such thing as local variables. Since the order of processing scripts of different packages is not defined, you should never rely on any variable having values defined by another package.

Arrays

If you want to access elements of a %-variable (of an array) you have to use the original name of the variable like mentioned in the file `check/<PACKAGE>.txt` and add an index for each “%” sign by using “[*Index*]”.

Example: If you want to access the elements of variable `PF_USR_CHAIN_%_RULE_%` you need two indices because the variable has two “%” signs. All elements may be printed for example using the following code (the `foreach`-loop is explained in [see below](#) (Page 29)):

```

foreach i in pf_usr_chain_n
do
    # only one index needed, only one '%' in the variable's name
    set j_n=pf_usr_chain_%_rule_n[i]
    # Attention: a
    # foreach j in pf_usr_chain_%_rule_n[i]
    # is not possible, hence the use of j_n!
    foreach j in j_n
    do
        # two indices needed, two '%' in the variable's name
        set rule=pf_usr_chain_%_rule_%[i][j]
        warning "Rule $i/$j: ${rule}"
    done
done

```

With this sample configuration

```

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'

```

the following output is printed:

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

Alternatively, you can iterate directly over all values of the array (but the exact indices of the entries are not always known, because this is not required):

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

That produces the following output with the sample configuration from above:

```
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'
```

The second example nicely shows the meaning of the %<Name>-syntax: Within the string %rule is substituted by the *name* of the variable in question (for example PF_USR_CHAIN_1_RULE_1), while \$rule is substituted by its *content* (i.e. ACCEPT).

Encryption of Passwords: crypt

Some variables contain passwords that should not be noted in plain text in `rc.cfg`. These variables can be encrypted by the use of `crypt` and are transferred to a format also needed on the router. Use this like here:

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

The `crypt` function is the *only* point at which a configuration variable can be changed.

Querying File Properties: stat

`stat` is used to query file properties. At the moment only file size can be accessed. If checking for files under the current configuration directory you may use the internal variable `config_dir`. The Syntax:

```
stat (<file name>, <key>)
```

The command looks like this (the parameters used are only examples):

```
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
```

The example checks whether a file exists in the current configuration directory. If `OPENVPN_1_SECRET='test'` is set in the configuration file, the loop in the first run checks for the existence of the file `etc/openvpn/test.secret` in the current configuration directory. After the call two variables are defined:

- `<Key>_res`: Result of the system call `stat()` (“OK”, if system call was successful, else the error message of the system call)
- `<Key>_size`: File size

It may for example look like this:

```
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
```

A file name passed as a string constant is subject of [variable substitution](#) (Page 20).

Search files: `fgrep`

If you wish to search a file via “`grep`”⁴ you may use the `fgrep` command. The syntax is:

```
fgrep (<File name>, <RegEx>)
```

If the file `<File name>` does not exist `mkfli41` will abort with a fatal error! If it is not sure if the file exists, test this before with `stat`. After calling `fgrep` the search result is present in an array called `FGREP_MATCH_%`, with its index x as usual ranging from one to `FGREP_MATCH_N`. `FGREP_MATCH_1` points to the whole range of the line the regular expression has matched, while `FGREP_MATCH_2` to `FGREP_MATCH_N` contain the $n-1$ th part in brackets.

A first example will illustrate the use. The file `opt/etc/shells` contains the line:

```
/bin/sh
```

The following code

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

produces this output:

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

⁴“`grep`” is a common command on *nix-like OSES for filtering text streams.

The RegEx has (only) matched with “/bin/” (only this part of the line is contained in the variable `FGREP_MATCH_1`). The first bracketed part in the expression only matches the first char after the first “/”, this is why only “b” is contained in `FGREP_MATCH_2`. The second bracketed part contains the rest after “b” up to the last “/”, hence “in” is noted in variable `FGREP_MATCH_3`.

The following second example demonstrates an usual use of `fgrep` taken from `check/base.ext`. It will be tested if all `tmpl:-`references given in `PF_FORWARD_x` are really present.

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

Both a filename value as well as a regular expression passed as a string constant are subject to [variable substitution](#) (Page 20).

Splitting Parameters: `split`

Often variables can be assigned with several parameters, which then have to be split apart again in the startup scripts. If it is desired to split these previously and perform tests on them `split` can be used. The syntax is like this:

```
split (<String>, <Array>, <Separator>)
```

The string can be specified by a variable or directly as a constant. `mkfli41` splits it where a separator is found and generates an element of the array for each part. You may iterate over these elements later on and perform tests. If nothing is found between two separators an array element with an empty string as its value is created. The exception is “ ”: Here all spaces are deleted and no empty variable is created.

If the elements generated by such a split should be in a numeric context (e.g. as indices) this has to be specified when calling `split`. This is done by the additional attribute “numeric”. Such a call looks as follows:

```
split (<String>, <Array>, <Separator>, numeric)
```

An example:

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

the output looks like this:

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

Hint: If using the “numeric” variant `mkfli41` will *not* check the generated string parts for really being numeric! If you use such a non-numeric construct later in a numeric context (i.e. in an addition) `mkfli41` will raise a fatal error. Example:

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

A string constant passed to `split` in the first parameter is subject of [variable substitution](#) (Page 20).

Adding Files to the Archives: `add_to_opt`

The function `add_to_opt` can add additional files to the Opt- or RootFS-Archives. *All* files under `opt/` or from the configuration directory may be chosen. There is no limitation to only files from a specific package. If a file is found under `opt/` as well as in the configuration directory, `add_to_opt` will prefer the latter. The function `add_to_opt` is typically used if complex logical rules decide if and what files have to be included in the archives.

The syntax looks like this:

```
add_to_opt <File> [<Flags>]
```

Flags are optional. The defaults from table [1.2](#) are used if no flags are given. See an example from the package “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

```

[stat](#) (Page 25) at first checks for the file existing in the configuration directory. If it is, it will be included in the archive, if not, `mkfli41` will abort with an error message.

Hint: Also for `add_to_opt` `mkfli41` will first [check](#) (Page 11) if the file to be copied can be found in the configuration directory.

Filenames as well flags passed as string constants are subject of [variable substitution](#) (Page 20).

Control Flow

```

if (expr)
then
    statement
else
    statement
fi

```

A classic case distinction, as we know it. If the condition is true, the **then** part is executed, if the condition is wrong the **else** part.

If you want to run tests on array variables, you have to test every single variable. The **foreach** loop in two variants for this.

1. Iterate over array variables:

```

foreach <control variable> in <array variable>
do
    <instruction>
done

foreach <control variable> in <array variable-1> <array variable-2> ...
do
    <instruction>
done

```

This loop iterates over all of the specified array variables, each starting with the first to the last element, the number of elements in this array is taken from the N-variable

associated with this array. The control variable takes the values of the respective array variables. It should be noted that when processing optional array variables that are not present in the configuration, an empty element is generated. You may have to take this into account in the script, for example like this:

```
foreach i in template_var_opt_%
do
    if (i != "")
    then
        warning "%i is present (%i='$i')"
    else
        warning "%i is undefined (empty)"
    fi
done
```

As you also can see in the example, the *name* of the respective array variables can be determined with the `%<control variable>` construction.

The instruction in the loop may be one of the above control elements or functions (`if`, `foreach`, `provides`, `depends`, ...).

If you want to access exactly one element of an array, you can address it using the syntax `<Array>[<Index>]`. The index can be a normal variable, a numeric constant or again an indexed array.

2. Iteration over N-variables:

```
foreach <control variable> in <N-variable>
do
    <instruction>
done
```

This loop executes from 1 to the value that is given in the N-variable. You can use the control variable to index array variables. So if you want to iterate over not only one but more array variables at the same time all controlled by the *same* N-variable you take this variant of the loop and use the control variable for indexing multiple array variables. Example:

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

The resulting content of the `HOST_%_NAME`- and `HOST_%_IP4`-arrays for this example:

```
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
```

Expressions

Expressions link values and operators to a new value. Such a value can be an normal variable, an array element, or a constant (Number, string or version number). All string constants in expressions are subject to [variable substitution](#) (Page 20).

Operators allow just about everything you could want from a programming language. A test for the equality of two variables could look like this:

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

It should be noted that the comparison is done depending on the type that was defined for the variable in `check/<PACKAGE>.txt`. If one of the two variables is [numeric](#) (Page 20) the comparison is made numeric-based, meaning that the strings are converted to numbers and then compared. Otherwise, the comparison is done string-based; comparing `"05" == "5"` gives the result `"false"`, a comparison `"18" < "9"` `"true"` due to the lexicographical string order: the digit `"1"` precedes the digit `"9"` in the ASCII character set.

For the comparison of version numbers the construct `numeric(version)` is introduced, which generates the numeric value of a version number for comparison purposes. Here applies:

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

whereas `"major"` is the first component of the version number, `"minor"` the second and `"sub"` the third. If `"sub"` is missing the term in the addition above is omitted (in other words `"sub"` will be equalled to zero).

A complete list of all expressions can be found in table 1.3. `"val"` stands for any value of any type, `"number"` for a numeric value and `"string"` for a string.

Table 1.3: Logical Epressions

| Expression | true if |
|---|---|
| <code>id</code> | <code>id == "yes"</code> |
| <code>val == val</code> | values of identical type are equal |
| <code>val != val</code> | values of identical type are unequal |
| <code>val == number</code> | numeric value of <code>val == number</code> |
| <code>val != number</code> | numeric value of <code>val != number</code> |
| <code>val < number</code> | numeric value of <code>val < number</code> |
| <code>val > number</code> | numeric value of <code>val > number</code> |
| <code>val == version</code> | <code>numeric(val) == numeric(version)</code> |
| <code>val < version</code> | <code>numeric(val) < numeric(version)</code> |
| <code>val > version</code> | <code>numeric(val) > numeric(version)</code> |
| <code>val =~ string</code> | regular expression in string matches val |
| <code>(expr)</code> | Expression in brackets is true |
| <code>expr && expr</code> | both expressions are true |
| <code>expr expr</code> | at least one of both expressions is true |
| <code>copy_pending(id)</code> | see description |
| <code>samenet (string1, string2)</code> | string1 describes the same net as string2 |
| <code>subnet (string1, string2)</code> | string1 describes a subnet of string2 |

Match-Operator

With the match operator `=~` you can check whether a regular expression matches the value of a variable. Furthermore, one can also use the operator to extract subexpressions from a variable. After successfully applying a regular expression on a variable the array `MATCH_%` contains the parts found. May look like this:

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

Calling `mkfli4l` then would lead to this output:

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

When using `=~` you may take all existing regular expressions into account. If you i.e. want to check whether a PCMCIA Ethernet driver is selected without `OPT_PCMCIA` being set to “yes”, it might look like this:

```
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
```

As demonstrated in the example, it is important to *anchor* the regular expression with `^` and `$` if intending to apply the expression on the *complete* variable. Otherwise, the match-expression already returns “true” if only a *part* of the variable is covered by the regular expression, which is certainly not desired in this case.

Check if a File has been copied depending on the Value of a Variable: `copy_pending`

With the information gained during the checking process the function `copy_pending` tests if a file has been copied depending on the value of a variable or not. This can be used i.e. in order to test whether the driver specified by the user really exists and has been copied. `copy_pending` accepts the name to be tested in the form of a variable or a string. ⁵ In order to accomplish this `copy_pending` checks whether

⁵As described before the string is subject of variable substitution, i.e via a [foreach-loop](#) (Page 29) and a [%<Name>-substitution](#) (Page 20) all elements of an array may be examined.

- the variable is active (if it depends on an OPT it has to be set to “yes”),
- the variable was referenced in an `opt/<PACKAGE>.txt`-file and
- whether a file was copied dependant on the current value.

`copy_pending` will return “true” if it detects that during the last step *no* file was copied, the copy process hence still is “pending”.

A small example of the use of all these functions can be found 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
```

Alle elements of the array `NET_DRV_*` are detected for which no copy action has been done because there is no corresponding entry existing in `opt/base.txt`.

Comparison of Network Addresses: `samenet` und `subnet`

For testing routes from time to time a test is needed whether two networks are identical or if one is a subnet of the other. The two functions `samenet` and `subnet` are of help here.

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

returns “true” if both nets are identical and

```
subnet (net1, net2)
```

returns “true” if “net1” is a subnet of “net2”.

Expanding the Kernel Command Line

If an OPT must pass other boot parameters to the kernel, in former times the variable `KERNEL_BOOT_OPTION` had to be checked whether the required parameter was included, and if necessary, a warning or error message had to be displayed. With the internal variable `KERNEL_BOOT_OPTION_EXT` you may add a necessary but missing option directly in an ext-script. An Example taken from `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
```

This passes “acpi=off” to the kernel if no or “APM”-type power management is desired.

1.3.8 Support for Different Kernel Version Lines

Different kernel version lines often differ in some details:

- changed drivers are provided, some are deleted, others are added
- module names simply differ
- module dependencies are different
- modules are stored in different locations

These differences are mostly handled automatically by `mkfli41`. To describe the available modules you can, on one hand expand tests dependant on the version ([conditional regular expressions](#) (Page 16)), or, on the other hand `mkfli41` allows *version dependant* `opt/<PACKAGE>.txt`-files. These are then named `opt/<PACKAGE>_<Kernel-Version>.txt`, where the components of the kernel version are separated from each other by underscores. An example: the package “base” contains these files in its `opt`-directory:

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

the first file (`base.txt`) is *always* considered. Both other files are only considered if the kernel version is called “3.18(.*?)” resp. “3.19(.*?)”. As seen here, some parts of the version may be omitted in file names, if a group of kernels should be addressed. If `KERNEL_VERSION='3.18.9'` is given, the following files (if existing) are considered for the package `<PACKAGE>`:

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

1.3.9 Documentation

Documentation should be placed in the files

- `doc/<LANGUAGE>/opt/<PACKAGE>.txt`
- `doc/<LANGUAGE>/opt/<PACKAGE>.html`.

HTML-files may be splitted, meaning one for each OPT contained. Nevertheless a file `<PACKAGE>.html` has to be created linking to the other files. Changes should be documented in:

- `changes/<PACKAGE>.txt`

The entire text documentation may not contain any tabs and has to have a line feed no later than after 79 characters. This ensures that the documentation can also be read correctly with an editor without automatic line feed.

Also a documentation in L^AT_EX-format is possible, with HTML and PDF versions generated from it. The documentation of fli4l may serve as an example here. A documentation framework for required L^AT_EX-macros can be found in the package “template”. A brief description is to be found in the following subsections.

The fli4l documentation is currently available in the following languages: German (<LANGUAGE> = “deutsch”), English (<LANGUAGE> = “english”) and French (<LANGUAGE> = “french”). It is the package developer’s decision to document his package in any language. For the purposes of clarity it is recommended to create a documentation in German and/or English (ideally in both languages).

Prerequisites for Creating a L^AT_EX Documentation

To create a documentation from L^AT_EX-sources the following requirements apply:

- Linux/OS X-Environment: For ease of production, a makefile exists to automate all other calls (Cygwin should work too, but is not tested by the fli4l team)
- LaTeX2HTML for the HTML version
- of course L^AT_EX (Recommended: “TeX Live” for Linux/OS X and “MiKTeX” for Microsoft Windows) the “pdftex”program and these T_EX-packages:
 - current KOMA-Skript (at least version 2)
 - all packages necessary for pdftex
 - unpacked documentation package for fli4l, it provides the necessary makefiles and T_EX-styles

File Names

The documentation files are named according to the following scheme:

<PACKAGE>_main.tex: This file contains the main part of the documentation. <PACKAGE> stands for the name of the package to be described (in lowercase letters).

<PACKAGE>_appendix.tex: If further comments should be added to the package, they should be placed there.

The files should be stored in the directory fli4l/<PACKAGE>/doc/<SPRACHE>/tex/<PACKAGE>. For the package sshd this looks like here:

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

The Makefile is responsible for generating the documentation, the sshd.tex-file provides a framework for the actual documentation and the appendix, which is located in the other two files. See an example in the documentation of the package “template”.

L^AT_EX-Basics

L^AT_EX is, just like HTML, “Tag-based” , only that the tags are called “commands” and have this format: `\command` resp. `\begin{environment} ... \end{environment}`

By the help of commands you should rather emphasize the *importance* of the text less the *display*. It is therefore of advantage to use

```
\warning{Please_do_not...}
```

instead of

```
\emph{Please_do_not...}
```

Each command resp. each environment may take some more parameters noted like this: `\command{parameter1}{parameter2}{parameterN}`.

Some commands have optional parameters in square (instead of curly) brackets: `\command[optionalParameter]{parameter1} ...` Usually only one optional parameter is used, in rare cases there may be more.

Individual paragraphs in the document are separated by blank lines. Within these paragraphs L^AT_EX itself takes care of line breaks and hyphenation.

The following characters have special meaning in L^AT_EX and, if occurring in normal text, must be masked prefixed by a `\`: `# $ & _ % { }`. “~” and “^” have to be written as follows: `\verb?~? \verb?^?`

The main L^AT_EX-commands are explained in the documentation of the package “template”.

1.3.10 File Formats

All text files (both documentation and scripts, which later reside on the router) should be added to the package in DOS file format, with CR/LF instead of just LF at the end of a line. This ensures that Windows users can read the documentation even with “notepad” and that after changing a script under Windows everything still is executable on the router.

The scripts are converted to the required format during archive creation (see the description of the flags in table 1.2).

1.3.11 Developer Documentation

If a program from the package defines a new interface that other programs can use, please store the documentation for this interface in a separate documentation in `doc/dev/<PACKAGE>.txt`.

1.3.12 Client Programs

If a package also provides additional client programs, please store them in the directory `windows/` for Windows clients and in the directory `unix/` for *nix and Linux clients.

1.3.13 Source Code

Customized programs and source code may be enclosed in the directory `src/<PACKAGE>/`. If the programs should be built like the rest of the f4l programs, please have a look at the documentation of the “src”-package (Page ??) .

1.3.14 More Files

All files, which will be copied to the router have to be stored under `opt/`. Be under

- `opt/etc/boot.d/` and `opt/etc/rc.d/`: scripts, that should be executed on system start
- `opt/etc/rc0.d/`: scripts, that should be executed on system shutdown
- `opt/etc/ppp/`: scripts, that should be executed on dialin or hangup
- `opt/`: executable programs and other files according to their positions in the file system (for example the file `opt/bin/busybox` will later be situated in the directory `/bin` on the router)

Scripts in `opt/etc/boot.d/`, `opt/etc/rc.d/` and `opt/etc/rc0.d/` have the following naming scheme:

```
rc<number>.<name>
```

The number defines the order of execution, the name gives a hint on what program/package is processed by this script.

1.4 Creating Scripts for fli4l

The following is *not* a general introduction to shell scripts, everyone can read about this topic on the Internet. It is only about the fli4l-specific things. Further information is available in the various *nix/Linux help pages. The following links may be used as entry points to this topic:

- Introduction to shell scripts:
 - http://linuxcommand.org/writing_shell_scripts.php
- Help pages online:
 - <http://linux.die.net/>
 - <http://heapsort.de/man2web>
 - <http://man.he.net/>
 - http://www.linuxcommand.org/superman_pages.php

1.4.1 Structure

In the *nix world, it is necessary to begin a script with the name of the interpreter, hence the first line is:

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

To easily recognize later what a script does and who created it, this line should now be followed by a short header, like so:

```
#-----
# /etc/rc.d/rc500.dummy - start my cool dummy server
#
# Creation:      19.07.2001 Sheldon Cooper  <sheldon@nerd.net>
# Last Update:   11.11.2001 Howard Wolowitz <howard@nerd.net>
#-----
```

Now for the real stuff to start...

1.4.2 Handling of Configuration Variables

Packages are configured via the file `config/<PACKAGE>.txt`. The [active variables](#) (Page 11) contained there are transferred to the file `rc.cfg` during creation of the medium. This file is processed during router boot before any rc-script (scripts under `/etc/rc.d/`) gets started. The script then may access all configuration variables by reading the content of `$<variable name>`.

If the values of configuration variables are needed after booting they may be extracted from `/etc/rc.cfg` to which the configuration of the boot medium was written during the boot process. If for example the value of the variable `OPT_DNS` should be processed in a script this can be achieved as follows:

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

This works efficiently also with multiple variables (with calling the `grep` program only once):

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

1.4.3 Persistent Data Storage

Occasionally a package needs the possibility to store data persistent, surviving the reboot of the router. For this purpose, the function `map2persistent` exists and can be called from a script in `/etc/rc.d/`. It expects a variable that contains a path and a subdirectory. The idea is that the variable is either describing an actual path – then this path is used because the user explicitly has done so, or the string “auto” – then a subdirectory on a persistent medium is created corresponding to the second parameter. The function returns the result in the variable passed by name as the first parameter. An example will make this clear. `VBOX_SPOOLPATH` is a variable, that contains a path or the string “auto”. The call

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

results in the variable `VBOX_SPOOLPATH` either not being changed at all (if it contains a path), or being changed to `/var/lib/persistent/vbox/spool` (if it contains the string “auto”). `/var/lib/persistent` then points⁶ to a directory on a non-volatile and writable storage medium, `<SCRIPT>` is the name of the calling script in lowercase (this name is derived from

⁶ by the use of a so-called “bind”-mount

the first argument of the `begin_script-call` (Page 39)). If no suitable medium should exist (which may well be), `/var/lib/persistent` is a directory in the RAM disk.

Please note that the path returned by `map2persistent` is *not* created automatically – The caller has to do that by himself (ie. by calling `mkdir -p <path>`).

The file `/var/run/persistent.conf` allows for checking if persistent data storage is possible. Example:

```
. /var/run/persistent.conf
case $SAVETYPE in
persistent)
    echo "persistent data storage is possible!"
    ;;
transient)
    echo "persistent data storage is NOT possible!"
    ;;
esac
```

1.4.4 Debugging

For startup scripts it is often useful to run them in debug mode in a shell when you are in need to determine where they fail. For this purpose, insert the following at the beginning and at the end:

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

At the start and at the end of the script the specified text will now appear, preceded by “finished”.

If you want to debug the scripts, you must do two things:

1. You have to set `DEBUG_STARTUP` (Page ??) to “yes”.
2. You have to activate debugging for the OPT. This is usually done by the entry

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

in the config file.⁷ What happens during runtime is now displayed in detail on screen.

Further Variables helpful for Debugging

DEBUG_ENABLE_CORE This variable allows the creation of “Core-Dumps”. If a program crashes due to an error an image of the current state in the file system is stored which can be used to analyse the problem. The core dumps are stored under `/var/log/dumps/`.

DEBUG_IP Activating this variable will log all calls of the program `ip`.

DEBUG_IPUP Setting this variable to “yes” will log all executed instructions of the `ip-up`- and `ip-down`-scripts to syslog.

⁷Sometimes multiple start-scripts are used, which then have different names for their debug-variables. Have a quick look at the scripts for clarification.

LOG_BOOT_SEQ Setting this variable to “yes” will cause `bootlogd` to log all console output during boot to the file `/var/tmp/boot.log`. This variable has “yes” as a default value.

DEBUG_KEEP_BOOTLOGD Normally `bootlogd` is terminated at the end of the boot process. Activating this variable prevents this and thus allows for logging console output during the whole runtime.

DEBUG_MDEV Setting this variable generates a logfile for the `mdev-daemon`, which is responsible for creating device nodes under `/dev`.

1.4.5 Hints

- It is *always* better to use curly brackets “{...}” instead of normal ones“(...)”. However, care must be taken to ensure that after the opening bracket a space or a new line follows before the next command and before the closing brackets a Semicolon or a new line. For example:

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

is equal to:

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

- A script may be stopped by an “exit” at any time. This is deadly for start scripts (`opt/etc/boot.d/...`, `opt/etc/rc.d/...`), stop-scripts (`opt/etc/rc0.d/...`) and ip-up/ip-down-scripts (`opt/etc/ppp/...`) because the following scripts will not be run as well. If in doubt, keep your fingers away.
- KISS – Keep it small and simple. You want to use Perl for scripting? The scripting abilities of `fi4l` are not enough for you? Rethink your attitude! Is your OPT really necessary? `fi4l` after all is “only” a router and a router should not really offer server services.
- The error message “: not found” usually means the script is still in DOS format. Another source of errors: the script is not executable. In both cases the `opt/<PACKAGE>.txt` file should be checked whether it contains the correct options for “mode”, “gid”, “uid” and Flags. If the script is created during boot, execute “`chmod +x <script name>`”.
- Use the path `/tmp` for temporary files. However, it is important to keep in mind that there is little space there because it is situated in the rootfs-RAM disc! If more space is needed, you have to create and mount your own ramdisk. Detailed informations on this topic can be found in the section “RAM-Disks” of this documentation.
- In order to create temporary files with unique names you should always append the current process-ID stored in the shell variable “\$” to the file name. `/tmp/<OPT-Name>.$$` hence is a perfect file name, `/tmp/<OPT-Name>` rather not (`<OPT-Name>` of course has to be replaced by the according OPT-Name).

1.5 Using The Packet Filter

1.5.1 Adding Own Chains And Rules

A set of routines is provided to manipulate the packet filter to add or delete so-called “chains” and “rules”. A chain is a named list of ordered rules. There is a set of predefined chains (PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING), using this set of routines more chains can be created as needed.

add_chain/add_nat_chain <chain>: Adds a chain to the “filter-” or “nat-” table.

flush_chain/flush_nat_chain <chain>: Deletes all rules from a chain of the “filter-” or “nat-” table.

del_chain/del_nat_chain <chain>: Deletes a chain from the “filter-” or “nat-” table. Chains must be empty prior to deleting and all references to them have to be deleted as well before. Such a reference i.e. can be a JUMP-action with the chain defined as its target.

add_rule/ins_rule/del_rule: Adds rules to the end (**add_rule**) resp. at any place of a chain (**ins_rule**) or deletes rules from a chain (**del_rule**). Use the syntax like here:

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

where the parameters have the following meaning:

table The table in which the chain is

chain The chain, in which the rule is to be inserted

rule The rule which is to be inserted, the format corresponds to that used in the configuration file

position The position at which the rule will be added (only in **ins_rule**)

comment A comment that appears with the rule when somebody looks at the packet filter.

1.5.2 Integrating Into Existing Rules

fi4l configures the packet filter with a certain default rule set. If you want to add your own rules, you will usually want to insert them after the default rule set. You will also need to know what the action is desired by the user when dropping a packet. This information can be obtained for FORWARD- and INPUT chains by calling two functions, **get_defaults** and **get_count**. After calling

```
get_defaults <chain>
```

the following results are obtained:

drop: This variable contains the chain to which is branched when a packet is discarded.

reject: This variable contains the chain to which is branched when a packet is rejected.

After calling

```
get_count <chain>
```

the variable `res` contains the number of rules in the chain `<chain>`. This position is of importance because you can *not* simply use `add_rule` to add a rule at the end of the predefined “filter”-chains `INPUT`, `FORWARD` and `OUTPUT`. This is because these chains are completed with a default rule valid for all remaining packets depending on the content of the `PF_<chain>_POLICY`-variable. Adding a rule *after* this last rule hence has no effect. The function `get_count` instead allows to detect the position right *in front of* this last rule and to pass this position to the `ins_rule`-function as a parameter `<position>` in order to add the rule in the place at the end of the appropriate chain, but right in front of this last default rule targeting all remaining packets.

An example from the script `opt/etc/rc.d/rc390.dns_dhcp` from the package “`dns_dhcp`” shall make this clear:

```
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 "dhcprelay access"

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

    end_script
  ;;
esac
```

Here you can see in the middle of the loop a call to `get_count` followed by a call to the `ins_rule` function and, among other things, the `res` variable is passed as `position` parameter.

1.5.3 Extending The Packet Filter Tests

`fi4l` uses the syntax `match:params` in packet filter rules to add additional conditions for packet matching (see `mac:`, `limit:`, `length:`, `prot:`, ...). If you want to add tests you have to do this as follows:

1. Define a suitable name. The first character of this name has to be lower case a-z. The rest of the name can consist of any character or digit.

If the packet filter test is used within IPv6 rules it is to make sure that the name is not a valid IPv6 address component!

2. Creating a file `opt/etc/rc.d/fwrules-<name>.ext`. The content of this file is something like this:

```
# 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"
}
```

The packet filter test does not have to be implemented for both IPv4 and IPv6 (though this would be preferred if reasonable for both layer 3 protocols).

3. Testing the extension:

```
$ 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. Adding the extension and all other needed files (`iptables` components) to the archive using the known mechanisms.
5. Allowing the extension in the configuration by extending of `FW_GENERIC_MATCH` and/or `FW_GENERIC_MATCH6` in an `exp`-file, for example:

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

1.6 CGI-Creation for Package *httpd*

1.6.1 General information about the web server

The web server used in fli4l is `mini_httpd` by ACME Labs. The sources can be found at http://www.acme.com/software/mini_httpd/. However, a few changes in the current version were made for fli4l. The modifications are located in the `src` package in the directory `src/fbr/buildroot/package/mini_httpd`.

1.6.2 Script Names

The script names should be self-explanatory in order to be easy to distinguish from other scripts and even similar names have to be avoided to differ from other OPTs.

To make scripts executable and convert DOS line breaks to UNIX ones a corresponding entry has to be created in `opt/<PACKAGE>.txt`, see Table 1.2 (Page 9).

1.6.3 Menu Entries

To create an entry in the menu you have to enter it in the file `/etc/httpd/menu`. This mechanism enables OPTs to change the menu during runtime. This should only be done using the script `/etc/httpd/menu` because this will check for valid file formatting. New menu items are inserted as follows:

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

Thus, an entry with the name `<name>` is inserted to the `[section]`. If `[section]` is omitted, it will be inserted in the section “OPT-Packages” as default. `<link>` specifies the target of the new link. `<priority>` specifies the priority of a menu item in its section. If not set, the default priority used is 500. The priority should be a three digit number. The lower the priority, the higher the link is placed in the section. If an entry should be placed as far down as possible the priority to choose is e.g. 900. Entries with the same priority are sorted by the target of the link. In `[realm]` the range is specified for which a logged-in user must have `view` rights so the item is displayed for him. If `[realm]` is not specified, the menu item is always displayed. For this, see also the section “User access rights” (Page 49).

Example:

```
httpd-menu.sh add "newfile.cgi" "Click here" "Tools" "tools"
```

This example creates a link named “Click here” with the target “newfile.cgi” in the section “Tools” which will be created if not present.

The script may also delete entries from a menu:

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

By executing this the entry containing the link `<link>` will be deleted.

Important: *If several entries have the same link target file they will all be removed from the menu.*

Since sections can have priorities they can also be created manually. If a section was created automatically when adding a menu entry it defaults to priority 500. The syntax for creating sections is as follows:

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

<priority> should only be a three digit number.

In order to create meaningful priorities it is worthwhile to have a look at the file `/etc/httpd/menu` of `fl4l` during runtime, priorities are placed in the second column.

A short description of the file format of the file `menu` follows for completeness. Those satisfied with the function of `httpd-menu.sh` may skip this section. The file `/etc/httpd/menu` is divided into four columns. The first column is a letter identifying the line as a heading or a menu entry. The second column is the sort priority. The third column contains the target of the link for entries and for headlines a hyphen, as this field has no meaning for headings. The rest of the line is the text that will appear in the menu.

Headings use the letter “t”, a new menu section will be started then. Normal menu items use the letters “e”. An example:

```
t 300 - My beautiful OPT
e 200 myopt1.cgi Do something beautiful
e 500 myopt1.cgi?more=yes Do something even more beautiful
```

When editing this file you have to be aware that the script `httpd-menu.sh` always stores the file sorted. The individual sections are sorted and the entries in this section are sorted too. The sorting algorithm can be stolen from `httpd-menu.sh`, however, it would be better to expand the script itself with possible new functions, so that all menu-editing takes place at a central location.

1.6.4 Construction of a CGI script

The headers

All web server scripts are simple shell scripts (interpreter as e.g. Perl, PHP, etc. are much too big in filesize for `fl4l`). You should start with the mandatory script header (reference to the interpreter, name, what does the script, author, license).

Helper Script `cgi-helper`

After the header you should include the helper script `cgi-helper` with the following call:

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

A space between the dot and the slash is important!

This script provides several helper functions that should greatly simplify the creation of CGIs for `fl4l`. With the integration some standard tasks are performed, such as the parsing of variables that were passed via forms or via the URL or loading of language and CSS files.

Here is a small function overview:

Contents of a CGI script

To ensure consistency in design and especially the compatibility with future versions of `fl4l` it is highly recommended to use the functions of the `cgi-helper` script even if theoretically everything in a CGI could be generated from scratch.

A simple CGI script might look like this:

Table 1.4: Functions of the `cgi-helper` script

| Name | Function |
|-------------------------------|--|
| <code>check_rights</code> | Check for user access rights |
| <code>http_header</code> | Creation of a standard HTTP header or a special header, e.g. for file download |
| <code>show_html_header</code> | Creation of a complete page header (inc. HTTP header, headline and menu) |
| <code>show_html_footer</code> | Creation of a footer for the HTML page |
| <code>show_tab_header</code> | Creation of a content window with tabs |
| <code>show_tab_footer</code> | Creation of a footer for the content window |
| <code>show_error</code> | Creation of a box for error messages (background color: red) |
| <code>show_warn</code> | Creation of a box for warning messages (background color: yellow) |
| <code>show_info</code> | Creation of a box for information/ success messages (background color: green) |

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

show_html_header "My first CGI"
echo '    <h2>Welcome</h2>'
echo '    <h3>This is a CGI script example</h3>'
show_html_footer
```

The Function `show_html_header`

The `show_html_header` function expects a string as a parameter. This string represents the title of the generated page. It automatically generates the menu and includes associated CSS and language files as long as they can be found in the directories `/srv/www/css` resp. `/srv/www/lang` and have the same name (but of course a different extension) as the script. An example:

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

Both the use of language files and CSS files is optional. The files `css/main.css` and `lang/main.<lang>` (where `<lang>` refers to the chosen language) are always included.

Additional parameters can be passed to the function `show_html_header`. A call with all possible parameters might look like this:

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

Any additional parameters must, as shown in the example, be enclosed with quotation marks and separated by a semicolon. The syntax will *not* be checked! So it is necessary to pay close attention to the exact parameter syntax.

Here is a brief overview of the function of the parameters:

- `refresh=time`: Time in seconds in which the page should be reloaded by the browser.

- `url=url`: The URL which is reloaded on a refresh.
- `cssfile=cssfile`: Name of a CSS file if it differs from the name of the CGI.
- `showmenu=no`: By using this the display of the menu and the header can be suppressed.

Other Content Guidelines:

- Don't write novels, use short descriptions :-)
- Use clean HTML (SelfHTML⁸ is a good starting point)
- Omit the bells and whistles (JavaScript is OK, if it does not interfere and support the user, everything also has to work without JavaScript)

The Function `show_html_footer`

The function `show_html_footer` closes the block from the CGI script which was opened by the function `show_html_header`.

The Function `show_tab_header`

For good looking content of your generated webpage generated by the CGI you may use the `cgi-helper` function `show_tab_header`. It creates clickable "Tabs" in order to present your page divided into multiple logically separated areas.

Parameters are always passed in pairs to the `show_tab_header` function. The first value reflects the title of a tab, the second reflects the link. If the string "no" is passed as a link only the title will be created and the tab is not clickable (and blue).

In the following example a "window" with the title "A great window" is generated. In the window is "foo bar":

```
show_tab_header "A great window" "no"
echo "foo"
echo "bar"
show_tab_footer
```

In this example, two clickable selection tabs are generated that pass the variable `action` to the script, each with a different value.

```
show_tab_header "1st selection tab" "$myname?action=dothis" \
                "2nd selection tab" "$myname?action=dothat"
echo "foo"
echo "bar"
show_tab_footer
```

Now the script can change the content of the variable `FORM_action` (see variable evaluation below) and provide different content depending on the selection. For the clicked tab to appear selected and not clickable anymore, a "no" would have to be passed to the function instead of the link. But there is an easier way, if you hold to the convention in the following example:

⁸see <http://de.selfhtml.org/>

```
_opt_dothis="1st selection tab"
_opt_dothat="2nd selection tab"
show_tab_header "$_opt_dothis" "$myname?action=opt_dothis" \
                "$_opt_dothat" "$myname?action=opt_dothat"
case $FORM_action in
    _opt_dothis) echo "foo" ;;
    _opt_dothat) echo "bar" ;;
esac
show_tab_footer
```

Hence, if a variable whose name equals the content of the variable `action` with a leading underscore (`_`) is passed as the title then the tab will be displayed selected.

The Function `show_tab_footer`

The function `show_tab_footer` closes the block in the CGI script that was opened by the function `show_tab_header`.

Multi-Language Capabilities

The helper script `cgi-helper` furthermore contains functions to create multi-lingual CGI scripts. You only have to use variables with a leading underscore (`_`) for all text output. This variables have to be defined in the respective language files.

Example:

Let `lang/opt.de` contain:

```
_opt_dothis="Eine Ausgabe"
```

Let `lang/opt.en` contain:

```
_opt_dothis="An Output"
```

Let `admin/opt.cgi` contain:

```
...
echo $_opt_dothis
...
```

Form Evaluation

To process forms you have to know a few more things. Regardless of using the form's `GET` or `POST` methods, after including the `cgi-helper` script (which in turn calls the utility `proccgi`) the parameters can be accessed by variables named `FORM_<Parameter>`. If i.e. the form field had the name "input" the CGI script can access its content in the shell variable `$FORM_input`.

Further informations on the program `proccgi` can be found under <http://www.fpx.de/fp/Software/ProcCGI.html>.

User access rights: The Function `check_rights`

At the beginning of a CGI scripts the `check_rights` function has to be called in order to check whether a user has sufficient rights to use the script. Do this like here:

```
check_rights <Section> <Action>
```

The CGI script will only be executed if the user, who is logged in at the moment

- has all rights (`HTTPD_RIGHTS_x='all'`), or
- has all rights for the current area (`HTTPD_RIGHTS_x='<Bereich>:all'`), or
- has the right to execute the function in the current area (`HTTPD_RIGHTS_x='<Bereich>:<Aktion>'`).

The Function `show_error`

This function displays an error message in a red box. It expects two parameters: a title and a message. Example:

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

The Function `show_warn`

This function displays a warning message in a yellow box. It expects two parameters: a title and a message. Example:

```
show_info "Warning" "No connection at the moment!"
```

The Function `show_info`

This function displays an information or success message in a green box. It expects two parameters: a title and a message. Example:

```
show_info "Info" "Action successfully executed!"
```

The Helper Script `cgi-helper-ip4`

Right after `cgi-helper` the helper script `cgi-helper-ip4` may be included by writing the following line:

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

A space between the dot and the slash is important!

The script provides helper functions for checking IPv4 addresses.

The Function `ip4_isvalidaddr`

This function checks if a valid IPv4 address was passed. Example:

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

The Function `ipv4_normalize`

This function removes leading zeros from the passed IPv4 address. Example:

```
ipv4_normalize ${FORM_inputip}
IP=$res
if [ -n "$IP" ]
then
    ...
fi
```

The Function `ipv4_isindhcprange`

This function checks whether the passed IPv4 address is ranged between the passed start and end addresses. Example:

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

1.6.5 Miscellaneous

This and that (yes, also important!):

- `mini_httpd` does not protect subdirectories with a password. Each directory must contain a `.htaccess` file or a link to another `.htaccess` file.
- KISS - Keep it simple, stupid!
- This information may change at any time without prior notice!

1.6.6 Debugging

To ease debugging of a CGI script you may activate the debugging mode by sourcing the `cgi-helper` script. Set the variable `set_debug` to “yes” in order to do so. This will create a file `debug.log` which may be loaded down with the URL `http://<fli4l-Host>/admin/debug.log`. It contains all calls of the CGI script. The variable `set_debug` is not a global one, it has to be set anew for each CGI in question. Example:

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

Furthermore, `cURL`⁹ is ideal for troubleshooting, especially if the HTTP headers are not assembled correctly or the browser displays only blank pages. Also, the caching behavior of modern Web browser is obstructive when troubleshooting.

Example: Get a dump of the HTTP-Header with (“`dump`”, `-D`) and the normal output of the CGI `admin/my.cgi`. The “`user`” (`-u`) name here shall be “`admin`”.

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

⁹see <http://de.wikipedia.org/wiki/CURL>

1.7 Boot, Reboot, Dialin And Hangup Under fli4l

1.7.1 Boot Concept

FLI4L 2.0 should offer a clean install on a hard disk or a CompactFlash (TM) media, but also an installation on a Zip medium or the creation of a bootable CD-ROM should be possible. In addition, the hard drive version should not be fundamentally different from the one on an installation disk¹⁰.

These requirements have been implemented by making it possible to move the files of the `opt.img` archive from the previous RAM disk to another medium, be it a partition on a hard disk or a CF medium. This second volume is mounted to `/opt` and only symbolic links are created from there to the rootfs. The resulting layout in the root file system then corresponds to the one unpacked in the `opt` directory of the fli4l distribution with one exception – the `files` prefix is not applicable. The file `opt/etc/rc` is then found directly under `/etc/rc`, `opt/bin/busybox` under `/bin/busybox`. It can be ignored that these files may be only links to a directory mounted read only as long as you do not want to modify them. If you want to do this, you have to make the files writable before by using `mk_writable` (see below).

1.7.2 Start And Stop Scripts

Scripts intended to be executed on system boot are located in the directories `opt/etc/boot.d/` and `opt/etc/rc.d/` and will also get executed in this sequence. Furthermore, scripts executed on shutdown are to be found in `opt/etc/rc0.d/`.

Important: *These script must not contain an “exit”, because no separate process is created for their execution. This command would lead to a premature ending of the boot process!*

Start Scripts in `opt/etc/boot.d/`

Scripts located in this directory are executed at first. They mount the boot volume, parse the config file `rc.cfg` located on the boot medium and unpack the `opt` archive. Depending on the boot type (Page ??) these scripts are more or less complex and do the following things:

- Loading of hardware drivers (optional)
- Mount the `boot` volume (optional)
- Read the config file `rc.cfg` off the boot volume and write it to the file `/etc/rc.cfg`
- Mount the `opt` volume (optional)
- Unpack the `opt` archive (optional)

To make the scripts aware of the fli4l configuration, the configuration file `/etc/rc.cfg` is also integrated in the Rootfs archive. The configuration variables in this file are parsed by the start scripts in `opt/etc/boot.d/`. After mounting the `boot` volume `/etc/rc.cfg` is replaced by the configuration file there, so that the the current configuration of the `boot` volume is available for startup scripts in `opt/etc/rc.d/` (see below). ¹¹

¹⁰ Originally fli4l could be operated from a single floppy disk. This is no longer supported since it became too big in file size.

¹¹ Normally, these two files are identical. Discrepancies are possible only if the configuration file on the `boot`

Start Scripts in `opt/etc/rc.d/`

Commands that are executed at every start of the router can be stored in the directory `opt/etc/rc.d/`. The following conventions apply:

1. Name conventions:

```
rc<three-digit number>.<Name of the OPT>
```

The scripts are started in ascending order of the numbers. If multiple scripts have the same number assigned, they will be sorted alphabetically at that point. In case that the start of a package is dependant on another one, this is determined by the number.

Here's a general outline which numbers should be used for which tasks:

| Number | Task |
|---------|--|
| 000-099 | Base system (hardware, time zone, file system) |
| 100-199 | Kernel modules (drivers) |
| 200-299 | External connections (PPPoE, ISDN4Linux, PPTP) |
| 300-399 | Network (Routing, Interfaces, Packet filter) |
| 400-499 | Server (DHCP, HTTPD, Proxy, a.s.o.) |
| 500-900 | Any |
| 900-997 | Anything causing a dialin |
| 998-999 | reserved (please do not use!) |

2. These scripts *must* contain all functions changing the RootFS, ie. creating of a directory `/var/log/lpd`.
3. These scripts shall *not* contain writing to files that could be part of the `opt` archive, because these files could be located on a volume mounted in read-only mode. If you have to modify such a file, you have to make it writeable before by using the function `mk_writable` (see below). This will create a writable copy of the file in the RootFS if needed. If the file is already writable the call of `mk_writable` will have no effect.

Important: *`mk_writable` has to be applied directly to files in RootFS, not indirectly via the `opt` directory. If, for example, you want to modify `/usr/local/bin/foo`, the function `mk_writable` has to be called with the argument `/usr/local/bin/foo`.*

4. Before executing the actual commands these scripts have to check for the associated OPT really being active. This is usually done by a simple if-case:

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

volume was edited manually, for example to modify the configuration later on without the need to rebuild the `fl4l` archives.

5. For easier debugging the scripts should be enclosed in `begin_script` and `end_script`:

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

Debugging of start-scripts may be activated simply via `FOO_DO_DEBUG='yes'`.

6. All configuration variables are available to the scripts in direct. Explanations how to access configuration variables in scripts can be found in the section [“Working with configuration variables”](#) (Page 38).
7. The path `/opt` may not be used for storing OPT data. If in need of additional file space you should enable the user to define a suitable location by using a configuration variable. Depending on the type of data to be stored (persistent or transient data) different default assignments should be used. A path under `/var/run/` makes sense for transient data, while for persistent data it is advised to use the function [map2persistent](#) (Page 38) combined with a suitable configuration variable.

Stop Scripts in `opt/etc/rc0.d/`

Each machine must be shut down or restarted from time to time. It is perfectly possible that you have to perform operations before the computer is shut down or restarted. To shut down and restart the commands “halt” or “reboot” are used. These commands are also invoked when the corresponding buttons in IMONC or the Web GUI are clicked.

All stop scripts can be found in the `opt/etc/rc0.d/`. The file names have to be created using the same rules as for the scripts. They are as well executed in *ascending* order of numbers.

1.7.3 Helper Functions

`/etc/boot.d/base-helper` provides a number of different functions that can be used in Start- and other scripts. They contain support for debugging, loading of kernel modules, or message output. The functions are listed and explained in short below

Script Control

begin_script <Symbol> <Message>: Output of a message and activation of script debugging by calling `set -x`, if <Symbol>_DO_DEBUG is set to “yes”.

end_script: Output of an end-message and deactivation of debugging if it was activated with `begin_script`. For each `begin_script` call a corresponding `end_script` call has to exist (and vice versa).

Loading Of Kernel Modules

do_modprobe [-q] <Modul> <Parameter>*: Loads a kernel module including its parameters (if needed) while resolving its module dependencies. The parameter “-q” prevents error messages to be written to the console and to the boot log in case of failure. The function returns 0 for success and another value in case of error. This enables you to create code for handling failures while loading kernel modules:

```
if do_modprobe -q acpi-cpufreq
then
    # no CPU frequency scaling via ACPI
    log_error "CPU frequency scaling via ACPI not available!"
    # [...]
else
    log_info "CPU frequency scaling via ACPI activated."
    # [...]
fi
```

do_modprobe_if_exists [-q] <Module path> <Module> <Parameter>*:

Checks if the module `/lib/modules/<Kernel-Version>/<Module path>/<Module>` exists and, if so, invokes `do_modprobe`.

Important: *The module has to exist exactly by this name, no aliases may be used. When using an alias `do_modprobe` will be called immediately.*

Messages And Error Handling

log_info <Message>: Logs a message to the console and to `/bootmsg.txt`. If no message is passed as a parameter `log_info` reads the default input. The function always returns 0.

log_warn <Message>: Logs a warning message to the console and to `/bootmsg.txt`, using the string `WARN:` as a prefix. If no message is passed as a parameter `log_warn` reads the default input. The function always returns 0.

log_error <Message>: Logs an error message to the console and to `/bootmsg.txt`, using the string `ERR:` as a prefix. If no message is passed as a parameter `log_warn` reads the default input. The function always returns a non-zero value.

set_error <Message>: Output of an error message and setting of an internal error variable which can be checked later via `is_error`.

is_error: Clears the internal error variable and returns true if it was set before via `set_error`.

Network Functions

translate_ip_net <Value> <Variable name> [<Result variable>]:

Replaces symbolic references in parameters. At the moment the following translations are supported:

.*., **none**, **default**, **pppoe** will not be translated

any will be replaced by 0.0.0.0/0

dynamic will be replaced by the IP address of the router which represents the Internet connection

IP_NET_x will be replaced by the network found in the configuration

IP_NET_x_IPADDR will be replaced by the IP address found in the configuration

IP_ROUTE_x will be replaced by the routed network found in the configuration

@<Hostname> will be replaced by the Hosts IP address specified in the configuration

The result of the translation is stored in the variable whose name is passed in the third parameter, if this parameter is missing, the result is stored in the variable **res**. The variable name that is passed in the second parameter is used only for error messages if the translation fails, to enable the caller to pass the source of the value to be translated. In case of failure a message like

```
Unable to translate value '<Value>' contained in <Variable name>.
```

will be printed.

The return value is 0 in case of success, and unequal to zero in case of errors.

Miscellaneous

mk_writable <File>: Ensures that the given file is writable. If the file is located on a volume mounted in read-only mode and is only linked to the file system via a symbolic link, a local copy will be created which is then written to.

list_unique <List>: Removes duplicates from a list passed. The result is written to standard output. **list**.

1.7.4 mdev-Rules

OPTs may establish additional mdev-rules that trigger defined actions in appearance or disappearance of certain devices. **OPT_AUTOMOUNT** from package **hd**, for example, uses such a rule to mount storage devices automatically. If you want to integrate an additional mdev rule, you have to embed a script of the form

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

into the RootFS whereas the number, similar to the start- and stop scripts, has to consist of three digits while the name can be chosen arbitrarily. In the script all output to the standard output is integrated in the resulting **/etc/mdev.conf**. An example from the aforementioned **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
```

Reference the rule's syntax from the file header of `/etc/mdev.conf` and from the mdev documentation at <http://git.busybox.net/busybox/plain/docs/mdev.txt>. If a rule calls a script (such as `/lib/mdev/automount` in the example above) it has access to all variables of the triggering kernel-“uevent”, in particular:

- **ACTION** (typically `add` or `remove`, less common `change`)
- **DEVPATH** (sysfs path to the affected component)
- **SUBSYSTEM** (the affected kernel subsystem, see below)
- **DEVNAME** (the affected device file under `/dev`; missing if no devices have to be created or deleted, but for example, modules)
- **MDEV** (is set by mdev to the name of the finally created device)

Example for kernel subsystems:

block Block devices (storage media) like `sda` (first harddisk), `sr0` (first CD drive) or `ram1` (second RAM disk)

input Input devices for keyboard, mouse a.s.o. like `input/event0`; which device file is linked to which device is not set and has to be determined via sysfs

mem Devices to access the memory and hardware ports as `mem` and `ports`; this encloses also pseudo devices like `zero` (continuously delivers the ASCII character with value zero) and `null` (returns nothing, swallows everything) among them

sound various devices for sound output, no naming convention

tty devices for accessing physical and virtual consoles like `tty1` (first virtual console) or `ttyS0` (first serial console)

An example for the first two serial ports:

```
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
```

An example of a connected MF II keyboard:

```
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 ffffffff 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,ram4,10,
1,2,sfw
mdev[41]: SEQNUM=604
```

An example of a loaded USB kernel module (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
```

An example of a connected hard disk:

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

This is an ATA/IDE hard drive (ata1) that should be accessed via the device name **sda**.

An example of a remote block device (assigning an image file to a fli4l VM was dissolved via **virsh detach-device**):

```
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]: DEVTYPEDISK=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:d000000002v00001AF4
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

```

As you can see, at such a removal various kernel subsystems are involved (here `block`, `bdi`, `virtio` and `pci`).

1.7.5 ttyI Devices

For `ttyI` devices (`/dev/ttyI0 ... /dev/ttyI15`) used by the “modem emulation” of the ISDN card a counter exists to avoid conflicts between multiple packages using these devices. For this purpose the file `/var/run/next_ttyI` is created on router start which can be queried and incremented by the OPTs. The following example script can query this value, increment it by one and export it again for the next OPT.

```

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
    log_error "No ttyI device for <Name of your OPT> available!" # ttyI device available? no
    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 Dialin And Hangup Scripts

General

After dialin resp. hangup of a dial-up connection the scripts placed in `/etc/ppp/` are executed. OPTs may store actions here that have to be executed after connecting resp. hanging up of a connection. The name scheme for the files is as follows:

```

ip-up<three-digit number>.<OPT-Name>
ip-down<three-digit number>.<OPT-Name>

```

`ip-up` scripts will be excuted after *establishing* and `ip-down` scripts after *hangig up* of the connection.

Important: *In `ip-down` scripts no actions may be taken that lead to another dialin because this would create a permanent-online condition not desired for users without a flatrate.*

Important: *Since no separate process is created for these scripts, they may not invoke “exit” as well!*

Hint: If a script wants to check for `ip-up` scripts being executed the variable `ip_up_events` may be sourced from `rc400` and up. If it is set to “yes” dialup-connections exist and `ip-up` scripts will be executed. No dialup-connections are configured if it is set to “no” and `ip-up` scripts will not get executed. There is an exeception to this rule: If an Ethernet router is configured without dialup-connections but a default-Route (0.0.0.0/0) exists, `ip-up` scripts will get executed only once at the end of the boot process. (And as well the `ip-down` scripts on roter shutdown.)

Variables

Due to the special call concept of the `ip-up` and `ip-down` scripts the following variables apply:

| | |
|-------------------------------|--|
| <code>real_interface</code> | the real interface, ie. <code>ppp0</code> , <code>ippp0</code> , ... |
| <code>interface</code> | the IMOND interface, ie. <code>pppoe</code> , <code>ippp0</code> , ... |
| <code>tty</code> | terminal connected, may be empty! |
| <code>speed</code> | connection speed, for ISDN ie. 64000 |
| <code>local</code> | own IP address |
| <code>remote</code> | IP address of the Point-To-Point partner |
| <code>is_default_route</code> | specifies if the current <code>ip-up/ip-down</code> is for the interface of the default route (may be “yes” or “no”) |

Default Route

As of version 2.1.0 `ip-up/ip-down` scripts are executed for all connections, not only for the interface of the default route. To emulate the old behaviour you have to include the following in `ip-up` and `ip-down` scripts:

```
# is a default-route-interface going up?
if [ "$is_default_route" = "yes" ]
then
    # actions to be taken
fi
```

Of course, the new behaviour can also be used for specific actions.

1.8 Package “template”

To illustrate some of the things described before the fli4l distribution provides the package “template”. This explains by small examples how:

- a configuration files has to look like (`config/template.txt`)
- a check files is designed (`check/template.txt`)
- the extended checking mechanisms are used (`check/template.ext`)
- configuration variables are stored for later use (`opt/etc/rc.d/rc999.template`)
- stored configuration variables are processed (`opt/usr/bin/template_show_config`)

1.9 Structure of the Boot Medium

As of version 1.5 the program `syslinux` is used for booting. Its advantage is that a DOS-compatible file system is available on the boot medium.

The boot medium contains the following files:

| | |
|---------------------------|--|
| <code>ldlinux.sys</code> | the “boot loader” <code>syslinux</code> |
| <code>syslinux.cfg</code> | config file for <code>syslinux</code> |
| <code>kernel</code> | Linux kernel |
| <code>rootfs.img</code> | RootFS: programs needed for booting |
| <code>opt.img</code> | Optional files: drivers and packages |
| <code>rc.cfg</code> | config file containing the variables from all files in fli4l’s configuration directory |
| <code>boot.msg</code> | Text for the <code>syslinux</code> boot menu |
| <code>boot_s.msg</code> | Text for the <code>syslinux</code> boot menu |
| <code>boot_z.msg</code> | Text for the <code>syslinux</code> boot menu |
| <code>hd.cfg</code> | config file to assign partitions |

The script `mkfli4l.sh` (resp. `mkfli4l.bat`) at first generates the files `opt.img`, `syslinux.cfg` and `rc.cfg` as well as `rootfs.img`. The files needed are determined by the program `mkfli4l` (in the `unix-` resp. `windows-`directory). The kernel and other packages are included in those archives. The file `rc.cfg` can be found as well in the `Opt`-archive as on the boot medium.¹²

¹²The one in the `Opt`-archive is needed during early boot, because no boot volume is mounted at that time.

Subsequently, the files `kernel`, `rootfs.img`, `opt.img` and `rc.cfg` together with the `syslinux`-files are copied to the disk.

During boot `fli4l` uses the script `/etc/rc` to evaluate the file `rc.cfg` and integrate the compressed `opt.img`-archive into the RootFS-RAM-Disk (depending on the installation type the files are extracted directly into the `rootfs` ramdisk or integrated via symbolic links). Then the scripts in `/etc/rc.d/` are run in alphanumeric order and thus the drivers are loaded and all services get started.

1.10 Configuration Files

Here a short list of the files generated by `fli4l` “on-the-fly” at boot time.

1. Provider configuration
 - `etc/ppp/pap-secrets`
 - `etc/ppp/chap-secrets`
2. DNS configuration
 - `etc/resolv.conf`
 - `etc/dnsmasq.conf`
 - `etc/dnsmasq_dhcp.conf`
 - `etc/resolv.dnsmasq`
3. Hosts-File
 - `etc/hosts`
4. `imond`-configuration
 - `etc/imond.conf`

1.10.1 Provider Configuration

For the providers chosen User-ID and password are adapted in `etc/ppp/pap-secrets`.

Example for Provider Planet-Interkom:

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

In this example “anonymer” is the USER-ID. As a remote server in principle anybody is allowed (hence “*”). “surfer” is the password for the Provider Planet-Interkom.

1.10.2 DNS Configuration

You can use `fli4l` as a DNS server. Why this is meaningful (and for Windows PCs in the LAN even mandatory) is explained in the documentation of the “base” package.

The resolver file `etc/resolv.conf` contains the domain name and the name server to use. It has the following contents (where “domain.de” only is a placeholder for the value of the configuration variable `DOMAIN_NAME`):

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

The DNS server `dnsmasq` is configured by the file `etc/dnsmasq.conf`. It is automatically generated during boot by processing the scripts `rc040.dns-local` and `rc370.dnsmasq` and might look like this:

```
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 File

This file contains a mapping of host names to IP addresses. This assignment, however, is used only locally on the fil4 and is not visible for other computers in the LAN. This file is actually redundant if a local DNS server is started in addition.

1.10.4 imond Configuration

The file `etc/imond.conf` is constructed amongst others from the configuration variables `CIRC_x_NAME`, `CIRC_x_ROUTE`, `CIRC_x_CHARGEINT` and `CIRC_x_TIMES`. It can consist of up to 32 lines (except for comment lines). Each line has eight columns:

1. Range weekday to weekday
2. Range hour to hour
3. Device (`ippX` or `isdX`)
4. Circuit with default route: "yes"/"no"
5. Phone number
6. Name of the circuits
7. Phone charges per minute in Euros
8. Charge interval in seconds

Here an example:

| #day | hour | device | defroute | phone | name | charge | ch-int |
|-------|-------|--------|----------|--------------|------------|--------|--------|
| Mo-Fr | 18-09 | ipp0 | yes | 010280192306 | Addcom | 0.0248 | 60 |
| Sa-Su | 00-24 | ipp0 | yes | 010280192306 | Addcom | 0.0248 | 60 |
| Mo-Fr | 09-18 | ipp1 | yes | 019160 | Compuserve | 0.019 | 180 |
| Mo-Fr | 09-18 | isd2 | no | 0221xxxxxxx | Firma | 0.08 | 90 |
| Mo-Fr | 18-09 | isd2 | no | 0221xxxxxxx | Firma | 0.03 | 90 |
| Sa-Su | 00-24 | isd2 | no | 0221xxxxxxx | Firma | 0.03 | 90 |

Further explanations for Least-Cost-Routing can be found in the documentation of the package “base”.

1.10.5 The File `/etc/.profile`

The file `/etc/.profile` contains user-defined settings for the shell. To overwrite the default settings you have to create a file `etc/.profile` below the configuration directory. You may enter settings for the command prompt or abbreviations (so-called “Aliases”) here.

Important: *This file may not contain an `exit`!*

Examples:

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

1.10.6 Scripts in `/etc/profile.d/`

In the directory `/etc/profile.d/` one may store scripts that will be executed when starting a shell and thus may influence the shell’s environment. Typically OPT packages will place scripts there to define special environment variables necessary for the programs they contain.

If both scripts in `/etc/profile.d/` and the file `/etc/.profile` exist, scripts in `/etc/profile.d/` will be executed *after* the script `/etc/.profile`.

List of Figures

List of Tables

| | | |
|-----|---|----|
| 1.1 | Parameters for <code>mkfli41</code> | 6 |
| 1.2 | Options for Files | 9 |
| 1.3 | Logical Epressions | 31 |
| 1.4 | Functions of the <code>cgi-helper</code> script | 46 |

Index

DEBUG_ENABLE_CORE, [39](#)
DEBUG_IP, [39](#)
DEBUG_IPUP, [39](#)
DEBUG_KEEP_BOOTLOGD, [40](#)
DEBUG_MDEV, [40](#)

LOG_BOOT_SEQ, [39](#)