Introduction The job of this Awk script is to process a traditional UNIX mailbox file, determine the sender and recipient(s) of the mail as indicated by the `From:`, `To:`, `Cc:` and `Bcc:` mail headers, and then send the mail via an SMTP connection to a mail server.

The program relies on a not-yet-released version of `gawk` which can open sockets for reading and writing. As it's intended only for my use on my machine at home, and I maintain `gawk`, that's not a big deal.

The program will open a socket to the SMTP port on the mail server. For each message, it will send the appropriate SMTP commands and read the responses. It will try to do some minimal error checking and logging if mail can't be sent. Multiple messages can be sent by using sets of `MAIL FROM`, `RCPT TO` and `DATA` commands.

A Simple SMTP Server For Debugging Before writing an SMTP client, we need to have a simple SMTP "server" that we can use for testing. This server will send the appropriate replies to SMTP commands, capture incoming mail data, and log it to a log file in `/tmp` so that we can see what the sender is sending. As such, it should *not* un-escape any lines that start with the "." character.

1 ⟨*testserv.awk* 1⟩≡

```
#! /usr/local/bin/gawk-3.1.0 -f
BEGIN {
        ⟨server init 2a⟩
}

⟨rules for SMTP commands 4⟩

END {
        ⟨server close down 2b⟩
}

⟨log function 2c⟩
```

Root chunk (not used in this document).

Initialization is easy. We use gawk's IGNORECASE variable to ignore case, since SMTP commands can be any case, and we set the line terminators to CR-LF. Both of these steps are per RFC 821. Then we create the log file name.

For manually testing the debugging server where we type STMP commands from a terminal, we don't want to use CR-LF input/output. Instead, we want the normal behavior where newline ends records.

2a ⟨*server init* 2a⟩≡

```
IGNORECASE = 1          # SMTP commands are case independent
if (! Debug)
        RS = ORS = "\r\n"       # SMTP uses CR-LF input and output
Logfile = ("/tmp/ts.log." PROCINFO["pid"])      # unique log file name
```

Defines:
`Debug`, used in chunks 5, 18a, and 19a.
`Logfile`, used in chunks 2 and 7b.
This definition is continued in chunk 3b.
This code is used in chunk 1.

For debugging, we need to be able to log information. The BEGIN rule creates a unique log file name based on the process ID. One of the jobs in the END rule is to close that file (even though we can just let gawk close it for us, we like to be neat).

2b ⟨*server close down* 2b⟩≡

```
close(Logfile)
```

Uses `Logfile` 2a 7a.
This code is used in chunk 1.

The logit function logs the message, using printf to avoid the CR-LF that would otherwise come with print. The obvious name, log, is already reserved in Awk for people who want to compute logarithms. Since the logit function supplies the newline, we have to remember not to put one in the messages we generate for logging.

2c ⟨*log function* 2c⟩≡

```
# logit --- send a message to the log file

function logit(message)
{
        printf("%s\n", message) > Logfile
        fflush(Logfile)
}
```

Defines:
`logit`, used in chunks 3b, 4, 7a, 10–14, 18–20, and 22a.
Uses `Logfile` 2a 7a.
This code is used in chunks 1 and 6b.

The server simply reads and writes standard input and output. We assume it will be started from the client via the `|&` mechanism. (This is a new feature of `gawk` 3.1 that opens a *two-way* pipeline between `gawk` and the given program. When used with special filenames that start with `/inet`, `gawk` opens a socket.)

Part of the mechanics is to remember to flush standard output after every write. For communication with this routine, the rest of the program just sets a global variable, `Mesg`, to have the message to send. Cheesy, but good enough in a program this small.

The call to `fflush` uses a `gawk` extension; with an argument that is the null string, `gawk` flushes all open output files and pipes.

3a ⟨*print and flush message* 3a⟩≡
```
print Mesg
fflush("")
```
Defines:
`Mesg`, used in chunks 3b, 4, 10, and 12.
This code is used in chunks 3b and 4.

Some familiarity with RFC 821 (which defines SMTP) would be helpful at this point.

When the server starts up, it prints out an initial greeting message. This can be done as part of the initialization. The domain `deva.com` doesn't really exist; one day I hope to register it for myself... We log both the server and client transactions, and prefix each one with `S:` and `C:` respectively.

3b ⟨*server init* 2a⟩+≡
```
# print initial message
Mesg = "220 skeeve.deva.com Fake SMTP server in gawk, how do you do?"
⟨print and flush message 3a⟩
logit("S: " Mesg)
```
Uses `Mesg` 3a and `logit` 2c.

Awk seems very nicely suited for writing a simple server like this. Just list the incoming commands and what to do when you get them.

4 ⟨*rules for SMTP commands* 4⟩≡
```
/HELO / {
        logit("C: " $0)
        Mesg = "250 nice to meet you " $2
        ⟨print and flush message 3a⟩
        logit("S: " Mesg)
        next
}

/MAIL / {
        logit("C: " $0)
        Sender = gensub(/MAIL *FROM:<(.+)>.*/, "\\1", 1, $0)
        logit("Server: extracted Sender ‘" Sender "’")
        Mesg = "250 Okey Dokey"
        ⟨print and flush message 3a⟩
        logit("S: " Mesg)
        next
}

/RCPT / {
        logit("C: " $0)
        Recipient = gensub(/RCPT *TO:<(.+)>.*/, "\\1", 1, $0)
        logit("Server: extracted Recipient ‘" Recipient "’")
        Mesg = "250 Okey Dokey"
        ⟨print and flush message 3a⟩
        logit("S: " Mesg)
        next
}

/DATA/ {
        logit("C: " $0)
        Mesg = "354 lemme have it"
        ⟨print and flush message 3a⟩
        logit("S: " Mesg)

        status = 1
        # read message data
        do {
                logit("C: " $0)
                if ($0 == ".")
                        break
                # A real SMTP server would de-escape the data here
        } while ((status = getline) > 0)
        if (status <= 0) {
                logit("Server: received EOF or error while reading DATA")
                exit 1
        }
```

```
            Mesg = "250 Okey Dokey"
            ⟨print and flush message 3a⟩
            logit("S: " Mesg)
            next
    }

    /QUIT/ {
            logit("C: " $0)
            Mesg = "221 skeeve.deva.com See ya next time!"
            ⟨print and flush message 3a⟩
            logit("S: " Mesg)
            exit 0
    }
```

Defines:
  `Recipient`, never used.
  `Sender`, used in chunk 19b.
Uses `Mesg` 3a, `Server` 7a, and `logit` 2c.
This code is used in chunk 1.

The rule for `/DATA/` ends up logging the `DATA` statement twice, but this minor glitch isn't worth twisting the logic around for in order to remove it. And that does it for the server! Not too bad.

Hmmm. The `Sender` and `Recipient` variables aren't actually used, but they're useful for logging purposes to make sure that the client has correctly abstracted all of the recipients and the sender for any given message.

(As an aside, the `noweb`'ed document indicates that `Sender` is used later on. This is perhaps a "weakness" in the way `noweb` assumes that there is only one program per file.)

Delivering Mail Now on to the main point of the program, which is the SMTP client.

The program is invoked this way:

5 ⟨*sample invocation* 5⟩≡

```
    sendout [-v Debug=1] /var/mailhack/outgoing/arnold
```

Uses `Debug` 2a 7a.
Root chunk (not used in this document).

If `Debug` is set to one on the command line, the client will invoke the debugging server and turn on its own debugging. Otherwise, it will open a real SMTP connection to the mail server. Note that the use of `-v` is required so that `Debug`'s value will be available inside the `BEGIN` rule.

The one required argument is the name of the file, in UNIX mailbox format, that contains the outgoing mail message(s).

The general outline of the program is as follows.

6a ⟨*sendout.awk* 6a⟩≡

```
#! /usr/local/bin/gawk-3.1.0 -f
BEGIN {
        # manifest constants
        TRUE = 1
        FALSE = 0

#       LINT = 1
        ⟨client init 7a⟩
}

⟨process mailbox file 16⟩

END {
        ⟨deliver messages 10⟩
        ⟨final logging 11⟩
        ⟨client close down 7b⟩
}

⟨auxiliary functions 6b⟩
```

Uses `FALSE` 6b, `LINT` 6b, and `TRUE` 6b.
Root chunk (not used in this document).

6b ⟨*auxiliary functions* 6b⟩≡

```
⟨log function 2c⟩
```

Defines:
`FALSE`, used in chunks 6a, 7a, and 18b.
`LINT`, used in chunk 6a.
`TRUE`, used in chunks 6a, 7a, and 16.
This definition is continued in chunks 13, 14b, 17b, 18c, and 22a.
This code is used in chunk 6a.

Note that we can re-use the `logit` function from the server, and that `noweb` makes this trivial to do. `:-)`

The `LINT` variable, when true, turns on warnings, as if `--lint` had been supplied on the command line. This is because I always forget to use that option.

Initialization sets up the name of the SMTP server and creates a client log file name for logging, if needed. If not debugging, we log to `/dev/null`. This simplifies the main code, we can just always do logging.

7a ⟨*client init* 7a⟩≡
```
if (Debug) {
        Server = "./testserv.awk"
        Logfile = ("/tmp/mh.log." PROCINFO["pid"])
        Myname = "ts008p4.rvt.netvision.net.il"
} else {
        # /inet/protocol/local-port/remote-host/remote-port
        # Server = "/inet/tcp/0/mailgw.netvision.net.il/smtp"
        Server = "/inet/tcp/0/mail.gezernet.co.il/smtp"
        Logfile = "/dev/null"
        "/var/mailhack/bin/getmyname" | getline Myname
        close("/var/mailhack/bin/getmyname")
}

# XXX --- temporarily, always log
Logfile = ("/tmp/mh.log." PROCINFO["pid"])

logit(sprintf("-- Debug = %s", Debug ? "TRUE" : "FALSE"))
logit(sprintf("-- Myname = %s", Myname))
logit(sprintf("-- Server = %s", Server))
```
Defines:
  `Debug`, used in chunks 5, 18a, and 19a.
  `Logfile`, used in chunks 2 and 7b.
  `Myname`, used in chunk 12.
  `Server`, used in chunks 4, 7b, 10, 12, and 13.
Uses `FALSE` 6b, `TRUE` 6b, and `logit` 2c.
This code is used in chunk 6a.

The server name is hardwired into the code, since this is just for my use. A more "production" oriented program would read such things from a configuration file, but if we want `sendmail`, we know where to find it, don't we?

We also have to close the log file when we're done. And while we're at it, we need to close the connection to the server.

7b ⟨*client close down* 7b⟩≡
```
close(Logfile)
close(Server)
```

Uses `Logfile` 2a 7a and `Server` 7a.
This code is used in chunk 6a.

A Small Detour — Determining Our Host Name Getting our actual hostname is a bit complicated. Because we're on a dialup connection, the PPP address changes every time. The address and corresponding name are available, but it takes a bit of work to get them.

The second line of the output of `ifconfig` contains the IP address in the second field, after a colon. Here is the result of `ifconfig ppp0`.

8a ⟨*output of* `ifconfig` 8a⟩≡

```
ppp0      Link encap:Point-Point Protocol
          inet addr:199.203.248.71  P-t-P:199.203.248.56  Mask:255.255.255.0
          UP POINTOPOINT RUNNING  MTU:1500  Metric:1
          RX packets:278 errors:0 dropped:0 overruns:0
          TX packets:266 errors:0 dropped:0 overruns:0
```

Root chunk (not used in this document).

Once you have an address, `dig` can be used to turn it into a hostname. Here is the result of `dig -x 199.203.248.71`.

8b ⟨*sample* `dig` *output* 8b⟩≡

```
; <<>> DiG 2.2 <<>> -x
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6
;; flags: qr aa rd ra; Ques: 1, Ans: 1, Auth: 2, Addit: 2
;; QUESTIONS:
;;      71.248.203.199.in-addr.arpa, type = ANY, class = IN

;; ANSWERS:
71.248.203.199.in-addr.arpa.    86400   PTR     ts003p15.rvt.netvision.net.il.

;; AUTHORITY RECORDS:
248.203.199.IN-ADDR.ARPA.       86400   NS      dns.netvision.net.il.
248.203.199.IN-ADDR.ARPA.       86400   NS      nypop.elron.net.

;; ADDITIONAL RECORDS:
dns.netvision.net.il.   86400   A       194.90.1.5
nypop.elron.net.        86400   A       199.203.1.20

;; Total query time: 673 msec
;; FROM: skeeve to SERVER: default -- 194.90.1.5
;; WHEN: Thu Nov  5 21:10:03 1998
;; MSG SIZE  sent: 45  rcvd: 191
```

Root chunk (not used in this document).

Here's the program to put the two together.

9 ⟨*getmyname.sh* 9⟩≡
```
#! /bin/sh
addr=$(/sbin/ifconfig ppp0 | sed -n 2p |
        awk '{ print $2 }' | sed 's/.*://')
dig -x $addr | grep . | grep -i PTR |
        sed 's/\.$//' | awk '{ print $NF }'
```

Root chunk (not used in this document).

The first line extracts the IP address and assigns it to `addr`. The second line looks up the IP address to get the hostname. The output of `dig` is actually in DNS resource record format, thus all the `grep`-ing, `sed`-ing, and `awk`-ing to extract the actual name.

Sending The Mail The design of our "data structures" (in Awk, global variables and associative arrays) will be driven by the delivery process, so we'll think about that first.

We will need the following:

1. A count of messages
2. A sender for each message;
3. A *list* of recipients for each message
4. The actual text (headers and body) of each message

Text transparency is also implemented at this point, although if we were really worried about efficiency when sending, we would probably do it when the messages are processed. Running with `LINT` enabled made me notice that the call to `substr` should be preceded by a check to make sure that the string is non-zero in length.

10 ⟨*deliver messages* 10⟩≡

```
⟨open SMTP connection 12⟩
for (Cur_mesg = 1; Cur_mesg <= Count; Cur_mesg++) {
        # sender
        Mesg = sprintf("MAIL FROM:<%s>", Sender[Cur_mesg])
        send_mesg_and_check_reply(Mesg, "250", "")

        # list of recipients
        num_recips = split(Recipients[Cur_mesg], destlist, " ")
        for (j = 1; j <= num_recips; j++) {
                Mesg = sprintf("RCPT TO:<%s>", destlist[j])
                send_mesg_and_check_reply(Mesg, "250", "")
        }

        # message text
        Mesg = "DATA"
        send_mesg_and_check_reply(Mesg, "354", "250")
        for (j = 1; (Cur_mesg, j) in Text; j++) {
                # handle transparency
                if (length(Text[Cur_mesg,j]) > 0 && substr(Text[Cur_mesg,j], 1, 1) == ".")
                        outline = ("." Text[Cur_mesg,j])
                else
                        outline = Text[Cur_mesg,j]
                print outline |& Server
                logit(outline)
        }
```

```
          Mesg = "."
          send_mesg_and_check_reply(Mesg, "250", "")

          # for logging, count of successfully sent messages
          Num_sent = Cur_mesg
  }
```

Defines:
`Count`, used in chunks 11, 14a, 17–20, and 22a.
`Cur_mesg`, used in chunk 14a.
`Num_sent`, used in chunk 11.
`Recipients`, used in chunks 18b and 22a.
`Sender`, used in chunk 19b.
`Text`, used in chunk 17c.
Uses `Mesg` 3a, `Server` 7a, `list` 19a, `logit` 2c, and `send_mesg_and_check_reply` 13.
This code is used in chunk 6a.

The data structures fall out of the above code. `Count` is the total number of outgoing messages. For any message `i`, `Sender[i]` is the sender of each message, `Recipients[i]` is the list of recipients of that message, and the text of the message is in `Text`.

The `Text` array is two dimensional. The first index represents the message number, and the second represents the line within the message. The use of the associative array test `(Cur_mesg, j) in Text` is an elegant way to loop through the text of the message without having to separately store a count of the number of lines in each message.

Initially, `Cur_mesg` was just a simple loop counter, `i`. However, it turns out to be useful to be able to know elsewhere in the program what the number of the current message is, mainly for error handling purposes.

When done, we should log how many messages were processed and sent.

11 ⟨*final logging* 11⟩≡

```
  logit(sprintf("-- Processed %d messages", Count))
  logit(sprintf("-- Sent %d messages", Num_sent))
```

Uses `Count` 10 17a, `Num_sent` 10, and `logit` 2c.
This code is used in chunk 6a.

Starting the Connection There is an initial transaction when first connecting to an SMTP server. The first one to say anything is actually the server, who responds to the SMTP connection with a greeting. Then the client sends a "Hello" message.

12 ⟨*open SMTP connection* 12⟩≡

```
RS = ORS = "\r\n"       # SMTP uses CR-LF to terminate lines
Server |& getline response
if (response !~ /^220/) {
        logit("-- Got bad initial connection mesg: " response)
        exit 1
} else
        logit("S: " response)
Mesg = "HELO " Myname
send_mesg_and_check_reply(Mesg, "250", "220")
```

Defines:
`response`, used in chunks 13 and 14a.
Uses `Mesg` 3a, `Myname` 7a, `Server` 7a, `logit` 2c, and `send_mesg_and_check_reply` 13.
This code is used in chunk 10.

Low Level Communications As in the server, the client uses the global variable `Mesg` for all communications. However, here, `Mesg` just gets passed as an argument to `send_mesg_and_check_reply`. But our interactions are more complicated, since for every command sent, we have to read a reply. We should also check that the reply is a good one, and do some kind of error recovery if not.

13 ⟨*auxiliary functions* 6b⟩+≡

```
# send_mesg_and_check_reply --- send a message and verify the response

function send_mesg_and_check_reply(text, expected, alt,         response, errmsg)
{
        print text |& Server
        logit("C: " text)
        # fflush(Server)

        if (alt == "") {
                Server |& getline response
                logit("S: " response)
        } else {
                do {
                        Server |& getline response
                        logit("S: " response)
                } while (response ~ ("^" alt))
        }

        if (response !~ ("^" expected)) {
                ⟨error handling 14a⟩
        }
}
```

Defines:
`send_mesg_and_check_reply`, used in chunks 10 and 12.
Uses `Server` 7a, `logit` 2c, and `response` 12.

It happens that `gawk` automatically flushes writes to two-way connections. We put in a commented-out call to `fflush` just to remind ourselves that in theory we should be doing it ourselves.

The `alt` response is for the case of some SMTP servers that send multiple "250" responses upon first connection. They may also send an additional "220" response before the "250."

Handling Errors The truth is that we don't expect there to be a lot of errors. SMTP servers tend to just accept data, and then process it later, sending back problems via email. But we ought to do some kind of error recovery, so we print a message to standard error indicating the message number, and what went wrong. We also print it to the log file and to a text file in a permanent location.

14a ⟨*error handling* 14a⟩≡
```
errmesg = sprintf("MAILHACK: stopped processing %s on message %d of %d, at %s",
        (1 in ARGV) ? ARGV[1] : "STDIN",        # text file name
        Cur_mesg, Count, strftime())
errmsg = errmsg sprintf(", expected '%s', got '%s'", expected, response)
fatal(errmsg)
```

Uses `Count` 10 17a, `Cur_mesg` 10, `fatal` 14b, and `response` 12.
This code is used in chunk 13.

14b ⟨*auxiliary functions* 6b⟩+≡
```
# fatal --- print error messages and die

function fatal( errmsg)
{
        logit(errmesg)
        printf("%s\n", errmsg) > "/dev/stderr"
        printf("%s\n", errmsg) >> "/var/mailhack/log/errlog"
        close("/var/mailhack/log/errlog")
        exit 1
}
```
Defines:
`fatal`, used in chunk 14a.
Uses `logit` 2c.

UNIX Mailbox Files Now that we've looked at the SMTP part of the program, let's handle reading the messages and storing them in the arrays we described earlier.

A description of the UNIX mailbox file format is in order at this point. Our program will rely on this format being already produced and ready.

A mailbox consists of a sequence of one or more messages. Messages in turn consist of a header and a body. Headers start with a so-called `From_` line, sometimes written 'From ' (note the space). It typically looks like this:

15a ⟨*sample* `From_` *line* 15a⟩≡

```
From carlie@ssc.com  Wed Sep 16 20:59:16 1998
```

Root chunk (not used in this document).

Note that it's very simple: the word `From`, a space, an address, and a rather generic date.

The rest of the header is in traditional RFC 822 format. For example:

15b ⟨*sample* `From:` *line* 15b⟩≡

```
From: Carlie Fairchild <carlie@ssc.com>
```

Root chunk (not used in this document).

There are other formats for `From:` lines, but in our case, the `From:` line will always look like this. (This is because separate software, that we control [not shown here], produces the `From:` line.)

The destination lines (`To:`, etc.) are more complicated, since there's the possibility that they may have multiple recipients, and/or be continued on the next line. Continued lines are indented from the line they continue with a tab. Addresses can also be complicated; this example shows the various forms they can take.

15c ⟨*sample* `To:` *line* 15c⟩≡

```
To: arnold@gnu.org, Miriam Robbins <mrobbins@netvision.net.il>,
        "Miriam Robbins" <miriam@skeeve.atl.ga.us>,
        arnold@uunet.uu.net (Arnold Robbins)
```

Root chunk (not used in this document).

The header ends with an absolutely blank line. The body starts on the next line and continues to the next `From_` line. There should be a blank line before the `From_` line that starts the next message, but our program will not need to rely on this. The separate software that creates the mailbox format file also handles escaping any occurences of "From" in the text.

There is one wrinkle with header processing. `Bcc:` ("blind carbon-copy") headers indicate people who should receive a copy of the message. Thus, addresses should be collected from a `Bcc:` line, but the header itself should *not* be passed into the message body.

With that introduction, we can start writing the code. First, we know that a new message starts with a `From_` line. So when we see one, we initialize any variables, and set a flag indicating that we're processing a message header. Upon seeing a blank line, we know the header has ended, and that the body continues until the next `From_` line is seen.

16 ⟨*process mailbox file* 16⟩≡

```
/^From /        {
        # new mailbox From_ line seen, start over
        ⟨initialize variables 17a⟩
        In_header = TRUE
        # don't save the From_ line
        next
}

/^$/ && In_header       {
        ⟨finish header processing 18b⟩

        save_line($0)
        next
}

⟨collect and process headers 18a⟩

# for body lines
! In_header {
        save_line($0)
}
```

Defines:
`In_header`, used in chunks 18 and 19a.
Uses `TRUE` 6b, `line` 19a, and `save_line` 17c.
This code is used in chunk 6a.

For variables, the main one to be reinitialized is the line counter for the lines in the message, and `Count` has to be incremented for each new message. We rely on the fact that Awk initializes `Count` to zero for us.

17a ⟨*initialize variables* 17a⟩≡

```
Count++
line_count = 1
```

Defines:
`Count`, used in chunks 11, 14a, 17–20, and 22a.
`line_count`, used in chunk 17c.
This definition is continued in chunk 17d.
This code is used in chunk 16.

Saving the line is a function, since it's done in multiple places. We also want call-by-value semantics. We used to rely on using `$0`, but that led to just all kinds of global-variable interference problems.

17b ⟨*auxiliary functions* 6b⟩+≡

```
⟨save_line function 17c⟩
```

17c ⟨`save_line` *function* 17c⟩≡

```
# save_line --- save a line in Text

function save_line(the_line)
{
        Text[Count, line_count] = the_line
        line_count++
}
```

Defines:
`save_line`, used in chunks 16 and 22b.
Uses `Count` 10 17a, `Text` 10, `line` 19a, and `line_count` 17a.
This code is used in chunk 17b.

Collecting the headers requires some thinking. An earlier version of this program tried to manually collect continuation lines and then separately handle the problem of dealing with any look-ahead lines. This leads to problems and duplicated logic. (Specifically, it didn't correctly handle a case where a `Bcc:` line followed a `To:` or `Cc:` line.)

So, we'll take a different approach of collecting lines one at a time, and handling them differently based on whether or not they are continuation lines. Once a non-continuation line is found, we can dispatch the previous collected header line.

For this, we'll need another variable: `Current_header` which will have the entire header after continuation lines have been added to it.

17d ⟨*initialize variables* 17a⟩+≡

```
Current_header = ""
```

Defines:
`Current_header`, used in chunk 18.

Because of the ordering of the rules, if `In_header` is `TRUE`, we know that the line is not empty. Otherwise, the rule that checks for an empty line would have reset `In_header`.

18a ⟨*collect and process headers* 18a⟩≡
```
In_header {
        if (Debug)
                logit(sprintf("-- header line, In_header = %d, input = <%s>",
                        In_header, $0))

        if (/^[ \t]/) {
                # it's a continuation line
                # join with \n so can resplit, later
                Current_header = Current_header "\n" $0
                next
        }

        if (Current_header != "")
                dispatch(Current_header)

        Current_header = $0
        next
}
```

Uses `Current_header` 17d, `Debug` 2a 7a, `In_header` 16, `dispatch` 19a, `line` 19a, and `logit` 2c.
This code is used in chunk 16.

When the header is finished, the last saved header has to be dispatched.

18b ⟨*finish header processing* 18b⟩≡
```
# dispatch last header
if (Current_header != "") {
        dispatch(Current_header)
}

In_header = FALSE
logit(sprintf("-- Final recipient list <%s>", Recipients[Count]))
```

Uses `Count` 10 17a, `Current_header` 17d, `FALSE` 6b, `In_header` 16, `Recipients` 10, `dispatch` 19a, `list` 19a, and `logit` 2c.
This code is used in chunk 16.

Dispatching the header perfoms an action based on what kind of header it is. `From:` lines are straight-forward. The real work is in the `To:`, `Cc:` and `Bcc:` lines.

18c ⟨*auxiliary functions* 6b⟩+≡
⟨`dispatch` *function* 19a⟩

19a ⟨dispatch *function* 19a⟩≡

```
# dispatch --- handle a header line

function dispatch(line,         list, n, i, textlines)
{
        if (Debug) {
                logit(sprintf("-- dispatch: In_header = %d", In_header))
                logit(sprintf("-- dispatch: dispatching <%s>", line))
        }

        # copy line because we may have to change it
        list = line

        if (line ~ /^From: /) {
                ⟨extract sender 19b⟩
        } else if (line ~ /^(To|Cc|Bcc): /) {
                ⟨get recipient list 19c⟩
        }
        if (line !~ /^Bcc: /) {
                ⟨resplit line and add to Text 22b⟩
        }
}
```

Defines:
- `dispatch`, used in chunk 18.
- `line`, used in chunks 16–18 and 22b.
- `list`, used in chunks 10, 18–20, and 22a.
- `textlines`, used in chunk 22b.

Uses `Debug` 2a 7a, `In_header` 16, and `logit` 2c.
This code is used in chunk 18c.

19b ⟨*extract sender* 19b⟩≡

```
Sender[Count] = gensub(/.*<(.+)>.*/, "\\1", 1, list)
logit(sprintf("-- Sender[%d] = <%s>", Count, Sender[Count]))
```

Uses `Count` 10 17a, `Sender` 4 10, `list` 19a, and `logit` 2c.
This code is used in chunk 19a.

Extracting the recipients is non-trivial, since there can be a fair amount of "gunk" (for our purposes) surrounding the actual email address that we want.

19c ⟨*get recipient list* 19c⟩≡

```
# remove leading header field
sub(/^(To|Cc|Bcc):[ \t]+/, "", list)
# turn newlines into spaces
gsub(/\n/, " ", list)
⟨split into individual addresses 20⟩
⟨get email addresses 21⟩
```

Uses `list` 19a.
This code is used in chunk 19a.

Multiple email addresses are separated by commas with spaces. As I've occasionally seen semi-colons used too, we'll use a fairly liberal regular expression to split the addresses apart.

20 ⟨*split into individual addresses* 20⟩≡
```
logit(sprintf("-- Splitting up List[%d] <%s>", Count, list))
num_addresses = split(list, to_address, /[ \t]*[;,][ \t]*/)
# should check that num_addresses > 0
```
Defines:
`num_addresses`, used in chunk 21.
`to_address`, used in chunk 21.
Uses `Count` 10 17a, `list` 19a, and `logit` 2c.
This code is used in chunk 19c.

As we saw in the sample `To:` line above, email addresses in headers can have several forms. We have to further split each address apart and try to guess which form it's in.

21 ⟨*get email addresses* 21⟩≡
```
for (i = 1; i <= num_addresses; i++) {
        if (to_address[i] !~ /[<>()"]/ && index(to_address[i], "@") != 0) {
                # it's a simple address:        arnold@gnu.org
                add_recip_address(to_address[i])
                continue
        }

        # at this point, there's more work to do

        nfields = split(to_address[i], hfields, " ")

        # Miriam Robbins <mrobbins@netvision.net.il>,
        #       or
        # "Miriam Robbins" <miriam@skeeve.atl.ga.us>,

        if (index(to_address[i], "<") > 0) {
                for (j = 1; j <= nfields; j++) {
                        if (substr(hfields[j], 1, 1) == "<") {
                                gsub(/[<>]/, "", hfields[j])
                                add_recip_address(hfields[j])
                                break
                        }
                }
        } else {
                # arnold@uunet.uu.net (Arnold Robbins)
                for (j = 1; j <= nfields; j++) {
                        if (index(hfields[j], "@") != 0) {
                                add_recip_address(hfields[j])
                                break
                        }
                }
        }
}
```
Defines:
`hfields`, never used.
`nfields`, never used.
Uses `add_recip_address` 22a, `num_addresses` 20, and `to_address` 20.
This code is used in chunk 19c.

22a ⟨*auxiliary functions* 6b⟩+≡
```
# add_recip_address --- add an address to the list of recipients

function add_recip_address(     address)
{
        logit(sprintf("-- Message %d: adding recipient %s", Count, address))
        if (Recipients[Count] == "")
                Recipients[Count] = address
        else
                Recipients[Count] = Recipients[Count] " " address
}
```
Defines:
`add_recip_address`, used in chunk 21.
Uses `Count` 10 17a, `Recipients` 10, `list` 19a, and `logit` 2c.

After all the header processing, the actual header text has to be added back to the text of the message that will be sent. This will have been checked earlier to only be called if not a `Bcc:` line.

22b ⟨*resplit* `line` *and add to* `Text` 22b⟩≡
```
# note, this uses the original variable 'lines'
n = split(line, textlines, "\n")
for (i = 1; i <= n; i++) {
        save_line(textlines[i])
}
```
Uses `line` 19a, `save_line` 17c, and `textlines` 19a.
This code is used in chunk 19a.

A Makefile No program is complete without a `Makefile` for using `make`. By default, just the programs are remade.

23 ⟨*Makefile* 23⟩≡

```
# Makefile for SMTP awk client written in noweb
# This has to be initially bootstrapped, but no big deal

PROGS = testserv.awk sendout.awk
DOCS = sendout.tex
SOURCE = sendout.nw

.SUFFIXES:
.SUFFIXES: .nw .tex .awk

# I dunno why, but I can't get implicit rules to work for the
# awk programs, so they're done by hand. Sheesh.

.nw.tex:
	noweave -delay -index $< | cpif $@

# default target is the programs
programs: $(PROGS)

$(PROGS): $(SOURCE)

testserv.awk: $(SOURCE)
	notangle -R$@ $< | cpif $@

sendout.awk: $(SOURCE)
	notangle -R$@ $< | cpif $@

Makefile: $(SOURCE)
	notangle -t8 -R$@ $< | cpif $@

$(DOCS): $(SOURCE)

sendout.dvi: $(DOCS)
	latex $(DOCS)   # twice to get xrefs right
	latex $(DOCS)

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

all: $(PROGS) $(DOCS) ps

ps: sendout.ps

dvi: sendout.dvi

clean:
```

```
-rm -f *.ps *.dvi *.awk *.tex *.aux *.dvi *.log
```

Root chunk (not used in this document).

Defined Chunks

Index