Interface Tools*

Anthony Towns
aj@azure.humbug.org.au

August 24, 1999

Contents

1 Introduction

This source defines the commands `ifup` and `ifdown`, used to manipulate interfaces in an easily controllable manner.

1.1 Assumed Knowledge

The reader is assumed to have knowledge of the C [1] and Perl [2] programming languages in a Unix environment. A cursory understanding of network administration on the appropriate platform is also assumed, along with access to the relevant manual pages as necessary.

This source has been written as a literate program using the `noweb` [3] tool suite, and typeset using LaTeX [4].

1.2 Program Structure

We shall decompose this program into four main areas of functionality: compile-time configuration, run-time configuration, execution, and the overall driver.

Compile-time configuration will deal with differing available address families (IPX vs TCP/IP, and so on), and the differing methods of enabling and disabling interfaces configured for each family. This will be implemented using the `addrfam` module, and various `.defn` files, for the address family definitions.

Run-time configuration will deal with determining the local setup based on the file `/etc/network/interfaces`, and producing a data structure encapsulating these details. This will be implemented in the `config` module.

Execution will deal with issues relating to working out exactly which commands to run based on a somewhat abstract description from the compile-time configuration and the details determined at run-time. This will be dealt with in the `execute` module.

The remaining work — argument parsing, error reporting, and, essentially, putting all the pieces together — is done by the `main` module.

We shall find that these modules are too closely linked to be completely separated in a convenient manner, so they will all make use of a single header file for each other's structure definitions, exported interfaces and so on.

1 ⟨*header.h* 1⟩≡

```
#ifndef HEADER_H
#define HEADER_H

⟨type definitions 14⟩
⟨function type definitions 19⟩
⟨structure definitions 15⟩
⟨constant definitions 120⟩
⟨exported symbols 16⟩

#endif /* HEADER_H */
```

2 The Build System

We shall begin with the template for the Makefile we shall use.

2 ⟨*Makefile* 2⟩≡

```
⟨make options 6⟩

CFILES := addrfam.c execute.c config.c main.c archlinux.c
HFILES := header.h archlinux.h
PERL := defn2c.pl
DEFNFILES := inet.defn ipx.defn inet6.defn

OBJ := main.o addrfam.o execute.o config.o inet.o ipx.o inet6.o archlinux.o

executables : ifup ifdown
all : executables ifupdown.ps.gz

.PHONY : executables
⟨phony targets 4⟩
⟨executable targets 3⟩
⟨implicit rules 5⟩

include ifupdown.d
```

We shall build exactly two executables, `ifup` and `ifdown`, which will in truth simply be two names for a single binary, albeit with different functionality.

3 ⟨*executable targets* 3⟩≡ (2)

```
ifup: $(OBJ)
        $(CC) $(CFLAGS) -o ifup $^

ifdown: ifup
        ln -s ifup ifdown
```

Further, for convenience, we'll make use of two phony targets, `clean` and `clobber`, which will delete working files, and everything that can be rebuilt, respectively.

4 ⟨*phony targets* 4⟩≡ (2)

```
.PHONY : clean clobber

install :
        install -m 0755 -d     ${BASEDIR}/sbin
        install -m 0755 ifup   ${BASEDIR}/sbin
        ln ${BASEDIR}/sbin/ifup ${BASEDIR}/sbin/ifdown

clean :
        rm -f *.aux *.toc *.log *.bbl *.blg *.ps
        rm -f *.o $(patsubst %.defn,%.c,$(DEFNFILES)) *~
        rm -f ifup ifdown ifupdown.dvi ifupdown.ps.gz

clobber : clean
        rm -f ifupdown.tex $(PERL) $(CFILES) $(HFILES) $(DEFNFILES) *.d
```

We have some fairly standard rules to build the printed version of the source code using LaTeX that are, unfortunately, not included in `make(1)`'s builtin rules, so we'll note them here.

5 ⟨*implicit rules* 5⟩≡ (2) 7 ▷

```
%.tex : %.nw
        noweave -index -latex $< >$@

%.bbl : %.tex biblio.bib
        latex $<
        bibtex $(basename $<)

%.dvi : %.tex %.bbl
        latex $<
        latex $<

%.ps : %.dvi
        dvips -o $@ $<

%.gz : %
        gzip --best --stdout $< >$@
```

Additionally, some of `make`'s builtin rules are fairly conservative, so we'll encourage it to use a more entertaining method of compiling source code.

6 ⟨*make options* 6⟩≡ (2)

```
CFLAGS := -Wall -W -g -O2
CC := gcc
```

2.1 Automatic Dependencies

To build the system, we'll make use of some techniques discussed in [5] for determining dependencies. Namely, a number of files will be associated with a `.d` file containing dynamically determined dependency information. The first such file we will construct is the dependency information for `noweb` source files, which can be identified by the `.nw` extension.

7 ⟨*implicit rules* 5⟩+≡ (2) ◁ 5 11 ▷

```
%.d: %.nw makenwdep.sh
        ./makenwdep.sh $< > $@
```

To construct the dependency information, we may use the `noroots(1)` command to determine the *root chunks* in the `noweb` source (stripping the unwanted `<<` and `>>` markers as we go, and denoting that in such a way that `noweb` doesn't mistakenly think the `sed` command is a chunk reference itself), and then noting down appropriate commands to construct the target.

8 ⟨*makenwdep.sh* 8⟩≡

```
⟨parse makenwdep arguments 10⟩

noroots $FILE | sed 's/^<''<\(.*\)>>$/\1/' |
```

```
while read chunk; do
        ⟨output dependency info for $chunk 9⟩
done
```

Our dependency information is straightforward. To construct a file from `noweb` source, we simply need to run `notangle(1)` over it. We add a couple of extra tweeks in order to only update files that were actually changed (the `cpif(1)` call), and to handle tabs properly.

We also need some extra things to take care of particular types of files. In particular its important to have our scripts marked executable, so we can use them as part of the build process itself, and it's also important to have the dependency information for our C files (which are dealt with next) included at some point.

9 ⟨*output dependency info for* `$chunk` 9⟩≡ (8)

```
case $chunk in
        *.pl|*.sh)
                echo -e "$chunk : $FILE"
                echo -e "\tnotangle -R\$@ $< >\$@"
                echo -e "\tchmod 755 $chunk"
                ;;
        *.c)
                echo -e "$chunk : $FILE"
                echo -e "\tnotangle -L -R\$@ $< >\$@"
                echo -e "include ${chunk%.c}.d"
                ;;
        *.h)
                echo -e "$chunk : $FILE"
                echo -e "\tnotangle -L -R\$@ $< | cpif \$@"
                ;;
        *)
                echo -e "$chunk : $FILE"
                echo -e "\tnotangle -t8 -R\$@ $< >\$@"
                ;;
esac
```

Finally, our fairly primitive argument parsing is simply:

10 ⟨*parse makenwdep arguments* 10⟩≡ (8)

```
FILE=$1

if [ "$FILE" = "" -o ! -f "$FILE" ]; then
        echo "Please specify a .nw file"
        exit 1
fi
```

We have a related system for object files generated from C source code. Since each object file depends not only on its source, but also the headers included in that source, we generate a `.d` file indicating exactly which headers need to be checked.

11 ⟨*implicit rules* 5⟩+≡ (2) ◁7 21▷

```
%.d: %.c makecdep.sh
        ./makecdep.sh $< > $@
```

We can do this using `gcc(1)`'s convenient `-MM -MG` options, which do exactly this, with the added proviso that the `.d` file itself can possibly depend on any of the header files being modified (and, in particular, `#include` lines being added or deleted).

12 ⟨*makecdep.sh* 12⟩≡

```
#!/bin/sh
⟨parse makecdep arguments 13⟩

gcc -MM -MG $FILE |
  sed -e 's@^\(.*\)\.o:@\1.o \1.d:@'
```

Deja vu, anyone?

13 ⟨*parse makecdep arguments* 13⟩≡ (12)

```
FILE=$1
if [ "$FILE" = "" -o ! -f "$FILE" ]; then
        echo "Please specify a .c file"
        exit 1
fi
```

3 Compile Time Configuration

At compile time we need to determine all the possible address families that may be used, and all the methods of setting up interfaces for those address families, along with the various possible options affecting each method.

Our key definition at this point is that of the `address_family` structure, which encapsulates all the compile time information about each address family.

14 ⟨*type definitions* 14⟩≡ (1) 17▷

```
typedef struct address_family address_family;
```

15 ⟨*structure definitions* 15⟩≡ (1) 18▷

```
struct address_family {
        char *name;
        int n_methods;
        method *method;
};
```

Each defined address family will be included in the `addr_fams` array, which becomes the *raison d'être* of the `addrfam` module.

16 ⟨*exported symbols* 16⟩≡ (1) 51▷

```
extern address_family *addr_fams[];
```

Each address family incorporates a number of methods, which encapsulate various ways of configuring an interface for a particular address family. There are two definining components of a method: two sets of commands to bring an interface up and down, and a number of options for the commands.

NB: I expect this will be extended sooner or later to make the options more flexible. Probably introducing some form of typing (for example to convert netmasks from CIDR specs to dotted-quads, ie /24 → 255.255.255.0), and some form of matching to allow multiple options to be deduced from a single configuration statement. — aj

17 ⟨*type definitions* 14⟩+≡ (1) ◁14 47▷

```
typedef struct method method;
```

18 ⟨*structure definitions* 15⟩+≡ (1) ◁15 48▷

```
struct method {
        char *name;
        command_set *up, *down;
        int n_options;
        char **option;
};
```

Each command set is implemented as a single function, accepting two parameters: the definitions of the interface the commands should deal with, and the function that should be used to execute them. See the `execute` module for more details.

19 ⟨*function type definitions* 19⟩≡ (1)

```
typedef int (execfn)(char *command);
typedef int (command_set)(interface_defn *ifd, execfn *e);
```

As our compile-time configuration is done at, well, compile-time, there is little need for functions in the actual module, and we can make do with a single exported array.

20 ⟨*addrfam.c* 20⟩≡

```
#include <stdlib.h>
#include "header.h"

⟨address family declarations 166⟩

address_family *addr_fams[] = {
        ⟨address family references 167⟩
        NULL
};
```

3.1 Address Family Definition Files

Each address family is defined by a single `.defn` file. Unfortunately, while this representation is reasonably convenient for human use, it's less convenient for a compiler. As such, at build time, we will build a single structure of type `address_family` in a separate module, and reference that from `addrfam.c`.

Naturally, we'll use a `perl` script to convert `.defn` files to C code.

21 ⟨*implicit rules* 5⟩+≡ (2) ◁11

```
%.c : %.defn defn2c.pl
        ./defn2c.pl $< > $@
```

The functionality of our program is pretty basic: read from the file specified as the argument, output to `stdout`; and correspondingly the structure of the program is similarly simple. We make use of a couple of global variables, a few helpful subroutines, and then build our C program.

22 ⟨*defn2c.pl* 22⟩≡

```
#!/usr/bin/perl -w

use strict;

# declarations
⟨defn2c variables 24⟩

# subroutines
⟨defn2c subroutines 29⟩

# main code
⟨output headers for address family 23⟩
⟨parse .defn file and output intermediate structures 30⟩
⟨output address family data structure 25⟩
```

Clearly we need to reference some of the data structures we defined above, so we can begin with the rather trivial:

23 ⟨*output headers for address family* 23⟩≡ (22)

```
print "#include \"header.h\"\n\n\n";
```

The overall purpose of the C code we're trying to construct is to define a structure of type `address family`, and have it externally visible. So we'd like to declare a statically-initialized structure of this type, and be done with it.

To do this, however, we need to have some way of uniquely identifying the structure to avoid naming conflicts. We'll do this by assuming we have a previously initialized variable, `$address_family`, and that names of the form `addr_foo` won't be used elsewhere.

24 ⟨*defn2c variables* 24⟩≡ (22) 26▷

```
my $address_family = "";
```

We also need to reference an array of pointers to this address family's `methods`, which will have to be initialized separately. We'll assume that previous code will create such a structure, and call it (imaginativly) `methods`.

25 ⟨*output address family data structure* 25⟩≡ (22)

```
⟨output methods data structure 27⟩

print <<EOF;
address_family addr_${address_family} = {
```

```
		"$address_family",
		sizeof(methods)/sizeof(struct method),
		methods
	};
	EOF
```

Our `methods` array will be a little more complicated to construct. The first difficulty is that it will actually require some state from having read the `.defn` file. To handle this, we'll introduce a hash `%methods` that has the value `1` for each method from the `.defn` file. We use a hash instead of a list because it makes some error checking more convenient later.

26 ⟨*defn2c variables* 24⟩+≡ (22) ◁24 28▷

```
my %methods = ();
```

We'll use standard names, such as `foo_up`, `foo_down`, and `foo_options` for the various elements of each method which cannot be defined inline. The two functions and the array will be declared static to avoid name conflicts.

27 ⟨*output* `methods` *data structure* 27⟩≡ (25)

```
print "static method methods[] = {\n";
my $method;
foreach $method (keys %methods) {
	print <<EOF;
	{
		"$method",
		${method}_up, ${method}_down,
		sizeof(${method}_options)/sizeof(char*),
		${method}_options
	},
EOF
}
print "};\n\n";
```

In a reasonably obvious manner we can then proceed to process the `.defn` file to initialize the aforementioned variables, and to output the method-specific functions and array. We'll begin by defining a variable to keep track of the current line.

28 ⟨*defn2c variables* 24⟩+≡ (22) ◁26 31▷

```
my $line = "";
```

Our semantics for this variable will be basically that it contains a valid, meaningful input line. It won't be blank, or contain comments, and if there aren't any more lines to be read, it will evaluate to `false`. In order to keep these semantics, we'll use the subroutine `nextline` to update it.

29 ⟨*defn2c subroutines* 29⟩≡ (22) 32▷

```
sub nextline {
	$line = <>;
	while($line and ($line =~ /^#/ or $line =~ /^\s*$/)) {
		$line = <>;
	}
```

```
        if (!$line) { return 0; }
        chomp $line;
        return 1;
}
```

Our high-level logic then looks basically like:

30 ⟨*parse* `.defn` *file and output intermediate structures* 30⟩≡ (22)

```
nextline;
while($line) {
        ⟨parse a top-level section 33⟩

        # ...otherwise
        die("Unknown command \"$line\"");
}
```

To make all this stuff easier, we'll use a 'matching' function to help with parsing lines: basically, given a line, a command, and possibly an indentation prefix (eg `"        "`).

31 ⟨*defn2c variables* 24⟩+≡ (22) ◃28

```
my $match = "";
```

32 ⟨*defn2c subroutines* 29⟩+≡ (22) ◃29 34▹

```
sub match {
        my $line = $_[0];
        my $cmd = "$_[1]" ? "$_[1]\\b\\s*" : "";;
        my $indentexp = (@_ == 3) ? "$_[2]\\s+" : "";

        if ($line =~ /^${indentexp}${cmd}(([^\s](.*[^\s])?)?)\s*$/) {
                $match = $1;
                return 1;
        } else {
                return 0;
        }
}
```

Okay. So, the first line we expect to see is the name of the address family we're defining.

33 ⟨*parse a top-level section* 33⟩≡ (30) 35▹

```
if (match($line, "address_family")) {
        get_address_family $match;
        next;
}
```

This is, as you'd imagine, pretty simple to deal with. We just need to store the address family's name, and move on to the next line.

34 ⟨*defn2c subroutines* 29⟩+≡ (22) ◃32 36▹

```
sub get_address_family {
        $address_family = $_[0] if ($address_family eq "");
        nextline;
}
```

Which brings us to determining the architecture.

35 ⟨*parse a top-level section* 33⟩+≡ (30) ◁33 37▷

```
if (match($line, "architecture")) {
        get_architecture $match;
        next;
}
```

You'd never guess what, but it's just as easy as the address family thing was.

36 ⟨*defn2c subroutines* 29⟩+≡ (22) ◁34 38▷

```
sub get_architecture {
        my $arch = $_[0];
        die("architecture declaration appears too late") if (keys %methods);
        print "#include \"arch${arch}.h\"\n\n\n";
        nextline;
}
```

Which leaves us with the hard bit, actually creating the functions and array for each method.

37 ⟨*parse a top-level section* 33⟩+≡ (30) ◁35

```
if (match($line, "method")) {
        get_method $match;
        next;
}
```

The basic premise is to check for each of our options in a given order: if they don't match, then we can presume they don't exist — any errors will be reported when the main function finds something weird going on. All we really have to take care of so far is ensuring an appropriate level of indentation, and that we're not defining the same method twice.

38 ⟨*defn2c subroutines* 29⟩+≡ (22) ◁36 40▷

```
sub get_method {
        my $method = $_[0];
        my $indent = ($line =~ /(\s*)[^\s]/) ? $1 : "";

        die "Duplicate method $method\n" if ($methods{$method}++);

        nextline;
        ⟨check for options list 39⟩
        ⟨check for up commands 41⟩
        ⟨check for down commands 42⟩
}
```

Checking the various components of each method is fairly simple: we need to see if it exists, and if it does, parse and output it, while if it doesn't, we need to output a place holder. For the `options` section, this looks like:

39 ⟨*check for options list* 39⟩≡ (38)

```
if (match($line, "options", $indent)) {
        get_options(${method});
} else {
        print "static char *${method}_options[] = { };\n"
}
```

Actually parsing this section is fairly easy at the moment — all we have to do is construct yet another static array containing the various options listed.

40 ⟨*defn2c subroutines* 29⟩+≡ (22) ◁38 43▷

```
sub get_options {
        my $struct = $_[0];
        my $indent = ($line =~ /(\s*)[^\s]/) ? $1 : "";

        print "static char *${struct}_options[] = {\n";
        while (nextline && match($line, "", $indent)) {
                print "    \"$match\",\n";
        }
        print "};\n";
}
```

We have a similar sort of thing for the up and down functions.

41 ⟨*check for up commands* 41⟩≡ (38)

```
if (match($line, "up", $indent)) {
        get_commands(${method}, "up");
} else {
        print "static int ${method}_up(interface_defn ifd) { return 0; }\n"
}
```

42 ⟨*check for down commands* 42⟩≡ (38)

```
if (match($line, "down", $indent)) {
        get_commands(${method}, "down");
} else {
        print "static int ${method}_down(interface_defn ifd) { return 0; }\n"
}
```

43 ⟨*defn2c subroutines* 29⟩+≡ (22) ◁40

```
sub get_commands {
        my $method = $_[0];
        my $mode = $_[1];
        my $function = "${method}_${mode}";
        my $indent = ($line =~ /(\s*)[^\s]/) ? $1 : "";

        print "static int ${function}(interface_defn *ifd, int (*exec)(char*)) {\n";

        while (nextline && match($line, "", $indent)) {
                if ( $match =~ /^(.*[^\s])\s+if\s*\((.*)\)\s*$/ ) {
                        print "if ( $2 ) if (!execute(\"$1\", ifd, exec)) return 0;\n";
                } else {
                        print "if (!execute(\"$match\", ifd, exec)) return 0;\n";
```

```
                }
        }

        print "return 1;\n";
        print "}\n";
}
```

4 Run-time Configuration

Our module is of the usual form, and we'll make use of a few fairly standard headers. Please move along, there's nothing to see here.

44 ⟨*config.c* 44⟩≡
```
⟨config headers 45⟩
⟨config function declarations 56⟩
⟨config functions 52⟩
```

45 ⟨*config headers* 45⟩≡ (44) 46 ▷
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
```

We'll also make use of some of our other modules. This is, after all, why we had a single header in the first place.

46 ⟨*config headers* 45⟩+≡ (44) ◁ 45 59 ▷
```
#include "header.h"
```

The key function we're interested in defining here is `read_interfaces()`, which will (wait for it) read an interfaces file. The intention is to make it really easy to deal with the vagaries of `/etc/network/interfaces` anywhere else.

So the first question we need to deal with is "What's a convenient form for other functions which deal with interfaces?" Our answer is basically a singly linked list of structures that essentially reflect the original configuration file.

At run-time, we require a way of representing each interface listed in the configuration file. This naturally needs to reference an address family and method, and all the options a user may specify about an interface.

47 ⟨*type definitions* 14⟩+≡ (1) ◁ 17 49 ▷
```
typedef struct interface_defn interface_defn;
```

48 ⟨*structure definitions* 15⟩+≡ (1) ◁ 18 50 ▷
```
struct interface_defn {
        interface_defn *next;

        char *iface;
        char *scheme;
        address_family *address_family;
        method *method;
```

```
        int automatic;

        int max_options;
        int n_options;
        variable *option;
};
```

The last component in the above, the options, is represented by a series of name/value pairs. *This is almost certain to change — aj.*

49 ⟨*type definitions* 14⟩+≡ (1) ◁47

```
typedef struct variable variable;
```

50 ⟨*structure definitions* 15⟩+≡ (1) ◁48

```
struct variable {
        char *name;
        char *value;
};
```

We can thus begin to instantiate our actual function. What we want is something that, given the name of a file, will produce the appropriate linked list of interfaces defined in it, or possibly give some sort of helpful error message. Pretty simple, hey?

51 ⟨*exported symbols* 16⟩+≡ (1) ◁16 110▷

```
interface_defn *read_interfaces(char *filename);
```

52 ⟨*config functions* 52⟩≡ (44) 61▷

```
interface_defn *read_interfaces(char *filename) {
        ⟨variables local to read interfaces 53⟩
        interface_defn *ifaces = NULL;

        ⟨open file or return NULL 54⟩

        while (⟨we've gotten a line from the file 58⟩) {
                ⟨process the line 79⟩
        }
        if (⟨an error occurred getting the line 60⟩) {
                ⟨report internal error and die 100⟩
        }

        ⟨close file 55⟩

        return ifaces;
}
```

4.1 File Handling

So, the first and most obvious thing to deal with is the file handling. Nothing particularly imaginative here.

53 ⟨*variables local to read interfaces* 53⟩≡ (52) 57▷
```
FILE *f;
```

54 ⟨*open file or* `return NULL` 54⟩≡ (52)
```
f = fopen(filename, "r");
if ( f == NULL ) return NULL;
```

55 ⟨*close file* 55⟩≡ (52)
```
fclose(f);
```

4.2 Line Parsing

Our next problem is to work out how to read a single line from our input file. While this is nominally easy, we also want to deal nicely with things like continued lines, comments, and very long lines.

So we're going to have to write and make use of a complicated little function, which we'll imaginatively call `get_line()`. It will need a pointer to the file it's reading from, as well as a buffer to store the line it finds. Since this buffer's size can't be known in advance we'll need to make it `realloc()`-able, which means we need to pass around references to both the buffer's location (which may change), and it's size (which probably will). Our function declaration is thus:

56 ⟨*config function declarations* 56⟩≡ (44) 77▷
```
static int get_line(char **result, size_t *result_len, FILE *f);
```

To use it, we'll need a couple of variables to stores the buffer's location, and it's current length.

57 ⟨*variables local to read interfaces* 53⟩+≡ (52) ◁53 75▷
```
char *buf = NULL;
size_t buf_len = 0;
```

Given these, and presuming we can actually implement the function, our key chunk can thus be implemented simply as:

58 ⟨*we've gotten a line from the file* 58⟩≡ (52)
```
get_line(&buf,&buf_len,f)
```

We'll also add the requirement that errors are indicated by the `errno` variable being non-zero, which is usual and reasonable for all the circumstances where `get_line()` might have problems.

59 ⟨*config headers* 45⟩+≡ (44) ◁46 69▷
```
#include <errno.h>
```

60 ⟨*an error occurred getting the line* 60⟩≡ (52)
```
errno != 0
```

Actually defining the function is, as you'd probably imagine, a little more complicated. We begin by reading a line from the file. If it was a comment (that is, it has a `#` character at the first non-blank position) then we try again. Otherwise, if the line is continued (indicated by a `\` character at the very end of the line) we append the next line to the buffer. We go to a little bit of effort to trim whitespace, and finally return a boolean result indicating whether we got a line or not.

61 ⟨*config functions* 52⟩+≡ (44) ◁52 78▷
```
static int get_line(char **result, size_t *result_len, FILE *f) {
        ⟨variables local to get line 62⟩

        do {
                ⟨clear buffer 63⟩
                ⟨append next line to buffer, or return 0 64⟩
                ⟨trim leading whitespace 70⟩
        } while (⟨line is a comment 72⟩);

        while (⟨buffer is continued 73⟩) {
                ⟨remove continuation mark 74⟩
                ⟨append next line to buffer, or return 0 64⟩
        }

        ⟨trim trailing whitespace 71⟩

        return 1;
}
```

In order to do string concatenation efficiently, we'll keep track of where the end of the line so far is — this is thus where the terminating `NUL` will be by the end of the function.

62 ⟨*variables local to get line* 62⟩≡ (61)
```
size_t pos;
```

We can thus clear the buffer by simply resetting where we append new text to the beginning of the buffer. What could be simpler?

63 ⟨*clear buffer* 63⟩≡ (61)
```
pos = 0;
```

We'll be making use of the `fgets()` function to read the line (rather than, say, `fgetc()`) so to get an entire line we may have to make multiple attempts (if the line is bigger than our buffer). Realising this, and the fact that we may not have any allocated space for our buffer initially, we need a loop something like:

64 ⟨*append next line to buffer, or* `return 0` 64⟩≡ (61)
```
do {
        ⟨reallocate buffer as necessary, or return 0 65⟩
        ⟨get some more of the line, or return 0 67⟩
} while(⟨the line isn't complete 66⟩);
```

```
⟨remove trailing newline 68⟩

assert( (*result)[pos] == '\0' );
```

When reallocating the buffer, we need to make sure it increases in chunks large enough that we don't have to do this too often, but not so huge that we run out of memory just to read an 81 character line. We'll use two fairly simple heuristics for this: if we've got room to add no more than 10 characters, we may as well reallocate the buffer, and when reallocating, we want to more or less double the buffer, but we want to at least add 80 characters. So we do both.

65 ⟨*reallocate buffer as necessary, or* `return 0` 65⟩≡ (64)

```
if (*result_len - pos < 10) {
        char *newstr = realloc(*result, *result_len * 2 + 80);
        if (newstr == NULL) {
                return 0;
        }
        *result = newstr;
        *result_len = *result_len * 2 + 80;
}
```

The only time we need to keep reading is when the buffer wasn't big enough for the entire line. This is indicated by a full buffer, with no newline at the end. There is, actually, one case where this can happen legitimately — where the last line of the file is *exactly* the length of the buffer. We need to detect this because `fgets()` will return `NULL` and indicate that it's hit the end of the file, but we won't want to indicate that until the *next* time we try to get a line. Complicated, isn't it?

66 ⟨*the line isn't complete* 66⟩≡ (64)

```
pos == *result_len - 1 && (*result)[pos-1] != '\n'
```

So having thought through all that, actually working with `fgets()` is fairly simple, especially since we deal with the actual errors elsewhere. All we need to do is make the call, update `pos` and check that the problems `fgets()` may have actually bother us.

67 ⟨*get some more of the line, or* `return 0` 67⟩≡ (64)

```
if (!fgets(*result + pos, *result_len - pos, f)) {
        if (errno == 0 && pos == 0) return 0;
        if (errno != 0) return 0;
}
pos += strlen(*result + pos);
```

`fgets()` leaves a `\n` in our buffer in some cases. We're never actually interested in it, however, so it's a good move to get rid of it.

68 ⟨*remove trailing newline* 68⟩≡ (64)

```
if (pos != 0 && (*result)[pos-1] == '\n') {
        (*result)[--pos] = '\0';
}
```

Pretty simple, hey? Now the next thing we want to do is get rid of some of the whitespace lying about. This is all pretty basic, and just involves finding where the whitespace begins and ends, and, well, getting rid of it.

69 ⟨*config headers* 45⟩+≡ (44) ◁59
```
#include <ctype.h>
```

70 ⟨*trim leading whitespace* 70⟩≡ (61)
```
{
        int first = 0;
        while (isspace((*result)[first]) && (*result)[first]) {
                first++;
        }

        memmove(*result, *result + first, pos - first + 1);
        pos -= first;
}
```

71 ⟨*trim trailing whitespace* 71⟩≡ (61)
```
while (isspace((*result)[pos-1])) { /* remove trailing whitespace */
        pos--;
}
(*result)[pos] = '\0';
```

As we mentioned earlier, a line is a comment iff it's first character is a `#` symbol. Similarly, it's continued iff it's very last character is a `\`. And, rather obviously, if we want to remove a single trailing `\`, we can do so by changing it to a `NUL`.

72 ⟨*line is a comment* 72⟩≡ (61)
```
(*result)[0] == '#'
```

73 ⟨*buffer is continued* 73⟩≡ (61)
```
(*result)[pos-1] == '\\'
```

74 ⟨*remove continuation mark* 74⟩≡ (61)
```
(*result)[--pos] = '\0';
```

4.3 Line Processing

So. We've gone to a lot of trouble to get a line that we can parse with a snap of our fingers, so we probably better snap to it, to mix some *cliché's*.

We have two bits of state that need to be maintained between lines, namely the name of the current scheme and the various details we've already obtained about the interface we're currently working with.

75 ⟨*variables local to read interfaces* 53⟩+≡ (52) ◁57 76▷
```
char scheme[80] = "*";
interface_defn *currif = NULL;
```

Since our configuration files are pretty basic, we can work out what any particular line means based on the first word in it. To cope with this, we'll thus make use of a couple of variables, one to store the first word, and the other to store the rest of the line.

76 ⟨*variables local to read interfaces* 53⟩+≡ (52) ◁75

```
char firstword[80];
char *rest;
```

To initialize these variables we'll make use of a function I'm overly fond of called `next_word()`. It copies the first word in a string to a given buffer, and returns a pointer to the rest of the buffer.

77 ⟨*config function declarations* 56⟩+≡ (44) ◁56 87▷

```
static char *next_word(char *buf, char *word, int maxlen);
```

78 ⟨*config functions* 52⟩+≡ (44) ◁61 89▷

```
static char *next_word(char *buf, char *word, int maxlen) {
        if (!buf) return NULL;
        if (!*buf) return NULL;

        while(!isspace(*buf) && *buf) {
                if (maxlen-- > 1) *word++ = *buf;
                buf++;
        }
        if (maxlen > 0) *word = '\0';

        while(isspace(*buf) && *buf) buf++;

        return buf;
}
```

So after all this, there are basically three different sorts of line we can get: the start of a new scheme, the start of a new interface or an option for whatever interface we're currently working with.

79 ⟨*process the line* 79⟩≡ (52)

```
rest = next_word(buf, firstword, 80);
if (rest == NULL) continue; /* blank line */

if (strcmp(firstword, "scheme") == 0) {
        ⟨process scheme line 80⟩
} else if (strcmp(firstword, "iface") == 0) {
        ⟨process iface line 81⟩
} else {
        ⟨process option line 97⟩
}
```

4.3.1 Scheme line

Coping with a new scheme is embarassingly easy — all we need to do is change the `scheme` value to be whatever the next word in the line was.

80 ⟨*process* `scheme` *line* 80⟩≡ (79)
```
rest = next_word(rest, scheme, 80);
if (!rest) {
        ⟨report missing scheme name and die 101⟩
}
```

4.3.2 Interface line

Declaring a new interface is somewhat more interesting.

81 ⟨*process* `iface` *line* 81⟩≡ (79)
```
{
        ⟨variables local to process iface line 83⟩

        ⟨allocate new interface 82⟩

        ⟨parse interface settings 84⟩

        ⟨set iface name 85⟩
        ⟨set scheme 86⟩
        ⟨set address family 88⟩
        ⟨set method 91⟩
        ⟨set other options to defaults 93⟩

        ⟨add to list of interfaces 94⟩
}
```

We'll deal with each of these phases one by one and pretty much in order, so prepare yourself for the intense excitement of memory allocation!

82 ⟨*allocate new interface* 82⟩≡ (81)
```
currif = malloc(sizeof(interface_defn));
if (!currif) {
        ⟨report internal error and die 100⟩
}
```

When we introduce a new interface, we simultaneously name the interface, the address family, and the method. We cope with this by, well, getting somewhere to store each of them, and then, well, storing them. Action! Drama!

83 ⟨*variables local to process* `iface` *line* 83⟩≡ (81)
```
char iface_name[80];
char address_family_name[80];
char method_name[80];
```

84 ⟨*parse interface settings* 84⟩≡ (81)
```
rest = next_word(rest, iface_name, 80);
rest = next_word(rest, address_family_name, 80);
rest = next_word(rest, method_name, 80);

if (rest == NULL) {
        ⟨report too few parameters for iface line and die 102⟩
```

```
  }

  if (rest[0] != '\0') {
          ⟨report too many parameters for iface line and die 103⟩
  }
```

We then want to store the interface and scheme names. Crazy! Never to be repeated!

85 ⟨*set iface name* 85⟩≡ (81)

```
  currif->iface = strdup(iface_name);
  if (!currif->iface) {
          ⟨report internal error and die 100⟩
  }
```

86 ⟨*set scheme* 86⟩≡ (81)

```
  currif->scheme = strdup(scheme);
  if (!currif->scheme) {
          ⟨report internal error and die 100⟩
  }
```

Setting the address family is a little more involved, because it's not very useful to know what the name of the address family is, you really want to know all the details recorded in the appropriate `address_family` structure. So we'll make use of a little helper function, called `get_address_family()` to convert the useless string, to the hopefully less useless structure.

87 ⟨*config function declarations* 56⟩+≡ (44) ◁77 90▷

```
  static address_family *get_address_family(address_family *af[], char *name);
```

88 ⟨*set address family* 88⟩≡ (81)

```
  currif->address_family = get_address_family(addr_fams, address_family_name);
  if (!currif->address_family) {
          ⟨report unknown address family and die 104⟩
  }
```

Of course, we probably need to actually implement the function too. We won't do anything particularly fancy here, just a simple linear search. *Should this really be here, or an exported symbol from* `addrfam.c`*? — aj*

89 ⟨*config functions* 52⟩+≡ (44) ◁78 92▷

```
  static address_family *get_address_family(address_family *af[], char *name) {
          int i;
          for (i = 0; af[i]; i++) {
                  if (strcmp(af[i]->name, name) == 0) {
                          return af[i];
                  }
          }
          return NULL;
  }
```

We do something incredibly similar when dealing with the method the user wishes to use, and we do it for incredibly similar reasons. Again we declare a cute little helper function, this time imaginatively called `get_method()`, and then go and use it and implement in almost exactly the same way as before. I told you this was going to be a thrill. *The same note applies here, too — aj*

90 ⟨*config function declarations* 56⟩+≡ (44) ◃87 95▹
```
static method *get_method(address_family *af, char *name);
```

91 ⟨*set method* 91⟩≡ (81)
```
currif->method = get_method(currif->address_family, method_name);
if (!currif->method) {
        ⟨report unknown method and die 105⟩
        return NULL; /* FIXME */
}
```

92 ⟨*config functions* 52⟩+≡ (44) ◃89 96▹
```
static method *get_method(address_family *af, char *name) {
        int i;
        for (i = 0; i < af->n_methods; i++) {
                if (strcmp(af->method[i].name, name) == 0) {
                        return &af->method[i];
                }
        }
        return NULL;
}
```

You'll continue to be enthralled as we set the remaining options to some default values. Hurray! Yippee!

93 ⟨*set other options to defaults* 93⟩≡ (81)
```
currif->automatic = 1;
currif->max_options = 0;
currif->n_options = 0;
currif->option = NULL;
```

Since we want to keep the interfaces in order, we have to go all the way to the end of the list of interfaces to add the new interface, and we can hence set the `next` pointer to NULL in all cases. Gee. Whiz.

Actually, I'm selling this a little short. We also want to make sure we don't try instantiating the same interface twice or anything. So we take care of that too. There now. Didn't that just get the adrenalin pumping?

94 ⟨*add to list of interfaces* 94⟩≡ (81)
```
currif->next = NULL;

if (ifaces == NULL) {
        ifaces = currif;
} else {
        interface_defn *checkif;
        checkif = ifaces;
        for(;;) {
```

```
                if (duplicate_if(checkif, currif)) {
                        ⟨report duplicate interface and die 106⟩
                }
                if (!checkif->next) break;
                checkif = checkif->next;
        }

        checkif->next = currif;
}
```

Duplicate interfaces are interfaces that share the same scheme, interface name and address family — while they may have different methods, that alone just isn't enough to differentiate them.

There are two issues here. First, is that since schemes are actually shell globs, having them be different isn't actually enough to make sure they're not repeated. But I don't want to leave them as shell globs anyway, so this will kinda resolve itself. The second issue, is that duplicate interfaces can be interesting, if you can rig it up so that, for example, it first tries dhcp, then bootp, then gives up and uses a static address. This sort of consideration will probably have to be dealt with by the sysadmin though, rather than in here. So we'll choose not to make duplicate interfaces meaningful. — aj

95 ⟨*config function declarations* 56⟩+≡ (44) ◁90

```
static int duplicate_if(interface_defn *ifa, interface_defn *ifb);
```

96 ⟨*config functions* 52⟩+≡ (44) ◁92

```
static int duplicate_if(interface_defn *ifa, interface_defn *ifb) {
        if (strcmp(ifa->scheme, ifb->scheme) != 0) return 0;
        if (strcmp(ifa->iface, ifb->iface) != 0) return 0;
        if (ifa->address_family != ifb->address_family) return 0;
        return 1;
}
```

Stay tuned for more action and adventure.

4.3.3 Options line

The final sort of line we have to deal with are interface options. The basic structure of this is fairly straightforward — the only option we deal with specially is the `noauto` option.

97 ⟨*process option line* 97⟩≡ (79)

```
if (!currif) {
        ⟨report option without interface and die 107⟩
} else {
        if (strcmp(firstword,"noauto") == 0) {
                currif->automatic = 0;
        } else {
                ⟨add option 98⟩
        }
}
```

Adding an option is also fairly straightforward: we simply construct a new variable and add it at the end of our array of variables, increasing the size of the array first if necessary.

98 ⟨*add option* 98⟩≡ (97)
```
if (currif->n_options >= currif->max_options) {
        ⟨increase max number of options 99⟩
}

currif->option[currif->n_options].name = strdup(firstword);
currif->option[currif->n_options].value = strdup(rest);

if (!currif->option[currif->n_options].name) {
        ⟨report internal error and die 100⟩
}

if (!currif->option[currif->n_options].value) {
        ⟨report internal error and die 100⟩
}

currif->n_options++;
```

We'll increase the space for variables by a constant amount each time, rather than doubling or anything smart like that.

99 ⟨*increase max number of options* 99⟩≡ (98)
```
{
        variable *opt;
        currif->max_options = currif->max_options + 10;
        opt = realloc(currif->option, sizeof(variable) * currif->max_options);
        if (opt == NULL) {
                ⟨report internal error and die 100⟩
        }
        currif->option = opt;
}
```

4.4 Error Handling

We don't do anything too fancy about handling errors that occur, we just print out a hopefully helpful error message, and return from the function. *We probably should also go to some effort to close files, and free memory, but well. Maybe version $n + 1$. — aj*

100 ⟨*report internal error and die* 100⟩≡ (52 82 85 86 98 99)
```
perror(filename);
return NULL;
```

101 ⟨*report missing scheme name and die* 101⟩≡ (80)
```
fprintf(stderr, "%s: scheme name missing\n", filename);
return NULL;
```

102 ⟨*report too few parameters for iface line and die* 102⟩≡ (84)
```
fprintf(stderr, "%s: too few parameters for iface line\n", filename);
return NULL;
```

103 ⟨*report too many parameters for iface line and die* 103⟩≡ (84)
```
fprintf(stderr, "%s: too many parameters for iface line\n", filename);
return NULL;
```

104 ⟨*report unknown address family and die* 104⟩≡ (88)
```
fprintf(stderr, "%s: unknown address type\n", filename);
return NULL;
```

105 ⟨*report unknown method and die* 105⟩≡ (91)
```
fprintf(stderr, "%s: unknown method\n", filename);
return NULL;
```

106 ⟨*report duplicate interface and die* 106⟩≡ (94)
```
fprintf(stderr, "%s: duplicate interface\n", filename);
return NULL;
```

107 ⟨*report option without interface and die* 107⟩≡ (97)
```
fprintf(stderr, "%s: option without interface\n", filename);
return NULL;
```

5 Execution

The `execute` module will be laid out in the standard manner, and will make use of the usual header files.

108 ⟨*execute.c* 108⟩≡
⟨*execute headers* 109⟩
⟨*execute function declarations* 112⟩
⟨*execute functions* 111⟩

109 ⟨*execute headers* 109⟩≡ (108) 124 ▷
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "header.h"
```

The key function we'll export from here is `execute()`, which, given a command, an interface will fill in any missing details in the command, and execute it.

110 ⟨*exported symbols* 16⟩+≡ (1) ◁ 51
```
int execute(char *command, interface_defn *ifd, execfn *exec);
```

At the somewhat abstract level, this is fairly trivial. The devil is in the details of the parsing, which makes up the rest of the module.

111 ⟨*execute functions* 111⟩≡ (108) 113 ▷

```
int execute(char *command, interface_defn *ifd, execfn *exec) {
        char *out;
        int ret;

        out = parse(command, ifd);
        if (!out) { return 0; }

        ret = exec(out);

        free(out);
        return ret;
}
```

5.1 Command Parsing

We'll need a basic parser function, which we'll call `parse()` to make the appropriate substitutions into a command. It's probably worth a note as to exactly what substitutions may be made:

- Special characters can be escaped with a backslash. eg `ls MoreThan80\%`.
- Variables can be substituted by including their name delimeted by percents. eg `ls %directory%`.
- Optional components may be enclosed in double square brackets. Optional components will be included exactly when every variable referenced within exists. eg `ls [[--color=%color%]] %directory%`. Optional components may be nested.

Most of the parsing is fairly straightforward – basically, we keep an output buffer, and add things to it as we stroll through the input buffer: either the actual character we want, or whatever the value of the variable we're looking at is, or whatever. The only particularly complicated bit is how we deal with the optional sections, which will be explained when we get to them.

112 ⟨*execute function declarations* 112⟩≡ (108) 116 ▷

```
static char *parse(char *command, interface_defn *ifd);
```

113 ⟨*execute functions* 111⟩+≡ (108) ◁ 111 117 ▷

```
static char *parse(char *command, interface_defn *ifd) {
        ⟨variables local to parse 114⟩

        while(*command) {
                switch(*command) {
                        ⟨handle a token 118⟩
                }
```

```
        }

        ⟨deal with error conditions 126⟩

        ⟨return result 115⟩
}
```

5.1.1 Maintain output buffer

So the first thing we need to do is actually write some code to deal with the output buffer, which will need to be dynamically resized and so on to take care of possibly long strings and what-not. It is the caller's responsibility to `free()` this buffer. We'll maintain two extra variables for convenience: who much memory we've allocated `len`, and where the next character should be stuck `pos`.

114 ⟨*variables local to parse* 114⟩≡ (113) 121 ▷

```
char *result = NULL;
size_t pos = 0, len = 0;
```

This makes it pretty easy to return the result to the caller, too.

115 ⟨*return result* 115⟩≡ (113)

```
return result;
```

The main thing to be done to this buffer is to add characters or strings to it. To deal with this, we'll make use of an `addstr()` function that resizes the buffer as necessary, and appends a string to it. So we can deal with single characters, and substrings in general, we'll specify the string to be added as a pointer-length combination, rather than as a `NUL` terminated string.

116 ⟨*execute function declarations* 112⟩+≡ (108) ◁ 112 132 ▷

```
void addstr(char **buf, size_t *len, size_t *pos, char *str, size_t strlen);
```

117 ⟨*execute functions* 111⟩+≡ (108) ◁ 113 133 ▷

```
void addstr(char **buf, size_t *len, size_t *pos, char *str, size_t strlen) {
        assert(*len >= *pos);
        assert(*len == 0 || (*buf)[*pos] == '\0');

        if (*pos + strlen >= *len) {
                char *newbuf;
                newbuf = realloc(*buf, *len * 2 + strlen + 1);
                if (!newbuf) {
                        perror("realloc");
                        exit(1); /* a little ugly */
                }
                *buf = newbuf;
                *len = *len * 2 + strlen + 1;
        }

        while (strlen-- >= 1) {
                (*buf)[(*pos)++] = *str;
```

```
                str++;
        }
        (*buf)[*pos] = '\0';
}
```

Given this, we can define our default behaviour for a character:

118 ⟨*handle a token* 118⟩≡ (113) 119 ▷

```
default:
        addstr(&result, &len, &pos, command, 1);
        command++;
        break;
```

5.1.2 Escaped characters

We can also deal pretty simply with escaped tokens. The only special circumstance is if the \ is at the very end of string. We don't want buffer overflows afterall.

119 ⟨*handle a token* 118⟩+≡ (113) ◁ 118 122 ▷

```
case '\\':
        if (command[1]) {
                addstr(&result, &len, &pos, command+1, 1);
                command += 2;
        } else {
                addstr(&result, &len, &pos, command, 1);
                command++;
        }
        break;
```

5.1.3 Optional components

Basically we keep track of each optional section we're in, whether we've been unable to fill in any variables, and where we started it. When we reach the end of an optional section, we check to see if we were unable to fill in any variables, and, if so, we discard any text we'd added within that block. This also allows us to neatly check for any errors trying to fill in variables that aren't in optional sections.

Basically what we'll do here is keep one stack to represent where the various thingos started, and another to represent whether any variables didn't exist. We'll use the bottom-most entry in the stack to represent the entire command, and thus keep track of whether or not we have to return an error because an undefined variable was used in a non-optional part of the command.

120 ⟨*constant definitions* 120⟩≡ (1) 125 ▷

```
#define MAX_OPT_DEPTH 10
```

121 ⟨*variables local to parse* 114⟩+≡ (113) ◁ 114

```
size_t old_pos[MAX_OPT_DEPTH] = {0};
int okay[MAX_OPT_DEPTH] = {1};
int opt_depth = 1;
```

Given this, when we encounter a double open bracket, we need to just add the appropriate values to our stacks, and, similarly, when we encounter a double close bracket, we simply need to pop the stack, and see whether we need to move back or not, as well as taking care of an possible errors, naturally. *We probably could actually give error messages here instead of just treating the brackets literally when they might cause problems. But there doesn't seem much point, really. — aj*

122 ⟨*handle a token* 118⟩+≡ (113) ◁119 123▷

```
case '[':
        if (command[1] == '[' && opt_depth < MAX_OPT_DEPTH) {
                old_pos[opt_depth] = pos;
                okay[opt_depth] = 1;
                opt_depth++;
                command += 2;
        } else {
                addstr(&result, &len, &pos, "[", 1);
                command++;
        }
        break;
```

123 ⟨*handle a token* 118⟩+≡ (113) ◁122 128▷

```
case ']':
        if (command[1] == ']' && opt_depth > 1) {
                opt_depth--;
                if (!okay[opt_depth]) {
                        pos = old_pos[opt_depth];
                        result[pos] = '\0';
                }
                command += 2;
        } else {
                addstr(&result, &len, &pos, "]", 1);
                command++;
        }
        break;
```

Finally, at the end of the function, the stacks can be left in an unacceptable state — either one of the optional blocks was never closed, or an undefined variable was used elsewhere. We'll note these circumstances by returning `NULL` and setting `errno`.

124 ⟨*execute headers* 109⟩+≡ (108) ◁109

```
#include <errno.h>
```

125 ⟨*constant definitions* 120⟩+≡ (1) ◁120 127▷

```
#define EUNBALBRACK 10001
#define EUNDEFVAR   10002
```

126 ⟨*deal with error conditions* 126⟩≡ (113)
```
if (opt_depth > 1) {
        errno = EUNBALBRACK;
        free(result);
        return NULL;
}

if (!okay[0]) {
        errno = EUNDEFVAR;
        free(result);
        return NULL;
}
```

5.1.4 Variables

Dealing with variables is comparatively fairly simple. We just need to find the next percent, and see if whatever's inbetween is a variable, and, if so, get it's value.

127 ⟨*constant definitions* 120⟩+≡ (1) ◁125
```
#define MAX_VARNAME   32
#define EUNBALPER   10000
```

128 ⟨*handle a token* 118⟩+≡ (113) ◁123
```
case '%':
{
        ⟨variables local to handle percent token 129⟩
        char *varvalue;

        ⟨determine variable name 130⟩

        ⟨get varvalue 136⟩

        if (varvalue) {
                addstr(&result, &len, &pos, varvalue, strlen(varvalue));
        } else {
                okay[opt_depth - 1] = 0;
        }

        ⟨move to token after closing percent 131⟩

        break;
}
```

We don't do anything particularly clever dealing with the next percent — just a pointer to the appropriate character.

129 ⟨*variables local to handle percent token* 129⟩≡ (128)
```
char *nextpercent;
```

130 ⟨*determine variable name* 130⟩≡ (128)
```
command++;
nextpercent = strchr(command, '%');
if (!nextpercent) {
        errno = EUNBALPER;
        free(result);
        return NULL;
}
```

131 ⟨*move to token after closing percent* 131⟩≡ (128)
```
command = nextpercent + 1;
```

The slightly tricky thing we do here is use a `strncmpz` function, which allows us to check that a string represented by a `char*` and a length is the same as a `NUL` terminated string.

132 ⟨*execute function declarations* 112⟩+≡ (108) ◁116 134▷
```
int strncmpz(char *l, char *r, size_t llen);
```

133 ⟨*execute functions* 111⟩+≡ (108) ◁117 135▷
```
int strncmpz(char *l, char *r, size_t llen) {
        int i = strncmp(l, r, llen);
        if (i == 0)
                return -r[llen];
        else
                return i;
}
```

Given the above, the implementation of the `get_var()` function to lookup the value of a variable, is reasonably straight forward.

134 ⟨*execute function declarations* 112⟩+≡ (108) ◁132
```
char *get_var(char *id, size_t idlen, interface_defn *ifd);
```

135 ⟨*execute functions* 111⟩+≡ (108) ◁133
```
char *get_var(char *id, size_t idlen, interface_defn *ifd) {
        int i;

        if (strncmpz(id, "iface", idlen) == 0) {
                return ifd->iface;
        } else {
                for (i = 0; i < ifd->n_options; i++) {
                        if (strncmpz(id, ifd->option[i].name, idlen) == 0) {
                                return ifd->option[i].value;
                        }
                }
        }

        return NULL;
}
```

Which means we can finish of the chunk, thus:

136 ⟨*get* `varvalue` 136⟩≡ (128)

```
varvalue = get_var(command, nextpercent - command, ifd);
```

6 The Driver

*Guess where I stopped. You might be better off looking at the generated file to get the gist of what this does. Hopefully it's not **too** complicated.*

137 ⟨*main.c* 137⟩≡

⟨*main headers* 138⟩

⟨*main function declarations* 139⟩
⟨*main functions* 140⟩

⟨*main* 152⟩

138 ⟨*main headers* 138⟩≡ (137) 160 ▷

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <assert.h>
#include <getopt.h>

#include "header.h"
```

139 ⟨*main function declarations* 139⟩≡ (137) 141 ▷

```
static int check(char *str);
```

140 ⟨*main functions* 140⟩≡ (137) 142 ▷

```
static int check(char *str) {
        return str != NULL;
}
```

141 ⟨*main function declarations* 139⟩+≡ (137) ◁ 139 143 ▷

```
static int doit(char *str);
```

142 ⟨*main functions* 140⟩+≡ (137) ◁ 140 144 ▷

```
static int doit(char *str) {
        system(str);
        return 1;
}
```

143 ⟨*main function declarations* 139⟩+≡ (137) ◁ 141 145 ▷

```
static int printit(char *str);
```

144 ⟨*main functions* 140⟩+≡ (137) ◁142 146▷

```
static int printit(char *str) {
        printf("  %s\n", str); fflush(stdout);
        return 1;
}
```

145 ⟨*main function declarations* 139⟩+≡ (137) ◁143 147▷

```
static int execute_all(interface_defn *ifd, int(*exec)(char*), char *opt);
```

146 ⟨*main functions* 140⟩+≡ (137) ◁144 149▷

```
static int execute_all(interface_defn *ifd, int(*exec)(char*), char *opt) {
        int i;
        for (i = 0; i < ifd->n_options; i++) {
                if (strcmp(ifd->option[i].name, opt) == 0) {
                        if (!exec(ifd->option[i].value)) {
                                return 0;
                        }
                }
        }
        return 1;
}
```

147 ⟨*main function declarations* 139⟩+≡ (137) ◁145 148▷

```
static int up(interface_defn *iface, int (*exec)(char*));
```

148 ⟨*main function declarations* 139⟩+≡ (137) ◁147 150▷

```
static int up(interface_defn *iface, int (*exec)(char*));
```

149 ⟨*main functions* 140⟩+≡ (137) ◁146 151▷

```
static int up(interface_defn *iface, int (*exec)(char*)) {
        if (!iface->method->up(iface,exec)) return 0;
        if (!execute_all(iface,exec,"up")) return 0;

        return 1;
}
```

150 ⟨*main function declarations* 139⟩+≡ (137) ◁148

```
static int down(interface_defn *iface, int (*exec)(char*));
```

151 ⟨*main functions* 140⟩+≡ (137) ◁149 156▷

```
static int down(interface_defn *iface, int (*exec)(char*)) {
        if (!execute_all(iface,exec,"down")) return 0;
        if (!iface->method->down(iface,exec)) return 0;

        return 1;
}
```

152 ⟨*main* 152⟩≡ (137)

```
int main(int argc, char **argv) {
        int i;

        char *interfaces = "/etc/network/interfaces";

        interface_defn *ifaces = NULL;
        interface_defn *currif = NULL;

        ⟨variables local to main 153⟩

        ⟨parse command name 154⟩
        ⟨parse arguments 158⟩

        ifaces = read_interfaces(interfaces);
        if ( !ifaces ) {
                exit(1);
        }

        ⟨run cmds for appropriate interfaces 159⟩

        return 0;
}
```

153 ⟨*variables local to main* 153⟩≡ (152) 155 ▷

```
int (*cmds)(interface_defn *, int (*exec)(char*)) = NULL;
char *command;
```

154 ⟨*parse command name* 154⟩≡ (152)

```
command = strrchr(argv[0],'/');
if (command == NULL) {
        command = argv[0];
} else {
        command++;
}

if (strcmp(command, "ifup")==0) {
        cmds = up;
} else if (strcmp(command, "ifdown")==0) {
        cmds = down;
} else {
        fprintf(stderr,"This command should be called as ifup or ifdown\n");
        exit(1);
}
```

155 ⟨*variables local to main* 153⟩+≡ (152) ◁ 153

```
struct option long_opts[] = {
        {"scheme",     required_argument, NULL, 's'},
        {"help",       no_argument,       NULL, 'h'},
```

```
        {"verbose",    no_argument,       NULL, 'v'},
        {"all",        no_argument,       NULL, 'a'},
        {"interfaces", required_argument, NULL, 'i'},
        {"no-act",     no_argument,       NULL, 'n'},
        {0,0,0,0}
};

int do_all = 0;
int no_act = 0;
int verbose = 0;
char *real_scheme = "*";
```

156 ⟨*main functions* 140⟩+≡ (137) ◁151 157▷

```
static void usage(void) {
        fprintf(stderr, "Use --help for help\n");
        exit(1);
}
```

157 ⟨*main functions* 140⟩+≡ (137) ◁156

```
static void help(char *execname) {
        printf("Usage: %s -ash IFACES...\n\n", execname);
        printf("\t-a --all\t\tde/configure all interfaces automatically\n");
        printf("\t-s --scheme SCHEME\tuse SCHEME as scheme\n");
        printf("\t-h --help\t\tthis help\n");
        printf("\t-i --interfaces FILE\tuse FILE for interface definitions\n");
        printf("\t-n --no-act\t\tprint out what would happen, but don't do it\n");
        printf("\t-v --verbose\t\tprint out what would happen before doing it\n");
        exit(0);
}
```

158 ⟨*parse arguments* 158⟩≡ (152)

```
for (;;) {
        int c;
        c = getopt_long(argc, argv, "s:i:hva", long_opts, NULL);
        if (c == EOF) break;

        switch(c) {
                case 's':
                        real_scheme = strdup(optarg);
                        break;
                case 'i':
                        interfaces = strdup(optarg);
                        break;
                case 'h':
                        help(argv[0]);
                        break;
                case 'v':
                        verbose = 1;
                        break;
                case 'a':
```

```
                        do_all = 1;
                        break;
                case 'n':
                        no_act = 1;
                        break;
                default:
                        usage();
                        break;
        }
}

if (argc - optind == 0 && !do_all) {
        usage();
}

if (argc - optind > 0 && do_all) {
        usage();
}
```

159 ⟨*run* cmds *for appropriate interfaces* 159⟩≡ (152)

```
if (do_all) {
        ⟨run cmds for all interfaces 162⟩
} else {
        ⟨run cmds for given arguments 161⟩
}
```

160 ⟨*main headers* 138⟩+≡ (137) ◁138

```
#include <fnmatch.h>
```

161 ⟨*run* cmds *for given arguments* 161⟩≡ (159)

```
for (i = optind; i < argc; i++) {
        int okay;
        okay = 0;
        for (currif = ifaces; currif; currif = currif->next) {
                if (strcmp(argv[i],currif->iface) == 0
                        && fnmatch(currif->scheme, real_scheme, 0) == 0)
                {
                        okay = 1;
                        ⟨run cmds for currif 163⟩
                        break;
                }
        }
        if (!okay) {
                fprintf(stderr, "Ignoring unknown interface %s.\n",
                        argv[i]);
        }
}
```

162 ⟨*run* cmds *for all interfaces* 162⟩≡ (159)

```
for (currif = ifaces; currif; currif = currif->next) {
        if (!currif->automatic
                || fnmatch(currif->scheme, real_scheme, 0) != 0)
        {
                continue;
        }

        ⟨run cmds for currif 163⟩
}
```

163 ⟨*run* cmds *for* currif 163⟩≡ (161 162)

```
if ( !cmds(currif,check) ) {
        printf("Don't seem to be have all the variables for %s.\n", currif->iface);
} else {
        if (no_act || verbose) {
                cmds(currif,printit);
        }
        if (!no_act) {
                cmds(currif,doit);
        }
}
```

A Linux Address Families

164 ⟨*archlinux.h* 164⟩≡

```
unsigned int mylinuxver();
unsigned int mylinux(int,int,int);
```

165 ⟨*archlinux.c* 165⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/utsname.h>

#include "archlinux.h"

unsigned int mylinuxver() {
        static int maj = -1, rev, min;

        if (maj == -1) {
                struct utsname u;
                char *pch;
                uname(&u);
                maj = atoi(u.release);
                pch = strchr(u.release, '.');
                rev = atoi(pch+1);
                pch = strchr(pch+1, '.');
                min = atoi(pch+1);
```

```
        }

        return mylinux(maj,rev,min);
}

unsigned int mylinux(int maj, int rev, int min) {
        return min | rev << 10 | maj << 13;
}
```

A.1 IPv4 Address Family

166 ⟨*address family declarations* 166⟩≡ (20) 174 ▷

```
extern address_family addr_inet;
```

167 ⟨*address family references* 167⟩≡ (20) 175 ▷

```
&addr_inet,
```

168 ⟨*inet.defn* 168⟩≡

```
address_family inet
architecture linux

# isup
#     /* some code to check if an interface is up? */

⟨inet methods 169⟩
```

169 ⟨*inet methods* 169⟩≡ (168) 170 ▷

```
method loopback
    up
        ifconfig %iface% 127.0.0.1 up
        route add -net 127.0.0.0       if ( mylinuxver() < mylinux(2,1,100) )

    down
        ifconfig %iface% down
```

170 ⟨*inet methods* 169⟩+≡ (168) ◁ 169 171 ▷

```
method static
  options
    address
    netmask
    broadcast
    network
    gateway

#??  code
#??    given %address%: possibly calculate %netmask%
#??    given %address%, %netmask%: calculate network, broadcast if necessary

  up
    ifconfig %iface% %address% netmask %netmask% [[broadcast %broadcast%]] up
```

```
    route add -net %network%            if ( mylinuxver() < mylinux(2,1,100) )
    [[ route add default gw %gateway% ]]

  down
    ifconfig %iface% down
```

171 ⟨*inet methods* 169⟩+≡ (168) ◁170 172▷

```
method dhcp
  options
    vendor
    client
    hostname
    leasetime

  up
    dhcpcd [[-h %hostname%]] [[-i %vendor%]] [[-I %clientid%]] [[-l %leasetime%]] %iface%                if ( mylinuxver() < mylinux(2,1,100) )
    dhcpcd-sv [[-h %hostname%]] [[-i %vendor%]] [[-I %clientid%]] [[-l %leasetime%]] %iface%             if ( mylinuxver() >= mylinux(2,1,100) )

  down
    dhcpcd -k %iface%    if ( mylinuxver() < mylinux(2,1,100) )
    dhcpcd-sv -k %iface% if ( mylinuxver() >= mylinux(2,1,100) )
    ifconfig %iface% down
```

172 ⟨*inet methods* 169⟩+≡ (168) ◁171 173▷

```
method bootp
  options
    bootfile
    server
    hwaddr

  up
    bootpc [[--bootfile %bootfile%]] --dev %iface% [[--server %server%]] [[--hwaddr %hwaddr%]] --returniffail --serverbcast

  down
    ifconfig down %iface%
```

173 ⟨*inet methods* 169⟩+≡ (168) ◁172

```
method ppp
  up
    pon
  down
    poff
```

A.2 IPv6 Address Family

174 ⟨*address family declarations* 166⟩+≡ (20) ◁166 177▷

```
extern address_family addr_inet6;
```

175 ⟨*address family references* 167⟩+≡ (20) ◁167 178▷

```
&addr_inet6,
```

176 ⟨*inet6.defn* 176⟩≡

```
address_family inet6
architecture linux

method loopback
  up
    ifconfig %iface% add ::1
  down
    ifconfig %iface% del ::1
```

A.3 IPX Address Family

177 ⟨*address family declarations* 166⟩+≡ (20) ◁174

```
extern address_family addr_ipx;
```

178 ⟨*address family references* 167⟩+≡ (20) ◁175

```
&addr_ipx,
```

179 ⟨*ipx.defn* 179⟩≡

```
address_family ipx
architecture linux

method static
  options
    device
    ipx_frame
    ipx_netnum

  up
    ipx_interface add %device% %ipx_frame% %ipx_netnum%

  down
    ipx_interface del %device% %ipx_frame%
```

References

[1] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[2] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl.* O'Reilley and Associates, 1996.

[3] Norman Ramsey. noweb home page. http://www.cs.virginia.edu/~nr/noweb/.

[4] Leslie Lamport. *LaTeX— A Document Preparation System.* Addison Wesley, 1994.

[5] Peter Miller <millerp@canb.auug.org.au>. Recursive make considered harmful, 1997.