Taylor UUCP

Version 1.06

Ian Lance Taylor <ian@airs.com>

Published by Ian Lance Taylor <ian@airs.com>.

Taylor UUCP 1.06

This is the documentation for the Taylor UUCP package, version 1.06. The programs were written by Ian Lance Taylor. The author can be reached at `<ian@airs.com>`, or at

Ian Lance Taylor
c/o Cygnus Support
48 Grove Street
Somerville, MA 02144
USA

There is a mailing list for discussion of the package. The list is hosted by Eric Schnoebelen at `cirr.com`. To join (or get off) the list, send mail to `taylor-uucp-request@cirr.com`. Mail to this address is answered by the majordomo program. To join the list, send the message '`subscribe` *address*' where *address* is your e-mail address. To send a message to the list, send it to `taylor-uucp@cirr.com`. The old list address, `taylor-uucp@gnu.ai.mit.edu`, will also work. There is an archive of all messages sent to the mailing list at `ftp.cirr.com`.

Taylor UUCP Copying Conditions

This package is covered by the GNU Public License. See the file 'COPYING' for details. If you would like to do something with this package that you feel is reasonable, but you feel is prohibited by the license, contact me to see if we can work it out.

The rest of this section is some descriptive text from the Free Software Foundation.

All the programs, scripts and documents relating to Taylor UUCP are *free*; this means that everyone is free to use them and free to redistribute them on a free basis. The Taylor UUCP-related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to Taylor UUCP, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the Taylor UUCP related programs, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to Taylor UUCP. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to Taylor UUCP are found in the General Public Licenses that accompany them.

1 Introduction to Taylor UUCP

General introductions to UUCP are available, and perhaps one day I will write one. In the meantime, here is a very brief one that concentrates on the programs provided by Taylor UUCP.

Taylor UUCP is a complete UUCP package. It is covered by the GNU Public License, which means that the source code is always available. It is composed of several programs; most of the names of these programs are based on earlier UUCP packages.

`uucp`

: The `uucp` program is used to copy file between systems. It is similar to the standard Unix `cp` program, except that you can refer to a file on a remote system by using '`system!`' before the file name. For example, to copy the file '`notes.txt`' to the system '`airs`', you would say '`uucp notes.txt airs!~/notes.txt`'. In this example '`~`' is used to name the UUCP public directory on '`airs`'. For more details, see Section 2.2 [Invoking uucp], page 10.

`uux`

: The `uux` program is used to request the execution of a program on a remote system. This is how mail and news are transferred over UUCP. As with `uucp`, programs and files on remote systems may be named by using '`system!`'. For example, to run the `rnews` program on '`airs`', passing it standard input, you would say '`uux - airs!rnews`'. The '`-`' means to read standard input and set things up such that when `rnews` runs on '`airs`' it will receive the same standard input. For more details, see Section 2.3 [Invoking uux], page 12.

Neither `uucp` nor `uux` actually do any work immediately. Instead, they queue up requests for later processing. They then start a daemon process which processes the requests and calls up the appropriate systems. Normally the system will also start the daemon periodically to check if there is any work to be done. The advantage of this approach is that it all happens automatically. You don't have to sit around waiting for the files to be transferred. The disadvantage is that if anything goes wrong it might be a while before anybody notices.

`uustat`

: The `uustat` program does many things. By default it will simply list all the jobs you have queued with `uucp` or `uux` that have not yet been processed. You can use `uustat` to remove any of your jobs from the queue. You can also it use it to show the status of the UUCP system in various ways, such as showing the connection status of all the remote systems your system knows about. The system administrator can use `uustat` to automatically discard old jobs while sending mail to the user who requested them. For more details, see Section 2.4 [Invoking uustat], page 16.

`uuname`

The `uuname` program by default lists all the remote systems your system knows about. You can also use it to get the name of your local system. It is mostly useful for shell scripts. For more details, see Section 2.5 [Invoking uuname], page 22.

`uulog`

The `uulog` program can be used to display entries in the UUCP log file. It can select the entries for a particular system or a particular user. You can use it to see what has happened to your queued jobs in the past. For more details, see Section 2.6 [Invoking uulog], page 23.

`uuto`

`uupick`

`uuto` is a simple shell script interface to `uucp`. It will transfer a file, or the contents of a directory, to a remote system, and notify a particular user on the remote system when it arrives. The remote user can then retrieve the file(s) with `uupick`. For more details, see Section 2.7 [Invoking uuto], page 24, and see Section 2.8 [Invoking uupick], page 25.

`cu`

The `cu` program can be used to call up another system and communicate with it as though you were directly connected. It can also do simple file transfers, though it does not provide any error checking. For more details, Section 2.9 [Invoking cu], page 26.

These eight programs just described, `uucp`, `uux`, `uuto`, `uupick`, `uustat`, `uuname`, `uulog`, and `cu` are the user programs provided by Taylor UUCP. `uucp`, `uux`, and `uuto` add requests to the work queue, `uupick` extracts files from the UUCP public directory, `uustat` examines the work queue, `uuname` examines the configuration files, `uulog` examines the log files, and `cu` just uses the UUCP configuration files.

The real work is actually done by two daemon processes, which are normally run automatically rather than by a user.

`uucico`

The `uucico` daemon is the program which actually calls the remote system and transfers files and requests. `uucico` is normally started automatically by `uucp` and `uux`.

Most systems will also start it periodically to make sure that all work requests are handled. uucico checks the queue to see what work needs to be done, and then calls the appropriate systems. If the call fails, perhaps because the phone line is busy, uucico leaves the requests in the queue and goes on to the next system to call. It is also possible to force uucico to call a remote system even if there is no work to be done for it, so that it can pick up any work that may be queued up remotely. For more details, see Section 2.10 [Invoking uucico], page 30.

`uuxqt`

The uuxqt daemon processes execution requests made by the uux program on remote systems. It also processes requests made on the local system which require files from a remote system. It is normally started by uucico. For more details, see Section 2.11 [Invoking uuxqt], page 33.

Suppose you, on the system 'bantam', want to copy a file to the system 'airs'. You would run the uucp command locally, with a command like 'uucp notes.txt airs!~/notes.txt'. This would queue up a request on 'bantam' for 'airs', and would then start the uucico daemon. uucico would see that there was a request for 'airs' and attempt to call it. When the call succeeded, another copy of uucico would be started on 'airs'. The two copies of uucico would tell each other what they had to do and transfer the file from 'bantam' to 'airs'. When the file transfer was complete the uucico on 'airs' would move it into the UUCP public directory.

UUCP is often used to transfer mail. This is normally done automatically by mailer programs. When 'bantam' has a mail message to send to 'ian' at 'airs', it executes 'uux - airs!rmail ian' and writes the mail message to the uux process as standard input. The uux program, running on 'bantam', will read the standard input and store it, as well as the rmail request itself, on the work queue for 'airs'. uux will then start the uucico daemon. The uucico daemon will call up 'airs', just as in the uucp example, and transfer the work request and the mail message. The uucico daemon on 'airs' will put the files on a local work queue. When the communication session is over, the uucico daemon on 'airs' will start the uuxqt daemon. uuxqt will see the request on the work queue, and will run 'rmail ian' with the mail message as standard input. The rmail program, which is not part of the UUCP package, is then responsible for either putting the message in the right mailbox on 'airs' or forwarding the message on to another system.

Taylor UUCP comes with a few other programs that are useful when installing and configuring UUCP.

`uuchk`

The uuchk program reads the UUCP configuration files and displays a rather lengthy description of what it finds. This is useful when configuring UUCP to make certain that the UUCP package will do what you expect it to do. For more details, see Section 2.12 [Invoking uuchk], page 34.

`uuconv`

The `uuconv` program can be used to convert UUCP configuration files from one format to another. This can be useful for administrators converting from an older UUCP package. Taylor UUCP is able to read and use old configuration file formats, but some new features can not be selected using the old formats. For more details, see Section 2.13 [Invoking uuconv], page 35.

`uusched`

The `uusched` script is provided for compatibility with older UUCP releases. It starts `uucico` to call, one at a time, all the systems for which work has been queued. For more details, see Section 2.14 [Invoking uusched], page 36.

`tstuu`

The `tstuu` program is a test harness for the UUCP package; it can help check that the package has been configured and compiled correctly. However, it uses pseudo-terminals, which means that it is less portable than the rest of the package. If it works, it can be useful when initially installing Taylor UUCP. For more details, see Section 3.2 [Testing the Compilation], page 39.

2 Invoking the UUCP Programs

This chapter describes how to run the UUCP programs.

2.1 Standard Options

All of the UUCP programs support a few standard options.

'-x type'
'--debug type'
: Turn on particular debugging types. The following types are recognized: 'abnormal', 'chat', 'handshake', 'uucp-proto', 'proto', 'port', 'config', 'spooldir', 'execute', 'incoming', 'outgoing'. Not all types of debugging are effective for all programs. See the debug configuration command for details (see Section 5.6.4 [Debugging Levels], page 70).

  Multiple types may be given, separated by commas, and the '--debug' option may appear multiple times. A number may also be given, which will turn on that many types from the foregoing list; for example, '--debug 2' is equivalent to '--debug abnormal,chat'. To turn on all types of debugging, use '-x all'.

  The uulog program uses '-X' rather than '-x' to select the debugging type; for uulog, '-x' has a different meaning, for reasons of historical compatibility.

'-I file'
'--config file'
: Set the main configuration file to use. See Section 5.6 [config File], page 65. When this option is used, the programs will revoke any setuid privileges.

'-v'
'--version'
: Report version information and exit.

'--help'
: Print a help message and exit.

2.2 Invoking uucp

2.2.1 uucp Description

```
uucp [options] ‘source-file’ ‘destination-file’
uucp [options] ‘source-file’... ‘destination-directory’
```

The `uucp` command copies files between systems. Each ‘`file`’ argument is either a file name on the local machine or is of the form ‘`system!file`’. The latter is interpreted as being on a remote system.

When `uucp` is used with two non-option arguments, the contents of the first file are copied to the second. With more than two non-option arguments, each source file is copied into the destination directory.

A file may be transferred to or from ‘`system2`’ via ‘`system1`’ by using ‘`system1!system2!file`’.

Any file name that does not begin with ‘`/`’ or ‘`~`’ will be prepended with the current directory (unless the ‘`-W`’ or ‘`--noexpand`’ options are used). For example, if you are in the directory ‘`/home/ian`’, then ‘`uucp foo remote!bar`’ is equivalent to ‘`uucp /home/ian/foo remote!/home/ian/bar`’. Note that the resulting file name may not be valid on a remote system.

A file name beginning with a simple ‘`~`’ starts at the UUCP public directory; a file name beginning with ‘`~name`’ starts at the home directory of the named user. The ‘`~`’ is interpreted on the appropriate system. Note that some shells will interpret an initial ‘`~`’ before `uucp` sees it; to avoid this the ‘`~`’ must be quoted.

The shell metacharacters ‘`?`’ ‘`*`’ ‘`[`’ and ‘`]`’ are interpreted on the appropriate system, assuming they are quoted to prevent the shell from interpreting them first.

The file copy does not take place immediately, but is queued up for the `uucico` daemon; the daemon is started immediately unless the ‘`-r`’ or ‘`--nouucico`’ option is given. The next time the remote system is called, the file(s) will be copied. See Section 2.10 [Invoking uucico], page 30.

The file mode is not preserved, except for the execute bit. The resulting file is owned by the uucp user.

2.2.2 uucp Options

The following options may be given to `uucp`.

‘`-c`’
‘`--nocopy`’
: Do not copy local source files to the spool directory. If they are removed before being processed by the `uucico` daemon, the copy will fail. The files must be readable by the `uucico` daemon, and by the invoking user.

‘`-C`’
‘`--copy`’
: Copy local source files to the spool directory. This is the default.

‘`-d`’
‘`--directories`’
: Create all necessary directories when doing the copy. This is the default.

‘`-f`’
‘`--nodirectories`’
: If any necessary directories do not exist for the destination file name, abort the copy.

‘`-R`’
‘`--recursive`’
: If any of the source file names are directories, copy their contents recursively to the destination (which must itself be a directory).

‘`-g grade`’
‘`--grade grade`’
: Set the grade of the file transfer command. Jobs of a higher grade are executed first. Grades run `0` to `9`, `A` to `Z`, `a` to `z`, from high to low. See Section 5.7.3.1 [When to Call], page 74.

‘`-m`’
‘`--mail`’
: Report completion or failure of the file transfer by sending mail.

‘`-n user`’
‘`--notify user`’
: Report completion or failure of the file transfer by sending mail to the named user on the destination system.

'-r'
'--nouucico'
: Do not start the `uucico` daemon immediately; merely queue up the file transfer for later execution.

'-j'
'--jobid'
: Print the jobid on standard output. The job may be later cancelled by passing this jobid to the '-kill' switch of `uustat`. See Section 2.4 [Invoking uustat], page 16.

  It is possible for some complex operations to produce more than one jobid, in which case each will be printed on a separate line. For example

```
uucp sys1!~user1/file1 sys2!~user2/file2 ~user3
```

  will generate two separate jobs, one for the system 'sys1' and one for the system 'sys2'.

'-W'
'--noexpand'
: Do not prepend remote relative file names with the current directory.

'-t'
'--uuto'
: This option is used by the `uuto` shell script; see Section 2.7 [Invoking uuto], page 24. It causes `uucp` to interpret the final argument as 'system!user'. The file(s) are sent to '~/receive/*user*/*local*' on the remote system, where *user* is from the final argument and *local* is the local UUCP system name. Also, `uucp` will act as though '--notify user' were specified.

'-x type'
'--debug type'
'-I file'
'--config file'
'-v'
'--version'
'--help'
: See Section 2.1 [Standard Options], page 9.

2.3 Invoking uux

2.3.1 uux Description

```
uux [options] command
```

The `uux` command is used to execute a command on a remote system, or to execute a command on the local system using files from remote systems. The command is not executed immediately;

the request is queued until the `uucico` daemon calls the system and transfers the necessary files. The daemon is started automatically unless one of the '`-r`' or '`--nouucico`' options is given.

The actual command execution is done by the `uuxqt` daemon on the appropriate system.

File arguments can be gathered from remote systems to the execution system, as can standard input. Standard output may be directed to a file on a remote system.

The command name may be preceded by a system name followed by an exclamation point if it is to be executed on a remote system. An empty system name is taken as the local system.

Each argument that contains an exclamation point is treated as naming a file. The system which the file is on is before the exclamation point, and the file name on that system follows it. An empty system name is taken as the local system; this form must be used to transfer a file to a command being executed on a remote system. If the file name is not absolute, the current working directory will be prepended to it; the result may not be meaningful on the remote system. A file name may begin with '`~/`', in which case it is relative to the UUCP public directory on the appropriate system. A file name may begin with '`~name/`', in which case it is relative to the home directory of the named user on the appropriate system.

Standard input and output may be redirected as usual; the file names used may contain exclamation points to indicate that they are on remote systems. Note that the redirection characters must be quoted so that they are passed to `uux` rather than interpreted by the shell. Append redirection ('`>>`') does not work.

All specified files are gathered together into a single directory before execution of the command begins. This means that each file must have a distinct name. For example,

```
uux 'sys1!diff sys2!~user1/foo sys3!~user2/foo >!foo.diff'
```

will fail because both files will be copied to '`sys1`' and stored under the name '`foo`'.

Arguments may be quoted by parentheses to avoid interpretation of exclamation points. This is useful when executing the `uucp` command on a remote system.

Most systems restrict the commands which may be executed using '`uux`'. Many permit only the execution of '`rmail`' and '`rnews`'.

A request to execute an empty command (e.g., 'uux sys!') will create a poll file for the specified system; see Section 4.1 [Calling Other Systems], page 43 for an example of why this might be useful.

2.3.2 uux Options

The following options may be given to uux.

'-'
'-p'
'--stdin'
: Read standard input up to end of file, and use it as the standard input for the command to be executed.

'-c'
'--nocopy'
: Do not copy local files to the spool directory. This is the default. If they are removed before being processed by the uucico daemon, the copy will fail. The files must be readable by the uucico daemon, as well as the by the invoker of uux.

'-C'
'--copy'
: Copy local files to the spool directory.

'-l'
'--link'
: Link local files into the spool directory. If a file can not be linked because it is on a different device, it will be copied unless one of the '-c' or '--nocopy' options also appears (in other words, use of '--link' switches the default from '--nocopy' to '--copy'). If the files are changed before being processed by the uucico daemon, the changed versions will be used. The files must be readable by the uucico daemon, as well as by the invoker of uux.

'-g grade'
'--grade grade'
: Set the grade of the file transfer command. Jobs of a higher grade are executed first. Grades run 0 to 9, A to Z, a to z, from high to low. See Section 5.7.3.1 [When to Call], page 74.

'-n'
'--notification=no'
: Do not send mail about the status of the job, even if it fails.

'-z'
'--notification=error'
: Send mail about the status of the job if an error occurs. For many uuxqt daemons, including the Taylor UUCP uuxqt, this is the default action; for those,

'--notification=error' will have no effect. However, some uuxqt daemons will send mail if the job succeeds, unless the '--notification=error' option is used. Some other uuxqt daemons will not send mail even if the job fails, unless the '--notification=error' option is used.

'-a address'
'--requestor address'
: Report job status, as controlled by the '--notification' option, to the specified mail address.

'-r'
'--nouucico'
: Do not start the uucico daemon immediately; merely queue up the execution request for later processing.

'-j'
'--jobid'
: Print the jobid on standard output. A jobid will be generated for each file copy operation required to execute the command. These file copies may be later cancelled by passing the jobid to the '-kill' switch of uustat. See Section 2.4 [Invoking uustat], page 16. Cancelling any file copies will make it impossible to complete execution of the job.

'-x type'
'--debug type'
'-v'
'--version'
'--help'
: See Section 2.1 [Standard Options], page 9.

2.3.3 uux Examples

Here are some examples of using uux.

```
uux -z - sys1!rmail user1
```

This will execute the command 'rmail user1' on the system 'sys1', giving it as standard input whatever is given to uux as standard input. If a failure occurs, mail will be sent to the user who ran the command.

```
uux 'diff -c sys1!~user1/file1 sys2!~user2/file2 >!file.diff'
```

This will fetch the two named files from system 'sys1' and system 'sys2' and execute 'diff', putting the result in 'file.diff' in the current directory on the local system. The current directory must be writable by the uuxqt daemon for this to work.

```
uux 'sys1!uucp ~user1/file1 (sys2!~user2/file2)'
```

Execute uucp on the system 'sys1' copying 'file1' (on system 'sys1') to 'sys2'. This illustrates the use of parentheses for quoting.

2.4 Invoking uustat

2.4.1 uustat Description

```
uustat -a
uustat --all
uustat [-eKRiMNQ] [-sS system] [-uU user] [-cC command] [-oy hours]
       [-B lines] [--executions] [--kill-all] [--rejuvenate-all]
       [--prompt] [--mail] [--notify] [--no-list] [--system system]
       [--not-system system] [--user user] [--not-user user]
       [--command command] [--not-command command] [--older-than hours]
       [--younger-than hours] [--mail-lines lines]
uustat [-kr jobid] [--kill jobid] [--rejuvenate jobid]
uustat -q [-sS system] [-oy hours] [--system system]
       [--not-system system ] [--older-than hours] [--younger-than hours]
uustat --list [-sS system] [-oy hours] [--system system ]
       [--not-system system] [--older-than hours] [--younger-than hours]
uustat -m
uustat --status
uustat -p
uustat --ps
```

The uustat command can display various types of status information about the UUCP system. It can also be used to cancel or rejuvenate requests made by uucp or uux.

With no options, uustat displays all jobs queued up for the invoking user, as if given the '--user' option with the appropriate argument.

If any of the '-a', '--all', '-e', '--executions', '-s', '--system', '-S', '--not-system', '-u', '--user', '-U', '--not-user', '-c', '--command', '-C', '--not-command', '-o', '--older-than', '-y', or '--younger-than' options are given, then all jobs which match the combined specifications are displayed.

The '-K' or '--kill-all' option may be used to kill off a selected group of jobs, such as all jobs more than 7 days old.

2.4.2 uustat Options

The following options may be given to uustat.

'-a'
'--all' List all queued file transfer requests.

'-e'
'--executions'
List queued execution requests rather than queued file transfer requests. Queued execution requests are processed by uuxqt rather than uucico. Queued execution requests may be waiting for some file to be transferred from a remote system. They are created by an invocation of uux.

'-s system'
'--system system'
List all jobs queued up for the named system. These options may be specified multiple times, in which case all jobs for all the named systems will be listed. If used with '--list', only the systems named will be listed.

'-S system'
'--not-system system'
List all jobs queued for systems other than the one named. These options may be specified multiple times, in which case no jobs from any of the specified systems will be listed. If used with '--list', only the systems not named will be listed. These options may not be used with '-s' or '--system'.

'-u user'
'--user user'
List all jobs queued up for the named user. These options may be specified multiple times, in which case all jobs for all the named users will be listed.

'-U user'

'--not-user user'
: List all jobs queued up for users other than the one named. These options may be specified multiple times, in which case no jobs from any of the specified users will be listed. These options may not be used with '-u' or '--user'.

'-c command'
'--command command'
: List all jobs requesting the execution of the named command. If 'command' is 'ALL' this will list all jobs requesting the execution of some command (as opposed to simply requesting a file transfer). These options may be specified multiple times, in which case all jobs requesting any of the commands will be listed.

'-C command'
'--not-command command'
: List all jobs requesting execution of some command other than the named command, or, if 'command' is 'ALL', list all jobs that simply request a file transfer (as opposed to requesting the execution of some command). These options may be specified multiple times, in which case no job requesting one of the specified commands will be listed. These options may not be used with '-c' or '--command'.

'-o hours'
'--older-than hours'
: List all queued jobs older than the given number of hours. If used with '--list', only systems whose oldest job is older than the given number of hours will be listed.

'-y hours'
'--younger-than hours'
: List all queued jobs younger than the given number of hours. If used with '--list', only systems whose oldest job is younger than the given number of hours will be listed.

'-k jobid'
'--kill jobid'
: Kill the named job. The job id is shown by the default output format, as well as by the '-j' or '--jobid' options to `uucp` or `uux`. A job may only be killed by the user who created the job, or by the UUCP administrator, or the superuser. The '-k' or '--kill' options may be used multiple times on the command line to kill several jobs.

'-r jobid'
'--rejuvenate jobid'
: Rejuvenate the named job. This will mark it as having been invoked at the current time, affecting the output of the '-o', '--older-than', '-y', or '--younger-than' options, possibly preserving it from any automated cleanup daemon. The job id is shown by the default output format, as well as by the '-j' or '--jobid' options to `uucp` or `uux`. A job may only be rejuvenated by the user who created the job, or by the UUCP

administrator, or the superuser. The '-r' or '--rejuvenate' options may be used multiple times on the command line to rejuvenate several jobs.

'-q'
'--list' Display the status of commands, executions and conversations for all remote systems for which commands or executions are queued. The '-s', '--system', '-S', '--not-system', '-o', '--older-than', '-y', and '--younger-than' options may be used to restrict the systems which are listed. Systems for which no commands or executions are queued will never be listed.

'-m'
'--status'
Display the status of conversations for all remote systems.

'-p'
'--ps' Display the status of all processes holding UUCP locks on systems or ports.

'-i'
'--prompt'
For each listed job, prompt whether to kill the job or not. If the first character of the input line is `y` or `Y`, the job will be killed.

'-K'
'--kill-all'
Automatically kill each listed job. This can be useful for automatic cleanup scripts, in conjunction with the '--mail' and '--notify' options.

'-R'
'--rejuvenate-all'
Automatically rejuvenate each listed job. This may not be used with '--kill-all'.

'-M'
'--mail' For each listed job, send mail to the UUCP administrator. If the job is killed (due to '--kill-all', or '--prompt' with an affirmative response) the mail will indicate that. A comment specified by the '--comment' option may be included. If the job is an execution, the initial portion of its standard input will be included in the mail message; the number of lines to include may be set with the '--mail-lines' option (the default is 100). If the standard input contains null characters, it is assumed to be a binary file and is not included.

'-N'
'--notify'
For each listed job, send mail to the user who requested the job. The mail is identical to that sent by the '-M' or '--mail' options.

'-W comment'
'--comment comment'
: Specify a comment to be included in mail sent with the '-M', '--mail', '-N', or '--notify' options.

'-B lines'
'--mail-lines lines'
: When the '-M', '--mail', '-N', or '--notify' options are used to send mail about an execution with standard input, this option controls the number of lines of standard input to include in the message. The default is 100.

'-Q'
'--no-list'
: Do not actually list the job, but only take any actions indicated by the '-i', '--prompt', '-K', '--kill-all', '-M', '--mail', '-N' or '--notify' options.

'-x type'
'--debug type'
'-I file'
'--config file'
'-v'
'--version'
'--help' See Section 2.1 [Standard Options], page 9.

2.4.3 uustat Examples

```
uustat --all
```

Display status of all jobs. A sample output line is as follows:

```
bugsA027h bugs ian 04-01 13:50 Executing rmail ian@airs.com (sending 12 bytes)
```

The format is

```
jobid system user queue-date command (size)
```

The jobid may be passed to the '--kill' or '--rejuvenate' options. The size indicates how much data is to be transferred to the remote system, and is absent for a file receive request. The '--system', '--not-system', '--user', '--not-user', '--command', '--not-command', '--older-than', and '--younger-than' options may be used to control which jobs are listed.

```
uustat --executions
```

Display status of queued up execution requests. A sample output line is as follows:

```
bugs bugs!ian 05-20 12:51 rmail ian
```

The format is

```
system requestor queue-date command
```

The '`--system`', '`--not-system`', '`--user`', '`--not-user`', '`--command`', '`--not-command`', '`--older-than`', and '`--younger-than`' options may be used to control which requests are listed.

```
uustat --list
```

Display status for all systems with queued up commands. A sample output line is as follows:

```
bugs            4C (1 hour)   0X (0 secs) 04-01 14:45 Dial failed
```

This indicates the system, the number of queued commands, the age of the oldest queued command, the number of queued local executions, the age of the oldest queued execution, the date of the last conversation, and the status of that conversation.

```
uustat --status
```

Display conversation status for all remote systems. A sample output line is as follows:

```
bugs           04-01 15:51 Conversation complete
```

This indicates the system, the date of the last conversation, and the status of that conversation. If the last conversation failed, `uustat` will indicate how many attempts have been made to call the system. If the retry period is currently preventing calls to that system, `uustat` also displays the time when the next call will be permitted.

```
uustat --ps
```

Display the status of all processes holding UUCP locks. The output format is system dependent, as `uustat` simply invokes `ps` on each process holding a lock.

```
uustat -c rmail -o 168 -K -Q -M -N -W "Queued for over 1 week"
```

This will kill all ‘rmail’ commands that have been queued up waiting for delivery for over 1 week (168 hours). For each such command, mail will be sent both to the UUCP administrator and to the user who requested the rmail execution. The mail message sent will include the string given by the ‘-W’ option. The ‘-Q’ option prevents any of the jobs from being listed on the terminal, so any output from the program will be error messages.

2.5 Invoking uuname

```
uuname [-a] [--aliases]
uuname -l
uuname --local
```

By default, the uuname program simply lists the names of all the remote systems mentioned in the UUCP configuration files.

The uuname program may also be used to print the UUCP name of the local system.

The uuname program is mainly for use by shell scripts.

The following options may be given to uuname.

‘-a’
‘--aliases’
: List all aliases for remote systems, as well as their canonical names. Aliases may be specified in the ‘sys’ file (see Section 5.7.2 [Naming the System], page 73).

‘-l’
‘--local’
: Print the UUCP name of the local system, rather than listing the names of all the remote systems.

‘-x type’
‘--debug type’
‘-I file’
‘--config file’
‘-v’
‘--version’
‘--help’
: See Section 2.1 [Standard Options], page 9.

2.6 Invoking uulog

```
uulog [-#] [-n lines] [-sf system] [-u user] [-DSF] [--lines lines]
      [--system system] [--user user] [--debuglog] [--statslog]
      [--follow] [--follow=system]
```

The `uulog` program may be used to display the UUCP log file. Different options may be used to select which parts of the file to display.

'`-#`'
'`-n lines`'
'`--lines lines`'
: Here '`#`' is a number; e.g., '`-10`'. The specified number of lines is displayed from the end of the log file. The default is to display the entire log file, unless the '`-f`', '`-F`', or '`--follow`' options are used, in which case the default is to display 10 lines.

'`-s system`'
'`--system system`'
: Display only log entries pertaining to the specified system.

'`-u user`'
'`--user user`'
: Display only log entries pertaining to the specified user.

'`-D`'
'`--debuglog`'
: Display the debugging log file.

'`-S`'
'`--statslog`'
: Display the statistics log file.

'`-F`'
'`--follow`'
: Keep displaying the log file forever, printing new lines as they are appended to the log file.

'`-f system`'
'`--follow=system`'
: Keep displaying the log file forever, displaying only log entries pertaining to the specified system.

'`-X type`'
'`--debug type`'
'`-I file`'

'--config file'
'-v'
'--version'
'--help' See Section 2.1 [Standard Options], page 9. Note that uulog specifies the debugging type using '-X' rather than the usual '-x'.

The operation of uulog depends to some degree upon the type of log files generated by the UUCP programs. This is a compile time option. If the UUCP programs have been compiled to use HDB style log files, uulog changes in the following ways:

- The new options '-x' and '--uuxqtlog' may be used to list the uuxqt log file.
- It is no longer possible to omit all arguments: one of '-s', '--system', '-f', '--follow=system', '-D', '--debuglog', '-S', '--statslog', '-x', or '--uuxqtlog' must be used.
- The option '--system ANY' may be used to list log file entries which do not pertain to any particular system.

2.7 Invoking uuto

```
uuto files... system!user
```

The uuto program may be used to conveniently send files to a particular user on a remote system. It will arrange for mail to be sent to the remote user when the files arrive on the remote system, and he or she may easily retrieve the files using the uupick program (see Section 2.8 [Invoking uupick], page 25). Note that uuto does not provide any security—any user on the remote system can examine the files.

The last argument specifies the system and user name to which to send the files. The other arguments are the files or directories to be sent.

The uuto program is actually just a trivial shell script which invokes the uucp program with the appropriate arguments. Any option which may be given to uucp may also be given to uuto. See Section 2.2 [Invoking uucp], page 10.

2.8 Invoking uupick

```
uupick [-s system] [--system system]
```

The `uupick` program is used to conveniently retrieve files transferred by the `uuto` program.

For each file transferred by `uuto`, `uupick` will display the source system, the file name, and whether the name refers to a regular file or a directory. It will then wait for the user to specify an action to take. One of the following commands must be entered:

'`q`' Quit out of `uupick`.

'`RETURN`' Skip the file.

'`m [directory]`'
Move the file or directory to the specified directory. If no directory is specified, the file is moved to the current directory.

'`a [directory]`'
Move all files from this system to the specified directory. If no directory is specified, the files are moved to the current directory.

'`p`' List the file on standard output.

'`d`' Delete the file.

'`! [command]`'
Execute '`command`' as a shell escape.

The '`-s`' or '`--system`' option may be used to restrict `uupick` to only present files transferred from a particular system. The `uupick` program also supports the standard UUCP program options; see Section 2.1 [Standard Options], page 9.

2.9 Invoking cu

2.9.1 cu Description

```
cu [options] [system | phone | "dir"]
```

The `cu` program is used to call up another system and act as a dial in terminal. It can also do simple file transfers with no error checking.

The `cu` program takes a single non-option argument.

If the argument is the string '`dir`' cu will make a direct connection to the port. This may only be used by users with write access to the port, as it permits reprogramming the modem.

Otherwise, if the argument begins with a digit, it is taken to be a phone number to call.

Otherwise, it is taken to be the name of a system to call.

The '`-z`' or '`--system`' options may be used to name a system beginning with a digit, and the '`-c`' or '`--phone`' options may be used to name a phone number that does not begin with a digit.

The `cu` program locates a port to use in the UUCP configuration files. If a simple system name is given, it will select a port appropriate for that system. The '`-p`', '`--port`', '`-l`', '`--line`', '`-s`', and '`--speed`' options may be used to control the port selection.

When a connection is made to the remote system, `cu` forks into two processes. One reads from the port and writes to the terminal, while the other reads from the terminal and writes to the port.

2.9.2 cu Commands

The `cu` program provides several commands that may be used during the conversation. The commands all begin with an escape character, which by default is `~` (tilde). The escape character is only recognized at the beginning of a line. To send an escape character to the remote system at the start of a line, it must be entered twice. All commands are either a single character or a word beginning with `%` (percent sign).

The `cu` program recognizes the following commands.

'`~.`' Terminate the conversation.

'`~! command`'
: Run command in a shell. If command is empty, starts up a shell.

'`~$ command`'
: Run command, sending the standard output to the remote system.

'`~| command`'
: Run command, taking the standard input from the remote system.

'`~+ command`'
: Run command, taking the standard input from the remote system and sending the standard output to the remote system.

'`~#, ~%break`'
: Send a break signal, if possible.

'`~c directory, ~%cd directory`'
: Change the local directory.

'`~> file`' Send a file to the remote system. This just dumps the file over the communication line. It is assumed that the remote system is expecting it.

'`~<`' Receive a file from the remote system. This prompts for the local file name and for the remote command to execute to begin the file transfer. It continues accepting data until the contents of the '`eofread`' variable are seen.

'`~p from to`'
'`~%put from to`'
: Send a file to a remote Unix system. This runs the appropriate commands on the remote system.

'`~t from to`'
'`~%take from to`'
: Retrieve a file from a remote Unix system. This runs the appropriate commands on the remote system.

'`~s variable value`'
: Set a `cu` variable to the given value. If value is not given, the variable is set to '`true`'.

'`~! variable`'
: Set a `cu` variable to '`false`'.

`~z'` Suspend the cu session. This is only supported on some systems. On systems for which `^Z` may be used to suspend a job, `~^Z'` will also suspend the session.

`~%nostop'`
: Turn off XON/XOFF handling.

`~%stop'` Turn on XON/XOFF handling.

`~v'` List all the variables and their values.

`~?'` List all commands.

2.9.3 cu Variables

The `cu` program also supports several variables. They may be listed with the `~v'` command, and set with the `~s'` or `~!'` commands.

`escape'` The escape character. The default is `~` (tilde).

`delay'` If this variable is true, `cu` will delay for a second, after recognizing the escape character, before printing the name of the local system. The default is true.

`eol'` The list of characters which are considered to finish a line. The escape character is only recognized after one of these is seen. The default is `carriage return`, `^U`, `^C`, `^O`, `^D`, `^S`, `^Q`, `^R`.

`binary'` Whether to transfer binary data when sending a file. If this is false, then newlines in the file being sent are converted to carriage returns. The default is false.

`binary-prefix'`
: A string used before sending a binary character in a file transfer, if the `binary'` variable is true. The default is `^V'`.

`echo-check'`
: Whether to check file transfers by examining what the remote system echoes back. This probably doesn't work very well. The default is false.

`echonl'` The character to look for after sending each line in a file. The default is carriage return.

`timeout'` The timeout to use, in seconds, when looking for a character, either when doing echo checking or when looking for the `echonl'` character. The default is 30.

`kill'` The character to use delete a line if the echo check fails. The default is `^U`.

`resend'` The number of times to resend a line if the echo check continues to fail. The default is 10.

'eofwrite'
: The string to write after sending a file with the '~>' command. The default is '^D'.

'eofread'
: The string to look for when receiving a file with the ' ~<' command. The default is '$', which is intended to be a typical shell prompt.

'verbose'
: Whether to print accumulated information during a file transfer. The default is true.

2.9.4 cu Options

The following options may be given to cu.

'-e'
'--parity=even'
: Use even parity.

'-o'
'--parity=odd'
: Use odd parity.

'--parity=none'
: Use no parity. No parity is also used if both '-e' and '-o' are given.

'-h'
'--halfduplex'
: Echo characters locally (half-duplex mode).

'--nostop'
: Turn off XON/XOFF handling (it is on by default).

'-E char'
'--escape char'
: Set the escape character. Initially ~ (tilde). To eliminate the escape character, use '-E '''.

'-z system'
'--system system'
: The system to call.

'-c phone-number'
'--phone phone-number'
: The phone number to call.

'-p port'
'-a port'

'--port port'
: Name the port to use.

'-l line'
'--line line'
: Name the line to use by giving a device name. This may be used to dial out on ports that are not listed in the UUCP configuration files. Write access to the device is required.

'-s speed'
'-#'
'--speed speed'
: The speed (baud rate) to use. Here, '-#' means an actual number; e.g., '-9600'.

'-n'
'--prompt'
: Prompt for the phone number to use.

'-d' Enter debugging mode. Equivalent to '--debug all'.

'-x type'
'--debug type'
'-I file'
'--config file'
'-v'
'--version'
'--help' See Section 2.1 [Standard Options], page 9.

2.10 Invoking uucico

2.10.1 uucico Description

```
uucico [options]
```

The uucico daemon processes file transfer requests queued by uucp and uux. It is started when uucp or uux is run (unless they are given the '-r' or '--nouucico' options). It is also typically started periodically using entries in the 'crontab' table(s).

When uucico is invoked with '-r1', '--master', '-s', '--system', or '-S', the daemon will place a call to a remote system, running in master mode. Otherwise the daemon will start in slave mode, accepting a call from a remote system. Typically a special login name will be set up for UUCP which automatically invokes uucico when a remote system calls in and logs in under that name.

When uucico terminates, it invokes the uuxqt daemon, unless the '-q' or '--nouuxqt' options were given; uuxqt executes any work orders created by uux on a remote system, and any work orders created locally which have received remote files for which they were waiting.

If a call fails, uucico will normally refuse to retry the call until a certain (configurable) amount of time has passed. This may be overriden by the '-f', '--force', or '-S' options.

The '-l', '--prompt', '-e', or '--loop' options may be used to force uucico to produce its own prompts of 'login: ' and 'Password:'. When another uucico daemon calls in, it will see these prompts and log in as usual. The login name and password will normally be checked against a separate list kept specially for uucico, rather than the '/etc/passwd' file (see Section 5.6.2 [Configuration File Names], page 68). It is possible, on some systems, to configure uucico to use '/etc/passwd'. The '-l' or '--prompt' options will prompt once and then exit; in this mode the UUCP administrator, or the superuser, may use the '-u' or '--login' option to force a login name, in which case uucico will not prompt for one. The '-e' or '--loop' options will prompt again after the first session is over; in this mode uucico will permanently control a port.

If uucico receives a SIGQUIT, SIGTERM or SIGPIPE signal, it will cleanly abort any current conversation with a remote system and exit. If it receives a SIGHUP signal it will abort any current conversation, but will continue to place calls to (if invoked with '-r1' or '--master') and accept calls from (if invoked with '-e' or '--loop') other systems. If it receives a SIGINT signal it will finish the current conversation, but will not place or accept any more calls.

2.10.2 uucico Options

The following options may be given to uucico.

'-r1'
'--master'
: Start in master mode: call out to a remote system. Implied by '-s', '--system', or '-S'. If no system is specified, sequentially call every system for which work is waiting to be done.

'-r0'
'--slave'
: Start in slave mode. This is the default.

'`-s system`'
'`--system system`'
: Call the specified system.

'`-S system`'
: Call the specified system, ignoring any required wait. This is equivalent to '`-s system -f`'.

'`-f`'
'`--force`'
: Ignore any required wait for any systems to be called.

'`-l`'
'`--prompt`'
: Prompt for login name and password using '`login: `' and '`Password:`'. This allows `uucico` to be easily run from `inetd`. The login name and password are checked against the UUCP password file, which need not be '`/etc/passwd`'. The '`--login`' option may be used to force a login name, in which cause `uucico` will only prompt for a password.

'`-p port`'
'`--port port`'
: Specify a port to call out on or to listen to.

'`-e`'
'`--loop`'
: Enter an endless loop of login/password prompts and slave mode daemon execution. The program will not stop by itself; you must use `kill` to shut it down.

'`-w`'
'`--wait`'
: After calling out (to a particular system when '`-s`', '`--system`', or '`-S`' is specifed, or to all systems which have work when just '`-r1`' or '`--master`' is specifed), begin an endless loop as with '`--loop`'.

'`-q`'
'`--nouuxqt`'
: Do not start the `uuxqt` daemon when finished.

'`-c`'
'`--quiet`'
: If no calls are permitted at this time, then don't make the call, but also do not put an error message in the log file and do not update the system status (as reported by `uustat`). This can be convenient for automated polling scripts, which may want to simply attempt to call every system rather than worry about which particular systems may be called at the moment. This option also suppresses the log message indicating that there is no work to be done.

'`-C`'
'`--ifwork`'
: Only call the system named by '`-s`', '`--system`', or '`-S`' if there is work for that system.

'-D'
'--nodetach'
: Do not detach from the controlling terminal. Normally uucico detaches from the terminal before each call out to another system and before invoking uuxqt. This option prevents this.

'-u name'
'--login name'
: Set the login name to use instead of that of the invoking user. This option may only be used by the UUCP administrator or the superuser. If used with '--prompt', this will cause uucico to prompt only for the password, not the login name.

'-z'
'--try-next'
: If a call fails after the remote system is reached, try the next alternate rather than simply exiting.

'-i type'
'--stdin type'
: Set the type of port to use when using standard input. The only supported port type is TLI, and this is only available on machines which support the TLI networking interface. Specifying '-i TLI' causes uucico to use TLI calls to perform I/O.

'-X type' Same as the standard option '-x type'. Provided for historical compatibility.

'-x type'
'--debug type'
'-I file'
'--config file'
'-v'
'--version'
'--help' See Section 2.1 [Standard Options], page 9.

2.11 Invoking uuxqt

```
uuxqt [-c command] [-s system] [--command command] [--system system]
```

The uuxqt daemon executes commands requested by uux from either the local system or from remote systems. It is started automatically by the uucico daemon (unless uucico is given the '-q' or '--nouuxqt' options).

There is normally no need to run `uuxqt`, since it will be invoked by `uucico`. However, `uuxqt` can be invoked directly to provide greater control over the processing of the work queue.

Multiple invocations of `uuxqt` may be run at once, as controlled by the `max-uuxqts` configuration command; see Section 5.6.1 [Miscellaneous (config)], page 66.

The following options may be given to `uuxqt`.

'`-c command`'
'`--command command`'
: Only execute requests for the specified command. For example, '`uuxqt --command rmail`'.

'`-s system`'
'`--system system`'
: Only execute requests originating from the specified system.

'`-x type`'
'`--debug type`'
'`-I file`'
'`--config`'
'`-v`'
'`--version`'
'`--help`'
: See Section 2.1 [Standard Options], page 9.

2.12 Invoking uuchk

```
uuchk [-s system] [--system system]
```

The `uuchk` program displays information read from the UUCP configuration files. It should be used to ensure that UUCP has been configured correctly.

The '`-s`' or '`--system`' options may be used to display the configuration for just the specified system, rather than for all systems. The `uuchk` program also supports the standard UUCP program options; see Section 2.1 [Standard Options], page 9.

2.13 Invoking uuconv

```
uuconv -i type -o type [-p program] [--program program]
uuconv --input type --output type [-p program] [--program program]
```

The `uuconv` program converts UUCP configuration files from one format to another. The type of configuration file to read is specified using the '`-i`' or '`--input`' options. The type of configuration file to write is specified using the '`-o`' or '`--output`' options.

The supported configuration file types are '`taylor`', '`v2`', and '`hdb`'. For a description of the '`taylor`' configuration files, see Chapter 5 [Configuration Files], page 53. The other types of configuration files are used by traditional UUCP packages, and are not described in this manual.

An input configuration of type '`v2`' or '`hdb`' is read from a compiled in directory (specified by '`oldconfigdir`' in '`Makefile`'). An input configuration of type '`taylor`' is read from a compiled in directory by default, but may be overridden with the standard '`-I`' or '`--config`' options (see Section 2.1 [Standard Options], page 9).

The output configuration is written to files in the directory in which `uuconv` is run.

Some information in the input files may not be representable in the desired output format, in which case `uuconv` will silently discard it. The output of `uuconv` should be carefully checked before it is used. The `uuchk` program may be used for this purpose; see Section 2.12 [Invoking uuchk], page 34.

The '`-p`' or '`--program`' option may be used to convert specific `cu` configuration information, rather than the default of only converting the `uucp` configuration information; see Section 5.6 [config File], page 65.

The `uuchk` program also supports the standard UUCP program options; see Section 2.1 [Standard Options], page 9.

2.14 Invoking uusched

The `uusched` program is actually just a shell script which invokes the `uucico` daemon. It is provided for backward compatibility. It causes `uucico` to call all systems for which there is work. Any option which may be given to `uucico` may also be given to `uusched`. See Section 2.10 [Invoking uucico], page 30.

3 Installing Taylor UUCP

These are the installation instructions for the Taylor UUCP package.

3.1 Compiling Taylor UUCP

If you have a source code distribution, you must first compile it for your system. Free versions of Unix, such as Linux, NetBSD, or FreeBSD, often come with pre-compiled binary distributions of UUCP. If you are using a binary distribution, you may skip to the configuration section (see Section 3.4 [Configuration], page 41).

Follow these steps to compile the source code.

1. Take a look at the top of '`Makefile.in`' and set the appropriate values for your system. These control where the programs are installed and which user on the system owns them (normally they will be owned by a special user `uucp` rather than a real person; they should probably not be owned by `root`).
2. Run the shell script `configure`. This script was generated using the `autoconf` program written by David MacKenzie of the Free Software Foundation. It takes a while to run. It will generate the file '`config.h`' based on '`config.h.in`', and, for each source code directory, will generate '`Makefile`' based on '`Makefile.in`'.

   You can pass certain arguments to `configure` in the environment. Because `configure` will compile little test programs to see what is available on your system, you must tell it how to run your compiler. It recognizes the following environment variables:

   '`CC`' The C compiler. If this is not set, then if `configure` can find '`gcc`' it will use it, otherwise it will use '`cc`'.

   '`CFLAGS`' Flags to pass to the C compiler when compiling the actual code. If this is not set, `configure` will use '`-g`'.

   '`LDFLAGS`' Flags to pass to the C compiler when only linking, not compiling. If this is not set, `configure` will use the empty string.

   '`LIBS`' Libraries to pass to the C compiler. If this is not set, `configure` will use the empty string.

   '`INSTALL`' The program to run to install UUCP in the binary directory. If this is not set, then if `configure` finds the BSD `install` program, it will set this to '`install -c`'; otherwise, it will use '`cp`'.

Suppose, for example, you want to set the environment variable 'CC' to 'rcc'. If you are using `sh`, `bash`, or `ksh`, invoke `configure` as 'CC=rcc configure'. If you are using `csh`, do 'setenv CC rcc; sh configure'.

On some systems you will want to use 'LIBS=-lmalloc'. On Xenix derived versions of Unix do not use 'LIBS=-lx' because this will bring in the wrong versions of certain routines; if you want to use '-lx' you must specify 'LIBS=-lc -lx'.

If `configure` fails for some reason, or if you have a very weird system, you may have to configure the package by hand. To do this, copy the file 'config.h.in' to 'config.h' and edit it for your system. Then for each source directory (the top directory, and the subdirectories 'lib', 'unix', and 'uuconf') copy 'Makefile.in' to 'Makefile', find the words within `@` characters, and set them correctly for your system.

3. Igor V. Semenyuk provided this (lightly edited) note about ISC Unix 3.0. The `configure` script will default to passing '-posix' to `gcc`. However, using '-posix' changes the environment to POSIX, and on ISC 3.0, at least, the default for `POSIX_NO_TRUNC` is 1. This can lead to a problem when `uuxqt` executes `rmail`. `IDA sendmail` has dbm configuration files named 'mailertable.{dir,pag}'. Notice these names are 15 characters long. When `uuxqt` compiled with the '-posix' executes `rmail`, which in turn executes `sendmail`, the later is run under the POSIX environment too. This leads to `sendmail` bombing out with ''error opening 'M' database: name too long' (mailertable.dir)'. It's rather obscure behaviour, and it took me a day to find out the cause. I don't use the '-posix' switch; instead, I run `gcc` with '-D_POSIX_SOURCE', and add '-lcposix' to 'LIBS'.
4. On some versions of BSDI there is a bug in the shell which causes the default value for 'CFLAGS' to be set incorrectly. If 'echo ${CFLAGS--g}' echoes 'g' rather than '-g', then you must set 'CFLAGS' in the environment before running configure. There is a patch available from BSDI for this bug. (Reported by David Vrona).
5. On AIX 3.2.5, and possibly other versions, 'cc -E' does not work, reporting 'Option NOROCONST is not valid'. Test this before running configure by doing something like 'touch /tmp/foo.c; cc -E /tmp/foo.c'. This may give a warning about the file being empty, but it should not give the 'Option NOROCONST' warning. The workaround is to remove the ',noroconst' entry from the 'options' clause in the 'cc' stanza in '/etc/xlc.cfg'. (Reported by Chris Lewis).
6. You should verify that `configure` worked correctly by checking 'config.h' and the instances of 'Makefile'.
7. Edit 'policy.h' for your local system. The comments explain the various choices. The default values are intended to be reasonable, so you may not have to make any changes.

   You must decide what type of configuration files to use; for more information on the choices, see Section 3.4 [Configuration], page 41.

   You must also decide what sort of spool directory you want to use. If this is a new installation, I recommend 'SPOOLDIR_TAYLOR'; otherwise, select the spool directory corresponding to your existing UUCP package.

8. Type 'make' to compile everything.

   The 'tstuu.c' file is not particularly portable; if you can't figure out how to compile it you can safely ignore it, as it is only used for testing. To use STREAMS pseudo-terminals, tstuu.c must be compiled with '-DHAVE_STREAMS_PTYS'; this is not determined by the configure script.

   If you have any other problems there is probably a bug in the configure script.

9. Please report any problems you have. That is the only way they will get fixed for other people. Supply a patch if you can (see Section 7.3 [Patches], page 137), or just ask for help.

3.2 Testing the Compilation

If your system supports pseudo-terminals, and you compiled the code to support the new style of configuration files (HAVE_TAYLOR_CONFIG was set to 1 in 'policy.h'), you should be able to use the tstuu program to test the uucico daemon. If your system supports STREAMS based pseudo-terminals, you must compile tstuu.c with '-DHAVE_STREAMS_PTYS'. (The STREAMS based code was contributed by Marc Boucher).

To run tstuu, just type 'tstuu' with no arguments. You must run it in the compilation directory, since it runs './uucp', './uux' and './uucico'. The tstuu program will run a lengthy series of tests (it takes over ten minutes on a slow VAX). You will need a fair amount of space available in '/usr/tmp'. You will probably want to put it in the background. Do not use ^Z, because the program traps on SIGCHLD and winds up dying. The tstuu program will create a directory '/usr/tmp/tstuu' and fill it with configuration files, and create spool directories '/usr/tmp/tstuu/spool1' and '/usr/tmp/tstuu/spool2'.

If your system does not support the FIONREAD call, the 'tstuu' program will run very slowly. This may or may not get fixed in a later version.

The tstuu program will finish with an execute file named 'X.*something*' and a data file named 'D.*something*' in the directory '/usr/tmp/tstuu/spool1' (or, more likely, in subdirectories, depending on the choice of SPOOLDIR in 'policy.h'). Two log files will be created in the directory '/usr/tmp/tstuu'. They will be named 'Log1' and 'Log2', or, if you have selected HAVE_HDB_LOGGING in 'policy.h', 'Log1/uucico/test2' and 'Log2/uucico/test1'. There should be no errors in the log files.

You can test uuxqt with './uuxqt -I /usr/tmp/tstuu/Config1'. This should leave a command file 'C.*something*' and a data file 'D.*something*' in '/usr/tmp/tstuu/spool1' or in subdirectories. Again, there should be no errors in the log file.

Assuming you compiled the code with debugging enabled, the '-x' switch can be used to set debugging modes; see the debug command for details (see Section 5.6.4 [Debugging Levels], page 70). Use '-x all' to turn on all debugging and generate far more output than you will ever want to see. The uucico daemons will put debugging output in the files 'Debug1' and 'Debug2' in the directory '/usr/tmp/tstuu'. After that, you're pretty much on your own.

On some systems you can also use tstuu to test uucico against the system uucico, by using the '-u' switch. For this to work, change the definitions of ZUUCICO_CMD and UUCICO_EXECL at the top of 'tstuu.c' to something appropriate for your system. The definitions in 'tstuu.c' are what I used for Ultrix 4.0, on which '/usr/lib/uucp/uucico' is particularly obstinate about being run as a child; I was only able to run it by creating a login name with no password whose shell was '/usr/lib/uucp/uucico'. Calling login in this way will leave fake entries in 'wtmp' and 'utmp'; if you compile 'tstout.c' (in the 'contrib' directory) as a setuid root program, tstuu will run it to clear those entries out. On most systems, such hackery should not be necessary, although on SCO I had to su to root (uucp might also have worked) before I could run '/usr/lib/uucp/uucico'.

You can test uucp and uux (give them the '-r' switch to keep them from starting uucico) to make sure they create the right sorts of files. Unfortunately, if you don't know what the right sorts of files are, I'm not going to tell you here.

If you can not run tstuu, or if it fails inexplicably, don't worry about it too much. On some systems tstuu will fail because of problems using pseudo terminals, which will not matter in normal use. The real test of the package is talking to another system.

3.3 Installing the Binaries

You can install the executable files by becoming root and typing 'make install'. Or you can look at what 'make install' does and do it by hand. It tries to preserve your old programs, if any, but it only does this the first time Taylor UUCP is installed (so that if you install several versions of Taylor UUCP, you can still go back to your original UUCP programs). You can retrieve the original programs by typing 'make uninstall'.

Note that by default the programs are compiled with debugging information, and they are not stripped when they are installed. You may want to strip the installed programs to save disk space. For more information, see your system documentation for the strip program.

Of course, simply installing the executable files is not enough. You must also arrange for them to be used correctly.

3.4 Configuring Taylor UUCP

You will have to decide what types of configuration files you want to use. This package supports a new sort of configuration file; see Chapter 5 [Configuration Files], page 53. It also supports V2 configuration files (`L.sys`, `L-devices`, etc.) and HDB configuration files (`Systems`, `Devices`, etc.). No documentation is provided for V2 or HDB configuration files. All types of configuration files can be used at once, if you are so inclined. Currently using just V2 configuration files is not really possible, because there is no way to specify a dialer (there are no built in dialers, and the program does not know how to read `acucap` or `modemcap`); however, V2 configuration files can be used with a new style dial file (see Section 5.9 [dial File], page 93), or with a HDB `Dialers` file.

Use of HDB configuration files has two known bugs. A blank line in the middle of an entry in the `Permissions` file will not be ignored as it should be. Dialer programs, as found in some versions of HDB, are not recognized directly. If you must use a dialer program, rather than an entry in `Devices`, you must use the `chat-program` command in a new style dial file; see Section 5.9 [dial File], page 93. You will have to invoke the dialer program via a shell script or another program, since an exit code of 0 is required to recognize success; the `dialHDB` program in the `contrib` directory may be used for this purpose.

The `uuconv` (see Section 2.13 [Invoking uuconv], page 35) program can be used to convert from V2 or HDB configuration files to the new style (it can also do the reverse translation, if you are so inclined). It will not do all of the work, and the results should be carefully checked, but it can be quite useful.

If you are installing a new system, you will, of course, have to write the configuration files; see Chapter 5 [Configuration Files], page 53 for details on how to do this.

After writing the configuration files, use the `uuchk` program to verify that they are what you expect; see Section 2.12 [Invoking uuchk], page 34.

3.5 Testing the Installation

After you have written the configuration files, and verified them with the `uuchk` program (see Section 2.12 [Invoking uuchk], page 34), you must check that UUCP can correctly contact another system.

Tell `uucico` to dial out to the system by using the '`-s`' system switch (e.g., '`uucico -s uunet`'). The log file should tell you what happens. The exact location of the log file depends upon the settings in '`policy.h`' when you compiled the program, and on the use of the `logfile` command in the '`config`' file. Typical locations are '`/usr/spool/uucp/Log`' or a subdirectory under '`/usr/spool/uucp/.Log`'.

If you compiled the code with debugging enabled, you can use debugging mode to get a great deal of information about what sort of data is flowing back and forth; the various possibilities are described with the `debug` command (see Section 5.6.4 [Debugging Levels], page 70). When initially setting up a connection '`-x chat`' is probably the most useful (e.g., '`uucico -s uunet -x chat`'); you may also want to use '`-x handshake,incoming,outgoing`'. You can use '`-x`' multiple times on one command line, or you can give it comma separated arguments as in the last example. Use '`-x all`' to turn on all possible debugging information.

The debugging information is written to a file, normally '`/usr/spool/uucp/Debug`', although the default can be changed in '`policy.h`', and the '`config`' file can override the default with the `debugfile` command. The debugging file may contain passwords and some file contents as they are transmitted over the line, so the debugging file is only readable by the `uucp` user.

You can use the '`-f`' switch to force `uucico` to call out even if the last call failed recently; using '`-S`' when naming a system has the same effect. Otherwise the status file (in the '`.Status`' subdirectory of the main spool directory, normally '`/usr/spool/uucp`') (see Section 4.4.2 [Status Directory], page 47) will prevent too many attempts from occurring in rapid succession.

On older System V based systems which do not have the `setreuid` system call, problems may arise if ordinary users can start an execution of `uuxqt`, perhaps indirectly via `uucp` or `uux`. UUCP jobs may wind up executing with a real user ID of the user who invoked `uuxqt`, which can cause problems if the UUCP job checks the real user ID for security purposes. On such systems, it is safest to put '`run-uuxqt never`' (see Section 5.6.1 [Miscellaneous (config)], page 66) in the '`config`' file, so that `uucico` never starts `uuxqt`, and invoke `uuxqt` directly from a '`crontab`' file.

Please let me know about any problems you have and how you got around them. If you do report a problem, please include the version number of the package you are using, the operating system you are running it on, and a sample of the debugging file showing the problem (debugging information is usually what is needed, not just the log file). General questions such as "why doesn't `uucico` dial out" are impossible to answer without much more information.

4 Using Taylor UUCP

4.1 Calling Other Systems

By default `uucp` and `uux` will automatically start up `uucico` to call another system whenever work is queued up. However, the call may fail, or you may have put in time restrictions which prevent the call at that time (perhaps because telephone rates are high) (see Section 5.7.3.1 [When to Call], page 74). Also, a remote system may have work queued up for your system, but may not be calling you for some reason (perhaps you have agreed that your system should always place the call). To make sure that work gets transferred between the systems withing a reasonable time period, you should arrange to periodically invoke `uucico`.

These periodic invocations are normally triggered by entries in the '`crontab`' file. The exact format of '`crontab`' files, and how new entries are added, varies from system to system; check your local documentation (try '`man cron`').

To attempt to call all systems with outstanding work, use the command '`uucico -r1`'. To attempt to call a particular system, use the command '`uucico -s` *system*'. To attempt to call a particular system, but only if there is work for it, use the command '`uucico -C -s` *system*'. (see Section 2.10 [Invoking uucico], page 30).

A common case is to want to try to call a system at a certain time, with periodic retries if the call fails. A simple way to do this is to create an empty UUCP command file, known as a *poll file*. If a poll file exists for a system, then '`uucico -r1`' will place a call to it. If the call succeeds, the poll file will be deleted.

A poll file can be easily created using the '`uux`' command, by requesting the execution of an empty command. To create a poll file for *system*, just do something like this:

```
uux -r system!
```

The '`-r`' tells '`uux`' to not start up '`uucico`' immediately. Of course, if you do want '`uucico`' to start up right away, omit the '`-r`'; if the call fails, the poll file will be left around to cause a later call.

For example, I use the following crontab entries locally:

```
45 * * * * /bin/echo /usr/lib/uucp/uucico -r1 | /bin/su uucpa
40 4,10,15 * * * /usr/bin/uux -r uunet!
```

Every hour, at 45 minutes past, this will check if there is any work to be done, and, if there is, will call the appropriate system. Also, at 4:40am, 10:40am, and 3:40pm, this will create a poll file file for '`uunet`', forcing the next run of `uucico` to call '`uunet`'.

4.2 Accepting Calls

To accept calls from another system, you must arrange matters such that when that system calls in, it automatically invokes `uucico` on your system.

The most common arrangement is to create a special user name and password for incoming UUCP calls. This user name typically uses the same user ID as the regular `uucp` user (Unix permits several user names to share the same user ID). The shell for this user name should be set to `uucico`.

Here is a sample '`/etc/passwd`' line to accept calls from a remote system named airs:

```
Uairs:password:4:8:airs UUCP:/usr/spool/uucp:/usr/lib/uucp/uucico
```

The details may vary on your system. You must use reasonable user and group ID's. You must use the correct file name for `uucico`. The *password* must appear in the UUCP configuration files on the remote system, but will otherwise never be seen or typed by a human.

Note that `uucico` appears as the login shell, and that it will be run with no arguments. This means that it will start in slave mode and accept an incoming connection. See Section 2.10 [Invoking uucico], page 30.

On some systems, creating an empty file named '`.hushlogin`' in the home directory will skip the printing of various bits of information when the remote `uucico` logs in, speeding up the UUCP connection process.

For the greatest security, each system which calls in should use a different user name, each with a different password, and the `called-login` command should be used in the '`sys`' file to ensure that the correct login name is used. See Section 5.7.4 [Accepting a Call], page 78, and see Section 5.11 [Security], page 97.

If you never need to dial out from your system, but only accept incoming calls, you can arrange for `uucico` to handle logins itself, completely controlling the port, by using the '`--endless`' option. See Section 2.10 [Invoking uucico], page 30.

4.3 Using UUCP for Mail and News.

Taylor UUCP does not include a mail package. All Unix systems come with some sort of mail delivery agent, typically `sendmail` or `MMDF`. Source code is available for some alternative mail delivery agents, such as `IDA sendmail` and `smail`.

Taylor UUCP also does not include a news package. The two major Unix news packages are `C-news` and `INN`. Both are available in source code form.

Configuring and using mail delivery agents is a notoriously complex topic, and I will not be discussing it here. Configuring news systems is usually simpler, but I will not be discussing that either. I will merely describe the interactions between the mail and news systems and UUCP.

A mail or news system interacts with UUCP in two ways: sending and receiving.

4.3.1 Sending mail or news via UUCP

When mail is to be sent from your machine to another machine via UUCP, the mail delivery agent will invoke `uux`. It will generally run a command such as '`uux -` *system*`!rmail` *address*', where *system* is the remote system to which the mail is being sent. It may pass other options to `uux`, such as '`-r`' or '`-g`' (see Section 2.3 [Invoking uux], page 12).

The news system also invokes `uux` in order to transfer articles to another system. The only difference is that news will use `uux` to invoke `rnews` on the remote system, rather than `rmail`.

You should arrange for your mail and news systems to invoke the Taylor UUCP version of `uux`. If you only have Taylor UUCP, or if you simply replace any existing version of `uux` with the Taylor UUCP version, this will probably happen automatically. However, if you have two UUCP packages installed on your system, you will probably have to modify the mail and news configuration files in some way.

Actually, if both the system UUCP and Taylor UUCP are using the same spool directory format, the system `uux` will probably work fine with the Taylor `uucico` (the reverse is not the case: the Taylor `uux` requires the Taylor `uucico`). However, data transfer will be somewhat more efficient if the Taylor `uux` is used.

4.3.2 Receiving mail or news via UUCP

To receive mail, all that is necessary is for UUCP to invoke `rmail`. Any mail delivery agent will provide an appropriate version of `rmail`; you must simply make sure that it is in the command path used by UUCP (it almost certainly already is). The default command path is set in '`policy.h`', and it may be overridden for a particular system by the `command-path` command (see Section 5.7.7 [Miscellaneous (sys)], page 87).

Similarly, for news UUCP must be able to invoke `rnews`. Any news system will provide a version of `rnews`, and you must ensure that is in a directory on the path that UUCP will search.

4.4 The Spool Directory Layout

In general, the layout of the spool directory may be safely ignored. However, it is documented here for the curious. This description only covers the `SPOOLDIR_TAYLOR` layout. The ways in which the other spool directory layouts differ are described in the source file '`unix/spool.c`'.

Directories and files are only created when they are needed, so a typical system will not have all of the entries described here.

4.4.1 System Spool Directories

'*system*'
: There is a subdirectory of the main spool directory for each remote system.

'*system*`/C.`'
: This directory stores files describing file transfer commands to be sent to the *system*. Each file name starts with '`C.`*g*', where *g* is the job grade. Each file contains one or more commands. For details of the commands, see Section 6.5.2 [UUCP Protocol Commands], page 111.

'*system*/D.'

This directory stores data files. Files with names like 'D.*gssss*', where *g* is the grade and *ssss* is a sequence number, are waiting to be transferred to the *system*, as directed by the files in the '*system*/C.' directory. Files with other names, typically 'D.*systemgssss*', have been received from *system* and are waiting to be processed by an execution file in the '*system*/X.' directory.

'*system*/D.X'

This directory stores data files which will become execution files on the remote system. In current practice, this directory rarely exists, because most simple executions, including typical uses of `rmail` and `rnews`, send an 'E' command rather than an execution file (see Section 6.5.2.4 [The E Command], page 116).

'*system*/X.'

This directory stores execution files which have been received from *system*. This directory normally exists, even though the corresponding 'D.X' directory does not, because `uucico` will create an execution file on the fly when it receives an 'E' command.

'*system*/SEQF'

This file holds the sequence number of the last job sent to *system*. The sequence number is used to ensure that file names are unique in the remote system spool directory. The file is four bytes long. Sequence numbers are composed of digits and the upper case letters.

4.4.2 Status Directory

'.Status'

This directory holds status files for each remote system. The name of the status file is the name of the system which it describes. Each status file describes the last conversation with the system. Running `uustat --status` basically just formats and prints the contents of the status files (see Section 2.4.3 [uustat Examples], page 20).

Each status file has a single text line with six fields.

code

A code indicating the status of the last conversation. The following values are defined, though not all are actually used.

'0' Conversation completed normally.

'1' `uucico` was unable to open the port.

'2' The last call to the system failed while dailing.

'3' The last call to the system failed while logging in.

'4' The last call to the system failed during the initial UUCP protocol handshake (see Section 6.5.1 [The Initial Handshake], page 107).

'5' The last call to the system failed after the initial handshake.

'6' `uucico` is currently talking to the system.

'7' The last call to the system failed because it was the wrong time to call (this is not used if calling the system is never permitted).

retries
The number of retries since the last successful call.

time of last call
The time of the last call, in seconds since the epoch (as returned by the `time` system call).

wait
If the last call failed, this is the number of seconds since the last call before `uucico` may attempt another call. This is set based on the retry time; see Section 5.7.3.1 [When to Call], page 74. The '`-f`' or '`-S`' options to `uucico` direct it to ignore this wait time; see Section 2.10 [Invoking uucico], page 30.

description
A text description of the status, corresponding to the code in the first field. This may contain spaces.

system name
The name of the remote system.

4.4.3 Execution Subdirectories

'`.Xqtdir`'
When `uuxqt` executes a job requested by `uux`, it first changes the working directory to the '`.Xqtdir`' subdirectory. This permits the job to create any sort of temporary file without worrying about overwriting other files in the spool directory. Any files left in the '`.Xqtdir`' subdirectory are removed after each execution is complete.

'`.Xqtdir`*nnnn*'
When several instances of `uuxqt` are executing simultaneously, each one executes jobs in a separate directory. The first uses '`.Xqtdir`', the second uses '`.Xqtdir0001`', the third uses '`.Xqtdir0002`', and so forth.

'`.Corrupt`'
If `uuxqt` encounters an execution file which it is unable to parse, it saves it in the '`.Corrupt`' directory, and sends mail about it to the UUCP administrator.

'.Failed' If uuxqt executes a job, and the job fails, and there is enough disk space to hold the command file and all the data files, then uuxqt saves the files in the '.Failed' directory, and sends mail about it to the UUCP administrator.

4.4.4 Other Spool Subdirectories

'.Sequence'
: This directory holds conversation sequence number files. These are used if the sequence command is used for a system (see Section 5.7.7 [Miscellaneous (sys)], page 87). The sequence number for the system *system* is stored in the file '.Sequence/*system*'. It is simply stored as a printable number.

'.Temp' This directory holds data files as they are being received from a remote system, before they are moved to their final destination. For file send requests which use a valid temporary file name in the *temp* field of the 'S' or 'E' command (see Section 6.5.2.1 [The S Command], page 111), uucico receives the file into '.Temp/*system*/*temp*', where *system* is the name of the remote system, and *temp* is the temporary file name. If a conversation fails during a file transfer, these files are used to automatically restart the file transfer from the point of failure.

If the 'S' or 'E' command does not include a temporary file name, automatic restart is not possible. In this case, the files are received into a randomly named file in the '.Temp' directory itself.

'.Preserve'
: This directory holds data files which could not be transferred to a remote system for some reason (for example, the data file might be large, and exceed size restrictions imposed by the remote system). When a locally requested file transfer fails, uucico will store the data file in the '.Preserve' directory, and send mail to the requestor describing the failure and naming the saved file.

'.Received'
: This directory records which files have been received. If a conversation fails just after uucico acknowledges receipt of a file, it is possible for the acknowledgement to be lost. If this happens, the remote system will resend the file. If the file were an execution request, and uucico did not keep track of which files it had already received, this could lead to the execution being performed twice.

To avoid this problem, when a conversation fails, uucico records each file that has been received, but for which the remote system may not have received the acknowledgement. It records this information by creating an empty file with the name '.Received/*system*/*temp*', where *system* is the name of the remote system, and *temp* is the *temp* field of the 'S' or 'E' command from the remote system (see Section 6.5.2.1

[The S Command], page 111). Then, if the remote system offers the file again in the next conversation, `uucico` refuses the send request and deletes the record in the '`.Received`' directory. This approach only works for file sends which use a temporary file name, but this is true of all execution requests.

4.4.5 Lock Files in the Spool Directory

Lock files for devices and systems are stored in the lock directory, which may or may not be the same as the spool directory. The lock directory is set at compilation time by `LOCKDIR` in '`policy.h`', which may be overridden by the `lockdir` command in the '`config`' file (see Section 5.6.1 [Miscellaneous (config)], page 66).

For a description of the names used for device lock files, and the format of the contents of a lock file, see Section 6.3 [UUCP Lock Files], page 104.

'`LCK..`*sys*' A lock file for a system, where *sys* is the system name. As noted above, these lock files are kept in the lock directory, which may not be the spool directory. These lock files are created by `uucico` while talking to a remote system, and are used to prevent multiple simultaneous conversations with a system.

On systems which limit file names to 14 characters, only the first eight characters of the system name are used in the lock file name. This requires that the names of each directly connected remote system be unique in the first eight characters.

'`LCK.XQT.`*NN*'
When `uuxqt` starts up, it uses lock files to determine how many other `uuxqt` daemons are currently running. It first tries to lock '`LCK.XQT.0`', then '`LCK.XQT.1`', and so forth. This is used to implement the `max-uuxqts` command (see Section 5.6.1 [Miscellaneous (config)], page 66). It is also used to parcel out the '`.Xqtdir`' subdirectories (see Section 4.4.3 [Execution Subdirectories], page 48).

'`LXQ.`*cmd*' When `uuxqt` is invoked with the '`-c`' or '`--command`' option (see Section 2.11 [Invoking uuxqt], page 33), it creates a lock file named after the command it is executing. For example, '`uuxqt -c rmail`' will create the lock file '`LXQ.rmail`'. This prevents other `uuxqt` daemons from executing jobs of the specified type.

'*system*`/X./L.`*xxx*'
While `uuxqt` is executing a particular job, it creates a lock file with the same name as the '`X.`' file describing the job, but replacing the initial '`X`' with '`L`'. This ensures that if multiple `uuxqt` daemons are running, they do not simultaneously execute the same job.

'LCK..SEQ'

This lock file is used to control access to the sequence files for each system (see Section 4.4.1 [System Spool Directories], page 46). It is only used on systems which do not support POSIX file locking using the `fcntl` system call.

4.5 Cleaning the Spool Directory

The spool directory may need to be cleaned up periodically. Under some circumstances, files may accumulate in various subdirectories, such as '.Preserve' (see Section 4.4.4 [Other Spool Subdirectories], page 49) or '.Corrupt' (see Section 4.4.3 [Execution Subdirectories], page 48).

Also, if a remote system stops calling in, you may want to arrange for any queued up mail to be returned to the sender. This can be done using the `uustat` command (see Section 2.4 [Invoking uustat], page 16).

The 'contrib' directory includes a simple 'uuclean' script which may be used as an example of a clean up script. It can be run daily out of 'crontab'.

You should periodically trim the UUCP log files, as they will otherwise grow without limit. The names of the log files are set in 'policy.h', and may be overridden in the configuration file (see Section 5.6 [config File], page 65). By default they are are '/usr/spool/uucp/Log' and '/usr/spool/uucp/Stats'. You may find the `savelog` program in the 'contrib' directory to be of use. There is a manual page for it in 'contrib' as well.

5 Taylor UUCP Configuration Files

This chapter describes the configuration files accepted by the Taylor UUCP package if compiled with `HAVE_TAYLOR_CONFIG` set to 1 in '`policy.h`'.

The configuration files are normally found in the directory *newconfigdir*, which is defined by the '`Makefile`' variable '`newconfigdir`'; by default *newconfigdir* is '`/usr/local/conf/uucp`'. However, the main configuration file, '`config`', is the only one which must be in that directory, since it may specify a different location for any or all of the other files. You may run any of the UUCP programs with a different main configuration file by using the '`-I`' or '`--config`' option; this can be useful when testing a new configuration. When you use the '`-I`' option the programs will revoke any setuid privileges.

5.1 Configuration File Overview

UUCP uses several different types of configuration files, each describing a different kind of information. The commands permitted in each file are described in detail below. This section is a brief description of some of the different types of files.

The '`config`' file is the main configuration file. It describes general information not associated with a particular remote system, such as the location of various log files. There are reasonable defaults for everything that may be specified in the '`config`' file, so you may not actually need one on your system.

There may be only one '`config`' file, but there may be one or more of each other type of file. The default is one file for each type, but more may be listed in the '`config`' file.

The '`sys`' files are used to describe remote systems. Each remote system to which you connect must be listed in a '`sys`' file. A '`sys`' file will include information for a system, such as the speed (baud rate) to use, or when to place calls.

For each system you wish to call, you must describe one or more ports; these ports may be defined directly in the '`sys`' file, or they may be defined in a '`port`' file.

The '`port`' files are used to describe ports. A port is a particular hardware connection on your computer. You would normally define as many ports as there are modems attached to your computer. A TCP connection is also described using a port.

The '`dial`' files are used to describe dialers. Dialer is essentially another word for modem. The '`dial`' file describes the commands UUCP should use to dial out on a particular type of modem. You would normally define as many dialers as there are types of modems attached to your computer. For example, if you have three Telebit modems used for UUCP, you would probably define three ports and one dialer.

There are other types of configuration files, but these are the important ones. The other types are described below.

5.2 Configuration File Format

All the configuration files follow a simple line-oriented '*keyword value*' format. Empty lines are ignored, as are leading spaces; unlike HDB, lines with leading spaces are read. The first word on each line is a keyword. The rest of the line is interpreted according to the keyword. Most keywords are followed by numbers, boolean values or simple strings with no embedded spaces.

The `#` character is used for comments. Everything from a `#` to the end of the line is ignored unless the `#` is preceded by a `\` (backslash); if the `#` is preceeded by a `\`, the `\` is removed but the `#` remains in the line. This can be useful for a phone number containing a `#`. To enter the sequence '`\#`', use '`\\#`'.

The backslash character may be used to continue lines. If the last character in a line is a backslash, the backslash is removed and the line is continued by the next line. The second line is attached to the first with no intervening characters; if you want any whitespace between the end of the first line and the start of the second line, you must insert it yourself.

However, the backslash is not a general quoting character. For example, you cannot use it to get an embedded space in a string argument.

Everything after the keyword must be on the same line. A *boolean* may be specified as `y`, `Y`, `t`, or `T` for true and `n`, `N`, `f`, or `F` for false; any trailing characters are ignored, so `true`, `false`, etc., are also acceptable.

5.3 Examples of Configuration Files

This section provides few typical examples of configuration files. There are also sample configuration files in the 'sample' subdirectory of the distribution.

5.3.1 config File Examples

To start with, here are some examples of uses of the main configuration file, 'config'. For a complete description of the commands that are permitted in 'config', see Section 5.6 [config File], page 65.

In many cases you will not need to create a 'config' file at all. The most common reason to create one is to give your machine a special UUCP name. Other reasons might be to change the UUCP spool directory, or to permit any remote system to call in.

If you have an internal network of machines, then it is likely that the internal name of your UUCP machine is not the name you want to use when calling other systems. For example, here at 'airs.com' our mail/news gateway machine is named 'elmer.airs.com' (it is one of several machines all named '*localname*.airs.com'). If we did not provide a 'config' file, then our UUCP name would be 'elmer'; however, we actually want it to be 'airs'. Therefore, we use the following line in 'config':

```
nodename airs
```

The UUCP spool directory name is set in 'policy.h' when the code is compiled. You might at some point decide that it is appropriate to move the spool directory, perhaps to put it on a different disk partition. You would use the following commands in 'config' to change to directories on the partition '/uucp':

```
spool /uucp/spool
pubdir /uucp/uucppublic
logfile /uucp/spool/Log
debugfile /uucp/spool/Debug
```

You would then move the contents of the current spool directory to '/uucp/spool'. If you do this, make sure that no UUCP processes are running while you change 'config' and move the spool directory.

Suppose you wanted to permit any system to call in to your system and request files. This is generally known as *anonymous UUCP*, since the systems which call in are effectively anonymous. By default, unknown systems are not permitted to call in. To permit this you must use the `unknown` command in '`config`'. The `unknown` command is followed by any command that may appear in the system file; for full details, see Section 5.7 [sys File], page 71.

I will show two possible anonymous UUCP configurations. The first will let any system call in and download files, but will not permit them to upload files to your system.

```
# No files may be transferred to this system
unknown receive-request no
# The public directory is /usr/spool/anonymous
unknown pubdir /usr/spool/anonymous
# Only files in the public directory may be sent (the default anyhow)
unknown remote-send ~
```

Setting the public directory is convenient for the systems which call in. It permits to request a file by prefixing it with '`~/`'. For example, assuming your system is known as '`server`', then to retrieve the file '`/usr/spool/anonymous/INDEX`' a user on a remote site could just enter '`uucp server!~/INDEX ~`'; this would transfer '`INDEX`' from '`server`''s public directory to the user's local public directory. Note that when using '`csh`' or '`bash`' the `!` and the second `~` must be quoted.

The next example will permit remote systems to upload files to a special directory named '`/usr/spool/anonymous/upload`'. Permitting a remote system to upload files permits it to send work requests as well; this example is careful to prohibit commands from unknown systems.

```
# No commands may be executed (the list of permitted commands is empty)
unknown commands
# The public directory is /usr/spool/anonymous
unknown pubdir /usr/spool/anonymous
# Only files in the public directory may be sent; users may not download
# files from the upload directory
unknown remote-send ~ !~/upload
# May only upload files into /usr/spool/anonymous/upload
unknown remote-receive ~/upload
```

5.3.2 Leaf Example

A relatively common simple case is a *leaf site*, a system which only calls or is called by a single remote site. Here is a typical '`sys`' file that might be used in such a case. For full details on what commands can appear in the '`sys`' file, see Section 5.7 [sys File], page 71.

This is the 'sys' file that is used at 'airs.com'. We use a single modem to dial out to 'uunet'. This example shows how you can specify the port and dialer information directly in the 'sys' file for simple cases. It also shows the use of the following:

`call-login`
: Using `call-login` and `call-password` allows the default login chat script to be used. In this case, the login name is specified in the call-out login file (see Section 5.6.2 [Configuration File Names], page 68).

`call-timegrade`
: 'uunet' is requested to not send us news during the daytime.

`chat-fail`
: If the modem returns 'BUSY' or 'NO CARRIER' the call is immediately aborted.

`protocol-parameter`
: Since 'uunet' tends to be slow, the default timeout has been increased.

This 'sys' file relies on certain defaults. It will allow 'uunet' to queue up 'rmail' and 'rnews' commands. It will allow users to request files from 'uunet' into the UUCP public directory. It will also allow 'uunet' to request files from the UUCP public directory; in fact 'uunet' never requests files, but for additional security we could add the line 'request false'.

```
# The following information is for uunet
system uunet

# The login name and password are kept in the callout password file
call-login *
call-password *

# We can send anything at any time.
time any

# During the day we only accept grade `Z' or above; at other times
# (not mentioned here) we accept all grades.  uunet queues up news
# at grade `d', which is lower than `Z'.
call-timegrade Z Wk0755-2305,Su1655-2305

# The phone number.
phone 7389449

# uunet tends to be slow, so we increase the timeout
chat-timeout 120

# We are using a preconfigured Telebit 2500.
port type modem
```

```
port device /dev/ttyd0
port speed 19200
port carrier true
port dialer chat "" ATZ\r\d\c OK ATDT\D CONNECT
port dialer chat-fail BUSY
port dialer chat-fail NO\sCARRIER
port dialer complete \d\d+++\d\dATH\r\c
port dialer abort \d\d+++\d\dATH\r\c

# Increase the timeout and the number of retries.
protocol-parameter g timeout 20
protocol-parameter g retries 10
```

5.3.3 Gateway Example

Many organizations have several local machines which are connected by UUCP, and a single machine which connects to the outside world. This single machine is often referred to as a *gateway* machine.

For this example I will assume a fairly simple case. It should still provide a good general example. There are three machines, 'elmer', 'comton' and 'bugs'. 'elmer' is the gateway machine for which I will show the configuration file. 'elmer' calls out to 'uupsi'. As an additional complication, 'uupsi' knows 'elmer' as 'airs'; this will show how a machine can have one name on an internal network but a different name to the external world. 'elmer' has two modems. It also has an TCP connection to 'uupsi', but since that is supposed to be reserved for interactive work (it is, perhaps, only a 9600 baud SLIP line) it will only use it if the modems are not available.

A network this small would normally use a single 'sys' file. However, for pedagogical purposes I will show two separate 'sys' files, one for the local systems and one for 'uupsi'. This is done with the `sysfile` command in the 'config' file. Here is the 'config' file.

```
# This is config
# The local sys file
sysfile /usr/local/lib/uucp/sys.local
# The remote sys file
sysfile /usr/local/lib/uucp/sys.remote
```

Using the defaults feature of the 'sys' file can greatly simplify the listing of local systems. Here is 'sys.local'. Note that this assumes that the local systems are trusted; they are permited to request any world readable file and to write files into any world writable directory.

```
# This is sys.local
```

```
# Get the login name and password to use from the call-out file
call-login *
call-password *

# The systems must use a particular login
called-login Ulocal

# Permit sending any world readable file
local-send /
remote-send /

# Permit receiving into any world writable directory
local-receive /
remote-receive /

# Call at any time
time any

# Use port1, then port2
port port1

alternate

port port2

# Now define the systems themselves.  Because of all the defaults we
# used, there is very little to specify for the systems themselves.

system comton
phone 5551212

system bugs
phone 5552424
```

The 'sys.remote' file describes the 'uupsi' connection. The `myname` command is used to change the UUCP name to 'airs' when talking to 'uupsi'.

```
# This is sys.remote
# Define uupsi
system uupsi

# The login name and password are in the call-out file
call-login *
call-password *

# We can call out at any time
time any

# uupsi uses a special login name
```

```
called-login Uuupsi

# uuspi thinks of us as ‘airs’
myname airs

# The phone number
phone 5554848

# We use port2 first, then port1, then TCP

port port2

alternate

port port1

alternate

# We don’t bother to make a special entry in the port file for TCP, we
# just describe the entire port right here.  We use a special chat
# script over TCP because the usual one confuses some TCP servers.
port type TCP
address uu.psi.com
chat ogin: \L word: \P
```

The ports are defined in the file ‘port’ (see Section 5.8 [port File], page 89). For this example they are both connected to the same type of 2400 baud Hayes-compatible modem.

```
# This is port

port port1
type modem
device /dev/ttyd0
dialer hayes
speed 2400

port port2
type modem
device /dev/ttyd1
dialer hayes
speed 2400
```

Dialers are described in the ‘dial’ file (see Section 5.9 [dial File], page 93).

```
# This is dial

dialer hayes
```

```
# The chat script used to dial the phone.  \D is the phone number.
chat "" ATZ\r\d\c OK ATDT\D CONNECT

# If we get BUSY or NO CARRIER we abort the dial immediately
chat-fail BUSY
chat-fail NO\sCARRIER

# When the call is over we make sure we hangup the modem.
complete \d\d+++\d\dATH\r\c
abort \d\d+++\d\dATH\r\c
```

5.4 Time Strings

Several commands use time strings to specify a range of times. This section describes how to write time strings.

A time string may be a list of simple time strings separated with a vertical bar '`|`' or a comma '`,`'.

Each simple time string must begin with '`Su`', '`Mo`', '`Tu`', '`We`', '`Th`', '`Fr`', or '`Sa`', or '`Wk`' for any weekday, or '`Any`' for any day.

Following the day may be a range of hours separated with a hyphen using 24 hour time. The range of hours may cross 0; for example '`2300-0700`' means any time except 7 AM to 11 PM. If no time is given, calls may be made at any time on the specified day(s).

The time string may also be the single word '`Never`', which does not match any time. The time string may also be a single word with a name defined in a previous `timetable` command (see Section 5.6.1 [Miscellaneous (config)], page 66).

Here are a few sample time strings with an explanation of what they mean.

'`Wk2305-0855,Sa,Su2305-1655`'
: This means weekdays before 8:55 AM or after 11:05 PM, any time Saturday, or Sunday before 4:55 PM or after 11:05 PM. These are approximately the times during which night rates apply to phone calls in the U.S.A. Note that this time string uses, for example, '`2305`' rather than '`2300`'; this will ensure a cheap rate phone call even if the computer clock is running up to five minutes ahead of the real time.

`Wk0905-2255,Su1705-2255`
: This means weekdays from 9:05 AM to 10:55 PM, or Sunday from 5:05 PM to 10:55 PM. This is approximately the opposite of the previous example.

`Any`
: This means any day. Since no time is specified, it means any time on any day.

5.5 Chat Scripts

Chat scripts are used in several different places, such as dialing out on modems or logging in to remote systems. Chat scripts are made up of pairs of strings. The program waits until it sees the first string, known as the *expect* string, and then sends out the second string, the *send* string.

Each chat script is defined using a set of commands. These commands always end in a string beginning with `chat`, but may start with different strings. For example, in the '`sys`' file there is one set of commands beginning with `chat` and another set beginning with `called-chat`. The prefixes are only used to disambiguate different types of chat scripts, and this section ignores the prefixes when describing the commands.

`chat` *strings*
: Specify a chat script. The arguments to the `chat` command are pairs of strings separated by whitespace. The first string of each pair is an expect string, the second is a send string. The program will wait for the expect string to appear; when it does, the program will send the send string. If the expect string does not appear within a certain number of seconds (as set by the `chat-timeout` command), the chat script fails and, typically, the call is aborted. If the final expect string is seen (and the optional final send string has been sent), the chat script is successful.

  An expect string may contain additional subsend and subexpect strings, separated by hyphens. If the expect string is not seen, the subsend string is sent and the chat script continues by waiting for the subexpect string. This means that a hyphen may not appear in an expect string; on an ASCII system, use '`\055`' instead.

  An expect string may simply be '`""`', meaning to skip the expect phase. Otherwise, the following escape characters may appear in expect strings:

  '`\b`' a backspace character

  '`\n`' a newline or line feed character

  '`\N`' a null character (for HDB compatibility)

  '`\r`' a carriage return character

'`\s`' a space character

'`\t`' a tab character

'`\\`' a backslash character

'`\`*ddd*' character *ddd*, where *ddd* are up to three octal digits

'`\x`*ddd*' character *ddd*, where *ddd* are hexadecimal digits.

As in C, there may be up to three octal digits following a backslash, but the hexadecimal escape sequence continues as far as possible. To follow a hexadecimal escape sequence with a hex digit, interpose a send string of '`""`'.

A chat script expect string may also specify a timeout. This is done by using the escape sequence '`\W`*seconds*'. This escape sequence may only appear at the very end of the expect string. It temporarily overrides the timeout set by `chat-timeout` (described below) only for the expect string to which it is attached.

A send string may simply be '`""`' to skip the send phase. Otherwise, all of the escape characters legal for expect strings may be used, and the following escape characters are also permitted:

'`EOT`' send an end of transmission character (`^D`)

'`BREAK`' send a break character (may not work on all systems)

'`\c`' suppress trailing carriage return at end of send string

'`\d`' delay sending for 1 or 2 seconds

'`\e`' disable echo checking

'`\E`' enable echo checking

'`\K`' same as '`BREAK`' (for HDB compatibility)

'`\p`' pause sending for a fraction of a second

Some specific types of chat scripts also define additional escape sequences that may appear in the send string. For example, the login chat script defines '`\L`' and '`\P`' to send the login name and password, respectively.

A carriage return will be sent at the end of each send string, unless the `\c` escape sequence appears in the string. Note that some UUCP packages use `\b` for break, but here it means backspace.

Echo checking means that after writing each character the program will wait until the character is echoed. Echo checking must be turned on separately for each send string for which it is desired; it will be turned on for characters following `\E` and turned off for characters following `\e`.

`chat-timeout` *number*

The number of seconds to wait for an expect string in the chat script, before timing out and sending the next subsend, or failing the chat script entirely. The default value is 10 for a login chat or 60 for any other type of chat.

`chat-fail` *string*

If the *string* is seen at any time during a chat script, the chat script is aborted. The string may not contain any whitespace characters: escape sequences must be used for them. Multiple `chat-fail` commands may appear in a single chat script. The default is to have none.

This permits a chat script to be quickly aborted if an error string is seen. For example, a script used to dial out on a modem might use the command '`chat-fail BUSY`' to stop the chat script immediately if the string '`BUSY`' was seen.

The `chat-fail` strings are considered in the order they are listed, so if one string is a suffix of another the longer one should be listed first. This affects the error message which will be logged. Of course, if one string is contained within another, but is not a suffix, the smaller string will always be found before the larger string could match.

`chat-seven-bit` *boolean*

If the argument is true, all incoming characters are stripped to seven bits when being compared to the expect string. Otherwise all eight bits are used in the comparison. The default is true, because some Unix systems generate parity bits during the login prompt which must be ignored while running a chat script. This has no effect on any `chat-program`, which must ignore parity by itself if necessary.

`chat-program` *strings*

Specify a program to run before executing the chat script. This program could run its own version of a chat script, or it could do whatever it wants. If both `chat-program` and `chat` are specified, the program is executed first followed by the chat script.

The first argument to the `chat-program` command is the program name to run. The remaining arguments are passed to the program. The following escape sequences are recognized in the arguments:

`\Y` port device name

`\S` port speed

`\\` backslash

Some specific uses of `chat-program` define additional escape sequences.

Arguments other than escape sequences are passed exactly as they appear in the configuration file, except that sequences of whitespace are compressed to a single space character (this exception may be removed in the future).

If the `chat-program` command is not used, no program is run.

On Unix, the standard input and standard output of the program will be attached to the port in use. Anything the program writes to standard error will be written to the UUCP log file. No other file descriptors will be open. If the program does not exit with a status of 0, it will be assumed to have failed. This means that the dialing programs used by some versions of HDB may not be used directly, but you may be able to run them via the `dialHDB` program in the '`contrib`' directory.

The program will be run as the `uucp` user, and the environment will be that of the process that started `uucico`, so care must be taken to maintain security.

No search path is used to find the program; a full file name must be given. If the program is an executable shell script, it will be passed to '`/bin/sh`' even on systems which are unable to execute shell scripts.

Here is a simple example of a chat script that might be used to reset a Hayes compatible modem.

```
chat "" ATZ OK-ATZ-OK
```

The first expect string is '`""`', so it is ignored. The chat script then sends '`ATZ`'. If the modem responds with '`OK`', the chat script finishes. If 60 seconds (the default timeout) pass before seeing '`OK`', the chat script sends another '`ATZ`'. If it then sees '`OK`', the chat script succeeds. Otherwise, the chat script fails.

For a more complex chat script example, see Section 5.7.3.3 [Logging In], page 77.

5.6 The Main Configuration File

The main configuration file is named '`config`'.

Since all the values that may be specified in the main configuration file also have defaults, there need not be a main configuration file at all.

Each command in '`config`' may have a program prefix, which is a separate word appearing at the beginning of the line. The currently supported prefixes are '`uucp`' and '`cu`'. Any command prefixed by '`uucp`' will not be read by the `cu` program. Any command prefixed by '`cu`' will only be read by the `cu` program. For example, to use a list of systems known only to `cu`, list them in a separate file '*file*' and put '`cu sysfile` '*file*'' in '`config`'.

5.6.1 Miscellaneous config File Commands

`nodename` *string*
`hostname` *string*
`uuname` *string*

These keywords are equivalent. They specify the UUCP name of the local host. If there is no configuration file, an appropriate system function will be used to get the host name, if possible.

`spool` *string*

Specify the spool directory. The default is from '`policy.h`'. This is where UUCP files are queued. Status files and various sorts of temporary files are also stored in this directory and subdirectories of it.

`pubdir` *string*

Specify the public directory. The default is from '`policy.h`'. When a file is named using a leading `~/`, it is taken from or to the public directory. Each system may use a separate public directory by using the `pubdir` command in the system configuration file; see Section 5.7.7 [Miscellaneous (sys)], page 87.

`lockdir` *string*

Specify the directory to place lock files in. The default is from '`policy.h`'; see the information in that file. Normally the lock directory should be set correctly in '`policy.h`', and not changed here. However, changing the lock directory is sometimes useful for testing purposes. This only affects lock files for devices and systems; it does not affect certain internal lock files which are stored in the spool directory (see Section 4.4.5 [Spool Lock Files], page 50).

`unknown` *string* ...

The *string* and subsequent arguments are treated as though they appeared in the system file (see Section 5.7 [sys File], page 71). They are used to apply to any unknown systems that may call in, probably to set file transfer permissions and the like. If the `unknown` command is not used, unknown systems are not permitted to call in.

`strip-login` *boolean*

If the argument is true, then, when `uucico` is doing its own login prompting with the '`-e`', '`-l`', or '`-w`' switches, it will strip the parity bit when it reads the login name and password. Otherwise all eight bits will be used when checking the strings against the UUCP password file. The default is true, since some other UUCP packages send parity bits with the login name and password, and few systems use eight bit characters in the password file.

`strip-proto` *boolean*

If the argument is true, then `uucico` will strip the parity bit from incoming UUCP protocol commands. Otherwise all eight bits will be used. This only applies to commands which are not encapsulated in a link layer protocol. The default is true, which should always be correct unless your UUCP system names use eight bit characters.

`max-uuxqts` *number*
: Specify the maximum number of `uuxqt` processes which may run at the same time. Having several `uuxqt` processes running at once can significantly slow down a system, but, since `uuxqt` is automatically started by `uucico`, it can happen quite easily. The default for `max-uuxqts` is 0, which means that there is no limit. If HDB configuration files are being read and the code was compiled without `HAVE_TAYLOR_CONFIG`, then, if the file '`Maxuuxqts`' in the configuration directory contains a readable number, it will be used as the value for `max-uuxqts`.

`run-uuxqt` *string* or *number*
: Specify when `uuxqt` should be run by `uucico`. This may be a positive number, in which case `uucico` will start a `uuxqt` process whenever it receives the given number of execution files from the remote system, and, if necessary, at the end of the call. The argument may also be one of the strings '`once`', '`percall`', or '`never`'. The string '`once`' means that `uucico` will start `uuxqt` once at the end of execution. The string '`percall`' means that `uucico` will start `uuxqt` once per call that it makes (this is only different from `once` when `uucico` is invoked in a way that causes it to make multiple calls, such as when the '`-r1`' option is used without the '`-s`' option). The string '`never`' means that `uucico` will never start `uuxqt`, in which case `uuxqt` should be periodically run via some other mechanism. The default depends upon which type of configuration files are being used; if `HAVE_TAYLOR_CONFIG` is used the default is '`once`', otherwise if `HAVE_HDB_CONFIG` is used the default is '`percall`', and otherwise, for `HAVE_V2_CONFIG`, the default is '`10`'.

`timetable` *string string*
: The `timetable` defines a timetable that may be used in subsequently appearing time strings; see Section 5.4 [Time Strings], page 61. The first string names the timetable entry; the second is a time string.

The following `timetable` commands are predefined. The NonPeak timetable is included for compatibility. It originally described the offpeak hours of Tymnet and Telenet, but both have since changed their schedules.

```
timetable Evening Wk1705-0755,Sa,Su
timetable Night Wk2305-0755,Sa,Su2305-1655
timetable NonPeak Wk1805-0655,Sa,Su
```

If this command does not appear, then, obviously, no additional timetables will be defined.

`v2-files` *boolean*
: If the code was compiled to be able to read V2 configuration files, a false argument to this command will prevent them from being read. This can be useful while testing. The default is true.

`hdb-files` *boolean*

If the code was compiled to be able to read HDB configuration files, a false argument to this command will prevent them from being read. This can be useful while testing. The default is true.

5.6.2 Configuration File Names

`sysfile` *strings*

Specify the system file(s). The default is the file '`sys`' in the directory *newconfigdir*. These files hold information about other systems with which this system communicates; see Section 5.7 [sys File], page 71. Multiple system files may be given on the line, and the `sysfile` command may be repeated; each system file has its own set of defaults.

`portfile` *strings*

Specify the port file(s). The default is the file '`port`' in the directory *newconfigdir*. These files describe ports which are used to call other systems and accept calls from other systems; see Section 5.8 [port File], page 89. No port files need be named at all. Multiple port files may be given on the line, and the `portfile` command may be repeated.

`dialfile` *strings*

Specify the dial file(s). The default is the file '`dial`' in the directory *newconfigdir*. These files describe dialing devices (modems); see Section 5.9 [dial File], page 93. No dial files need be named at all. Multiple dial files may be given on the line, and the `dialfile` command may be repeated.

`dialcodefile` *strings*

Specify the dialcode file(s). The default is the file '`dialcode`' in the directory *newconfigdir*. These files specify dialcodes that may be used when sending phone numbers to a modem. This permits using the same set of phone numbers in different area-codes or with different phone systems, by using dialcodes to specify the calling sequence. When a phone number goes through dialcode translation, the leading alphabetic characters are stripped off. The dialcode files are read line by line, just like any other configuration file, and when a line is found whose first word is the same as the leading characters from the phone number, the second word on the line (which would normally consist of numbers) replaces the dialcode in the phone number. No dialcode file need be used. Multiple dialcode files may be specified on the line, and the `dialcodefile` command may be repeated; all the dialcode files will be read in turn until a dialcode is located.

`callfile` *strings*

Specify the call out login name and password file(s). The default is the file '`call`' in the directory *newconfigdir*. If the call out login name or password for a system are given as `*` (see Section 5.7.3.3 [Logging In], page 77), these files are read to get the real login name or password. Each line in the file(s) has three words: the system name, the login name, and the password. The login name and password may contain escape sequences like those in a chat script expect string (see Section 5.5 [Chat Scripts], page 62). This file is only used when placing calls to remote systems; the password file described under `passwdfile` below is used for incoming calls. The intention of the call out file is to permit the system file to be publically readable; the call out files must obviously be kept secure. These files need not be used. Multiple call out files may be specified on the line, and the `callfile` command may be repeated; all the files will be read in turn until the system is found.

`passwdfile` *strings*

Specify the password file(s) to use for login names when `uucico` is doing its own login prompting, which it does when given the '`-e`', '`-l`' or '`-w`' switches. The default is the file '`passwd`' in the directory *newconfigdir*. Each line in the file(s) has two words: the login name and the password (e.g., `Ufoo foopas`). They may contain escape sequences like those in a chat script expect string (see Section 5.5 [Chat Scripts], page 62). The login name is accepted before the system name is known, so these are independent of which system is calling in; a particular login may be required for a system by using the `called-login` command in the system file (see Section 5.7.4 [Accepting a Call], page 78). These password files are optional, although one must exist if `uucico` is to present its own login prompts.

As a special exception, a colon may be used to separate the login name from the password, and a colon may be used to terminate the password. This means that the login name and password may not contain a colon. This feature, in conjunction with the `HAVE_ENCRYPTED_PASSWORDS` macro in '`policy.h`', permits using a standard Unix '`/etc/passwd`' as a UUCP password file, providing the same set of login names and passwords for both `getty` and `uucico`.

Multiple password files may be specified on the line, and the `passwdfile` command may be repeated; all the files will be read in turn until the login name is found.

5.6.3 Log File Names

`logfile` *string*

Name the log file. The default is from '`policy.h`'. Logging information is written to this file. If `HAVE_HDB_LOGGING` is defined in '`policy.h`', then by default a separate log file is used for each system; using this command to name a log file will cause all the systems to use it.

`statfile` *string*
: Name the statistics file. The default is from '`policy.h`'. Statistical information about file transfers is written to this file.

`debugfile` *string*
: Name the file to which all debugging information is written. The default is from '`policy.h`'. This command is only effective if the code has been compiled to include debugging (this is controlled by the `DEBUG` macro in '`policy.h`'). If debugging is on, messages written to the log file are also written to the debugging file to make it easier to keep the order of actions straight. The debugging file is different from the log file because information such as passwords can appear in it, so it must be not be publically readable.

5.6.4 Debugging Levels

`debug` *string* . . .
: Set the debugging level. This command is only effective if the code has been compiled to include debugging. The default is to have no debugging. The arguments are strings which name the types of debugging to be turned on. The following types of debugging are defined:

'`abnormal`'
: Output debugging messages for abnormal situations, such as recoverable errors.

'`chat`'
: Output debugging messages for chat scripts.

'`handshake`'
: Output debugging messages for the initial handshake.

'`uucp-proto`'
: Output debugging messages for the UUCP session protocol.

'`proto`'
: Output debugging messages for the individual link protocols.

'`port`'
: Output debugging messages for actions on the communication port.

'`config`'
: Output debugging messages while reading the configuration files.

'`spooldir`'
: Output debugging messages for actions in the spool directory.

'`execute`'
: Output debugging messages whenever another program is executed.

'`incoming`'
: List all incoming data in the debugging file.

'`outgoing`'
: List all outgoing data in the debugging file.

'`all`' All of the above.

The debugging level may also be specified as a number. A 1 will set '`chat`' debugging, a 2 will set both '`chat`' and '`handshake`' debugging, and so on down the possibilities. Currently an 11 will turn on all possible debugging, since there are 11 types of debugging messages listed above; more debugging types may be added in the future. The `debug` command may be used several times in the configuration file; every debugging type named will be turned on. When running any of the programs, the '`-x`' switch (actually, for `uulog` it's the '`-X`' switch) may be used to turn on debugging. The argument to the '`-x`' switch is one of the strings listed above, or a number as described above, or a comma separated list of strings (e.g., '`-x chat,handshake`'). The '`-x`' switch may also appear several times on the command line, in which case all named debugging types will be turned on. The '`-x`' debugging is in addition to any debugging specified by the `debug` command; there is no way to cancel debugging information. The debugging level may also be set specifically for calls to or from a specific system with the `debug` command in the system file (see Section 5.7.7 [Miscellaneous (sys)], page 87).

The debugging messages are somewhat idiosyncratic; it may be necessary to refer to the source code for additional information in some cases.

5.7 The System Configuration File

By default there is a single system configuration, named '`sys`' in the directory *newconfigdir*. This may be overridden by the `sysfile` command in the main configuration file; see Section 5.6.2 [Configuration File Names], page 68.

These files describe all remote systems known to the UUCP package.

5.7.1 Defaults and Alternates

The first set of commands in the file, up to the first `system` command, specify defaults to be used for all systems in that file. Each '`sys`' file uses a different set of defaults.

Subsequently, each set of commands from `system` up to the next `system` command describe a particular system. Default values may be overridden for specific systems.

Each system may then have a series of alternate choices to use when calling out or calling in. The first set of commands for a particular system, up to the first `alternate` command, provide the first choice. Subsequently, each set of commands from `alternate` up to the next `alternate` command describe an alternate choice for calling out or calling in.

When a system is called, the commands before the first `alternate` are used to select a phone number, port, and so forth; if the call fails for some reason, the commands between the first `alternate` and the second are used, and so forth. Well, not quite. Actually, each succeeding alternate will only be used if it is different in some relevant way (different phone number, different chat script, etc.). If you want to force the same alternate to be used again (to retry a phone call more than once, for example), enter the phone number (or any other relevant field) again to make it appear different.

The alternates can also be used to give different permissions to an incoming call based on the login name. This will only be done if the first set of commands, before the first `alternate` command, uses the `called-login` command. The list of alternates will be searched, and the first alternate with a matching `called-login` command will be used. If no alternates match, the call will be rejected.

The `alternate` command may also be used in the file-wide defaults (the set of commands before the first `system` command). This might be used to specify a list of ports which are available for all systems (for an example of this, see Section 5.3.3 [Gateway Example], page 58) or to specify permissions based on the login name used by the remote system when it calls in. The first alternate for each system will default to the first alternate for the file-wide defaults (as modified by the commands used before the first `alternate` command for this system), the second alternate for each system to the second alternate for the file-wide defaults (as modified the same way), and so forth. If a system specifies more alternates than the file-wide defaults, the trailing ones will default to the last file-wide default alternate. If a system specifies fewer alternates than the file-wide defaults, the trailing file-wide default alternates will be used unmodified. The `default-alternates` command may be used to modify this behaviour.

This can all get rather confusing, although it's easier to use than to describe concisely; the `uuchk` program may be used to ensure that you are getting what you want.

5.7.2 Naming the System

`system` *string*
: Specify the remote system name. Subsequent commands up to the next `system` command refer to this system.

`alternate` [*string*]
: Start an alternate set of commands (see Section 5.7.1 [Defaults and Alternates], page 72). An optional argument may be used to name the alternate. This name will be recorded in the log file if the alternate is used to call the system. There is no way to name the first alternate (the commands before the first `alternate` command).

`default-alternates` *boolean*
: If the argument is false, any remaining default alternates (from the defaults specified at the top of the current system file) will not be used. The default is true.

`alias` *string*
: Specify an alias for the current system. The alias may be used by local `uucp` and `uux` commands, as well as by the remote system (which can be convenient if a remote system changes its name). The default is to have no aliases.

`myname` *string*
: Specifies a different system name to use when calling the remote system. Also, if `called-login` is used and is not 'ANY', then, when a system logs in with that login name, *string* is used as the local system name. Because the local system name must be determined before the remote system has identified itself, using `myname` and `called-login` together for any system will set the local name for that login; this means that each locally used system name must have a unique login name associated with it. This allows a system to have different names for an external and an internal network. The default is to not use a special local name.

5.7.3 Calling Out

This section describes commands used when placing a call to another system.

5.7.3.1 When to Call

`time` *string* `[`*number*`]`

Specify when the system may be called. The first argument is a time string; see Section 5.4 [Time Strings], page 61. The optional second argument specifies a retry time in minutes. If a call made during a time that matches the time string fails, no more calls are permitted until the retry time has passed. By default an exponentially increasing retry time is used: after each failure the next retry period is longer. A retry time specified in the `time` command is always a fixed amount of time.

The `time` command may appear multiple times in a single alternate, in which case if any time string matches the system may be called. When the `time` command is used for a particular system, any `time` or `timegrade` commands that appeared in the system defaults are ignored.

The default time string is '`Never`'.

`timegrade` *character string* `[`*number*`]`

The *character* specifies a grade. It must be a single letter or digit. The *string* is a time string (see Section 5.4 [Time Strings], page 61). All jobs of grade *character* or higher (where `0 > 9 > A > Z > a > z`) may be run at the specified time. An ordinary `time` command is equivalent to using `timegrade` with a grade of `z`, permitting all jobs. If there are no jobs of a sufficiently high grade according to the time string, the system will not be called. Giving the '`-s`' switch to `uucico` to force it to call a system causes it to assume there is a job of grade `0` waiting to be run.

The optional third argument specifies a retry time in minutes. See the `time` command, above, for more details.

Note that the `timegrade` command serves two purposes: 1) if there is no job of sufficiently high grade the system will not be called, and 2) if the system is called anyway (because the '`-s`' switch was given to `uucico`) only jobs of sufficiently high grade will be transferred. However, if the other system calls in, the `timegrade` commands are ignored, and jobs of any grade may be transferred (but see `call-timegrade` and `called-timegrade`, below). Also, the `timegrade` command will not prevent the other system from transferring any job it chooses, regardless of who placed the call.

The `timegrade` command may appear multiple times without using `alternate`. When the `timegrade` command is used for a particular system, any `time` or `timegrade` commands that appeared in the system defaults are ignored.

If this command does not appear, there are no restrictions on what grade of work may be done at what time.

`max-retries` *number*

Gives the maximum number of times this system may be retried. If this many calls to the system fail, it will be called at most once a day whatever the retry time is. The default is 26.

`success-wait` *number*

A retry time, in seconds, which applies after a successful call. This can be used to put a limit on how frequently the system is called. For example, an argument of 1800 means that the system will not be called more than once every half hour. The default is 0, which means that there is no limit.

`call-timegrade` *character string*

The *character* is a single character `A` to `Z`, `a` to `z`, or `0` to `9` and specifies a grade. The *string* is a time string (see Section 5.4 [Time Strings], page 61). If a call is placed to the other system during a time which matches the time string, the remote system will be requested to only run jobs of grade *character* or higher. Unfortunately, there is no way to guarantee that the other system will obey the request (this UUCP package will, but there are others which will not); moreover, job grades are historically somewhat arbitrary, so specifying a grade will only be meaningful if the other system cooperates in assigning grades. This grade restriction only applies when the other system is called, not when the other system calls in.

The `call-timegrade` command may appear multiple times without using `alternate`. If this command does not appear, or if none of the time strings match, the remote system will be allowed to send whatever grades of work it chooses.

`called-timegrade` *character string*

The *character* is a single character `A` to `Z`, `a` to `z`, or `0` to `9` and specifies a grade. The *string* is a time string (see Section 5.4 [Time Strings], page 61). If a call is received from the other system during a time which matches the time string, only jobs of grade *character* or higher will be sent to the remote system. This allows the job grade to be set for incoming calls, overriding any request made by the remote uucico. As noted above, job grades are historically somewhat arbitrary, so specifying a grade will only be meaningful if the other system cooperates in assigning grades. This grade restriction only applies to jobs on the local system; it does not affect the jobs transferred by the remote system. This grade restriction only applies when the other system calls in, not when the other system is called.

The `called-timegrade` command may appear multiple times. If this command does not appear, or if none of the time strings match, any grade may be sent to the remote system upon receiving a call.

5.7.3.2 Placing the Call

`speed` *number*

`baud` *number*

Specify the speed (the term *baud* is technically incorrect, but widely understood) at which to call the system. This will try all available ports with that speed until an unlocked port is found. The ports are defined in the port file. If both `speed` and `port` commands appear, both are used when selecting a port. To allow calls at more than one speed, the `alternate` command must be used (see Section 5.7.1 [Defaults and Alternates], page 72). If this command does not appear, there is no default; the speed may be specified in the port file, but if it is not then the natural speed of the port will be used (whatever that means on the system). Specifying an explicit speed of 0 will request the natural speed of the port (whatever the system sets it to), overriding any default speed from the defaults at the top of the file.

`port` *string*

Name a particular port or type of port to use when calling the system. The information for this port is obtained from the port file. If this command does not appear, there is no default; a port must somehow be specified in order to call out (it may be specified implicitly using the `speed` command or explicitly using the next version of `port`). There may be many ports with the same name; each will be tried in turn until an unlocked one is found which matches the desired speed.

`port` *string* . . .

If more than one string follows the `port` command, the strings are treated as a command that might appear in the port file (see Section 5.8 [port File], page 89). If a port is named (by using a single string following `port`) these commands are ignored; their purpose is to permit defining the port completely in the system file rather than always requiring entries in two different files. In order to call out, a port must be specified using some version of the `port` command, or by using the `speed` command to select ports from the port file.

`phone` *string*

`address` *string*

Give a phone number to call (when using a modem port) or a remote host to contact (when using a TCP or TLI port). The commands `phone` and `address` are equivalent; the duplication is intended to provide a mnemonic choice depending on the type of port in use.

When used with a modem port, an `=` character in the phone number means to wait for a secondary dial tone (although only some modems support this); a `-` character means to pause while dialing for 1 second (again, only some modems support this). If the system has more than one phone number, each one must appear in a different alternate. The `phone` command must appear in order to call out on a modem; there is no default.

When used with a TCP port, the string names the host to contact. It may be a domain name or a numeric Internet address. If no address is specified, the system name is used.

When used with a TLI port, the string is treated as though it were an expect string in a chat script, allowing the use of escape characters (see Section 5.5 [Chat Scripts], page 62). The `dialer-sequence` command in the port file may override this address (see Section 5.8 [port File], page 89).

When used with a port that not a modem or TCP or TLI, this command is ignored.

5.7.3.3 Logging In

`chat` *strings*

`chat-timeout` *number*

`chat-fail` *string*

`chat-seven-bit` *boolean*

`chat-program` *strings*

These commands describe a chat script to use when logging on to a remote system. This login chat script is run after any chat script defined in the '`dial`' file (see Section 5.9 [dial File], page 93). Chat scripts are explained in Section 5.5 [Chat Scripts], page 62.

Two additional escape sequences may be used in send strings.

'`\L`' Send the login name, as set by the `call-login` command.

'`\P`' Send the password, as set by the `call-password` command.

Three additional escape sequences may be used with the `chat-program` command. These are '`\L`' and '`\P`', which become the login name and password, respectively, and '`\Z`', which becomes the name of the system of being called.

The default chat script is:

```
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \L word: \P
```

This will send a carriage return (the `\c` suppresses the additional trailing carriage return that would otherwise be sent) and waits for the string '`ogin:`' (which would be the last part of the '`login:`' prompt supplied by a Unix system). If it doesn't see '`ogin:`', it sends a break and waits for '`ogin:`' again. If it still doesn't see '`ogin:`', it sends another break and waits for '`ogin:`' again. If it still doesn't see '`ogin:`', the chat script aborts and hangs up the phone. If it does see '`ogin:`' at some point, it sends the login name (as specified by the `call-login` command) followed by a carriage return (since all send strings are followed by a carriage return unless `\c` is used) and waits for the string '`word:`' (which would be the last part of the '`Password:`' prompt supplied by a Unix system). If it sees '`word:`', it sends the password and a carriage return, completing the chat script. The program will then enter the handshake phase of the UUCP protocol.

This chat script will work for most systems, so you will only be required to use the `call-login` and `call-password` commands. In fact, in the file-wide defaults you could set defaults of '`call-login *`' and '`call-password *`'; you would then just have to make an entry for each system in the call-out login file.

Some systems seem to flush input after the '`login:`' prompt, so they may need a version of this chat script with a `\d` before the `\L`. When using UUCP over TCP, some servers will not be handle the initial carriage return sent by this chat script; in this case you may have to specify the simple chat script '`ogin: \L word: \P`'.

`call-login` *string*

Specify the login name to send with `\L` in the chat script. If the string is '`*`' (e.g., '`call-login *`') the login name will be fetched from the call out login name and password file (see Section 5.6.2 [Configuration File Names], page 68). The string may contain escape sequences as though it were an expect string in a chat script (see Section 5.5 [Chat Scripts], page 62). There is no default.

`call-password` *string*

Specify the password to send with `\P` in the chat script. If the string is '`*`' (e.g., '`call-password *`') the password will be fetched from the call-out login name and password file (see Section 5.6.2 [Configuration File Names], page 68). The string may contain escape sequences as though it were an expect string in a chat script (see Section 5.5 [Chat Scripts], page 62). There is no default.

5.7.4 Accepting a Call

`called-login` *strings*

The first *string* specifies the login name that the system must use when calling in. If it is '`ANY`' (e.g., '`called-login ANY`') any login name may be used; this is useful to override a file-wide default and to indicate that future alternates may have different login names. Case is significant. The default value is '`ANY`'.

Different alternates (see Section 5.7.1 [Defaults and Alternates], page 72) may use different `called-login` commands, in which case the login name will be used to select which alternate is in effect; this will only work if the first alternate (before the first `alternate` command) uses the `called-login` command.

Additional strings may be specified after the login name; they are a list of which systems are permitted to use this login name. If this feature is used, then normally the login name will only be given in a single `called-login` command. Only systems which appear on the list, or which use an explicit `called-login` command, will be permitted to use that login name. If the same login name is used more than once with a list of systems, all the lists are concatenated together. This feature permits you to restrict a login name to a particular set of systems without requiring you to use the `called-login` command for every single system; you can achieve a similar effect by using a different system file for each permitted login name with an appropriate `called-login` command in the file-wide defaults.

`callback` *boolean*

If *boolean* is true, then when the remote system calls `uucico` will hang up the connection and prepare to call it back. The default is false.

`called-chat` *strings*

`called-chat-timeout` *number*

`called-chat-fail` *string*

`called-chat-seven-bit` *boolean*

`called-chat-program` *strings*

These commands may be used to define a chat script (see Section 5.5 [Chat Scripts], page 62) that is run whenever the local system is called by the system being defined. The chat script defined by the `chat` command (see Section 5.7.3.3 [Logging In], page 77), on the other hand, is used when the remote system is called. This called chat script might be used to set special modem parameters that are appropriate to a particular system. It is run after protocol negotiation is complete, but before the protocol has been started. For additional escape sequence which may be used besides those defined for all chat scripts, see Section 5.7.3.3 [Logging In], page 77. There is no default called chat script. If the called chat script fails, the incoming call will be aborted.

5.7.5 Protocol Selection

`protocol` *string*

Specifies which protocols to use for the other system, and in which order to use them. This would not normally be used. For example, '`protocol tfg`'.

The default depends on the characteristics of the port and the dialer, as specified by the `seven-bit` and `reliable` commands. If neither the port nor the dialer use either of these commands, the default is to assume an eight-bit reliable connection. The commands '`seven-bit true`' or '`reliable false`' might be used in either the port or the dialer to change this. Each protocol has particular requirements that must be met before it will be considered during negotiation with the remote side.

The '`t`' and '`e`' protocols are intended for use over TCP or some other communication path with end to end reliability, as they do no checking of the data at all. They will only be considered on a TCP port which is both reliable and eight bit. For technical details, see Section 6.8 [t Protocol], page 124, and Section 6.9 [e Protocol], page 125.

The '`i`' protocol is a bidirectional protocol. It requires an eight-bit connection. It will run over a half-duplex link, such as Telebit modems in PEP mode, but for efficient use of such a connection you must use the `half-duplex` command (see Section 5.8 [port File], page 89). See Section 6.11 [i Protocol], page 125.

The '`g`' protocol is robust, but requires an eight-bit connection. See Section 6.6 [g Protocol], page 118.

The '`G`' protocol is the System V Release 4 version of the '`g`' protocol. See Section 6.10 [Big G Protocol], page 125.

The '`a`' protocol is a Zmodem like protocol, contributed by Doug Evans. It requires an eight-bit connection, but unlike the '`g`' or '`i`' protocol it will work if certain control characters may not be transmitted.

The '`j`' protocol is a variant of the '`i`' protocol which can avoid certain control characters. The set of characters it avoids can be set by a parameter. While it technically does not require an eight bit connection (it could be configured to avoid all characters with the high bit set) it would be very inefficient to use it over one. It is useful over a eight-bit connection that will not transmit certain control characters. See Section 6.12 [j Protocol], page 129.

The '`f`' protocol is intended for use with X.25 connections; it checksums each file as a whole, so any error causes the entire file to be retransmitted. It requires a reliable connection, but only uses seven-bit transmissions. It is a streaming protocol, so, while it can be used on a serial port, the port must be completely reliable and flow controlled; many aren't. See Section 6.7 [f Protocol], page 123.

The '`v`' protocol is the '`g`' protocol as used by the DOS program UUPC/Extended. It is provided only so that UUPC/Extended users can use it; there is no particular reason to select it. See Section 6.17 [v Protocol], page 134.

The '`y`' protocol is an efficient streaming protocol. It does error checking, but when it detects an error it immediately aborts the connection. This requires a reliable, flow controlled, eight-bit connection. In practice, it is only useful on a connection that is nearly always error-free. Unlike the '`t`' and '`e`' protocols, the connection need not be entirely error-free, so the '`y`' protocol can be used on a serial port. See Section 6.14 [y Protocol], page 131.

The protocols will be considered in the order shown above. This means that if neither the `seven-bit` nor the `reliable` command are used, the '`t`' protocol will be used over a TCP connection and the '`i`' protocol will be used over any other type of connection (subject, of course, to what is supported by the remote system; it may be assumed that all systems support the '`g`' protocol).

Note that currently specifying both '`seven-bit true`' and '`reliable false`' will not match any protocol. If this occurs through a combination of port and dialer specifications, you will have to use the `protocol` command for the system or no protocol will be selected at all (the only reasonable choice would be '`protocol f`').

A protocol list may also be specified for a port (see Section 5.8 [port File], page 89), but, if there is a list for the system, the list for the port is ignored.

`protocol-parameter` *character string . . .*

character is a single character specifying a protocol. The remaining strings are a command specific to that protocol which will be executed if that protocol is used. A typical command is something like '`window 7`'. The particular commands are protocol specific.

The '`i`' protocol supports the following commands, all of which take numeric arguments:

`window` The window size to request the remote system to use. This must be between 1 and 16 inclusive. The default is 16.

`packet-size`

The packet size to request the remote system to use. This must be between 1 and 4095 inclusive. The default is 1024.

`remote-packet-size`

If this is between 1 and 4095 inclusive, the packet size requested by the remote system is ignored, and this is used instead. The default is 0, which means that the remote system's request is honored.

`sync-timeout`

The length of time, in seconds, to wait for a SYNC packet from the remote system. SYNC packets are exchanged when the protocol is started. The default is 10.

`sync-retries`

The number of times to retry sending a SYNC packet before giving up. The default is 6.

`timeout` The length of time, in seconds, to wait for an incoming packet before sending a negative acknowledgement. The default is 10.

`retries` The number of times to retry sending a packet or a negative acknowledgement before giving up and closing the connection. The default is 6.

`errors` The maximum number of errors to permit before closing the connection. The default is 100.

`error-decay`
The rate at which to ignore errors. Each time this many packets are received, the error count is decreased by one, so that a long connection with an occasional error will not exceed the limit set by `errors`. The default is 10.

`ack-frequency`
The number of packets to receive before sending an acknowledgement. The default is half the requested window size, which should provide good performance in most cases.

The '`g`', '`G`' and '`v`' protocols support the following commands, all of which take numeric arguments, except `short-packets` which takes a boolean argument:

`window` The window size to request the remote system to use. This must be between 1 and 7 inclusive. The default is 7.

`packet-size`
The packet size to request the remote system to use. This must be a power of 2 between 32 and 4096 inclusive. The default is 64 for the '`g`' and '`G`' protocols and 1024 for the '`v`' protocol. Many older UUCP packages do not support packet sizes larger than 64, and many others do not support packet sizes larger than 128. Some UUCP packages will even dump core if a larger packet size is requested. The packet size is not a negotiation, and it may be different in each direction. If you request a packet size larger than the remote system supports, you will not be able to send any files.

`startup-retries`
The number of times to retry the initialization sequence. The default is 8.

`init-retries`
The number of times to retry one phase of the initialization sequence (there are three phases). The default is 4.

`init-timeout`
The timeout in seconds for one phase of the initialization sequence. The default is 10.

`retries` The number of times to retry sending either a data packet or a request for the next packet. The default is 6.

`timeout` The timeout in seconds when waiting for either a data packet or an acknowledgement. The default is 10.

`garbage` The number of unrecognized bytes to permit before dropping the connection. This must be larger than the packet size. The default is 10000.

`errors` The number of errors (malformed packets, out of order packets, bad checksums, or packets rejected by the remote system) to permit before dropping the connection. The default is 100.

`error-decay`
The rate at which to ignore errors. Each time this many packets are received, the error count is decreased by one, so that a long connection with an occasional error will not exceed the limit set by `errors`. The default is 10.

`remote-window`
If this is between 1 and 7 inclusive, the window size requested by the remote system is ignored and this is used instead. This can be useful when dealing with some poor UUCP packages. The default is 0, which means that the remote system's request is honored.

`remote-packet-size`
If this is between 32 and 4096 inclusive the packet size requested by the remote system is ignored and this is used instead. There is probably no good reason to use this. The default is 0, which means that the remote system's request is honored.

`short-packets`
If this is true, then the code will optimize by sending shorter packets when there is less data to send. This confuses some UUCP packages, such as System V Release 4 (when using the '`G`' protocol) and Waffle; when connecting to such a package, this parameter must be set to false. The default is true for the '`g`' and '`v`' protocols and false for the '`G`' protocol.

The '`a`' protocol is a Zmodem like protocol contributed by Doug Evans. It supports the following commands, all of which take numeric arguments except for `escape-control`, which takes a boolean argument:

`timeout` Number of seconds to wait for a packet to arrive. The default is 10.

`retries` The number of times to retry sending a packet. The default is 10.

`startup-retries`
The number of times to retry sending the initialization packet. The default is 4.

`garbage` The number of garbage characters to accept before closing the connection. The default is 2400.

`send-window`
: The number of characters that may be sent before waiting for an acknowledgement. The default is 1024.

`escape-control`
: Whether to escape control characters. If this is true, the protocol may be used over a connection which does not transmit certain control characters, such as `XON` or `XOFF`. The connection must still transmit eight bit characters other than control characters. The default is false.

The '`j`' protocol can be used over an eight bit connection that will not transmit certain control characters. It accepts the same protocol parameters that the '`i`' protocol accepts, as well as one more:

`avoid`
: A list of characters to avoid. This is a string which is interpreted as an escape sequence (see Section 5.5 [Chat Scripts], page 62). The protocol does not have a way to avoid printable ASCII characters (byte values from 32 to 126, inclusive); only ASCII control characters and eight-bit characters may be avoided. The default value is '`\021\023`'; these are the characters `XON` and `XOFF`, which many connections use for flow control. If the package is configured to use `HAVE_BSD_TTY`, then on some versions of Unix you may have to avoid '`\377`' as well, due to the way some implementations of the BSD terminal driver handle signals.

The '`f`' protocol is intended for use with error-correcting modems only; it checksums each file as a whole, so any error causes the entire file to be retransmitted. It supports the following commands, both of which take numeric arguments:

`timeout`
: The timeout in seconds before giving up. The default is 120.

`retries`
: How many times to retry sending a file. The default is 2.

The '`t`' and '`e`' protocols are intended for use over TCP or some other communication path with end to end reliability, as they do no checking of the data at all. They both support a single command, which takes a numeric argument:

`timeout`
: The timeout in seconds before giving up. The default is 120.

The '`y`' protocol is a streaming protocol contributed by Jorge Cwik. It supports the following commands, both of which take numeric arguments:

`timeout`
: The timeout in seconds when waiting for a packet. The default is 60.

`packet-size`
: The packet size to use. The default is 1024.

The protocol parameters are reset to their default values after each call.

5.7.6 File Transfer Control

`send-request` *boolean*

The *boolean* determines whether the remote system is permitted to request files from the local system. The default is yes.

`receive-request` *boolean*

The *boolean* determines whether the remote system is permitted to send files to the local system. The default is yes.

`request` *boolean*

A shorthand command, equivalent to specifying both '`send-request` *boolean*' and '`receive-request` *boolean*'.

`call-transfer` *boolean*

The *boolean* is checked when the local system places the call. It determines whether the local system may do file transfers queued up for the remote system. The default is yes.

`called-transfer` *boolean*

The *boolean* is checked when the remote system calls in. It determines whether the local system may do file transfers queued up for the remote system. The default is yes.

`transfer` *boolean*

A shorthand command, equivalent to specifying both '`call-transfer` *boolean*' and '`called-transfer` *boolean*'.

`call-local-size` *number string*

The *string* is a time string (see Section 5.4 [Time Strings], page 61). The *number* is the size in bytes of the largest file that should be transferred at a time matching the time string, if the local system placed the call and the request was made by the local system. This command may appear multiple times in a single alternate. If this command does not appear, or if none of the time strings match, there are no size restrictions.

With all the size control commands, the size of a file from the remote system (as opposed to a file from the local system) will only be checked if the other system is running this package: other UUCP packages will not understand a maximum size request, nor will they provide the size of remote files.

`call-remote-size` *number string*

Specify the size in bytes of the largest file that should be transferred at a given time by remote request, when the local system placed the call. This command may appear multiple times in a single alternate. If this command does not appear, there are no size restrictions.

`called-local-size` *number string*
: Specify the size in bytes of the largest file that should be transferred at a given time by local request, when the remote system placed the call. This command may appear multiple times in a single alternate. If this command does not appear, there are no size restrictions.

`called-remote-size` *number string*
: Specify the size in bytes of the largest file that should be transferred at a given time by remote request, when the remote system placed the call. This command may appear multiple times in a single alternate. If this command does not appear, there are no size restrictions.

`local-send` *strings*
: Specifies that files in the directories named by the *strings* may be sent to the remote system when requested locally (using `uucp` or `uux`). The directories in the list should be separated by whitespace. A '`~`' may be used for the public directory. On a Unix system, this is typically '`/usr/spool/uucppublic`'; the public directory may be set with the `pubdir` command. Here is an example of `local-send`:

```
local-send ~ /usr/spool/ftp/pub
```

Listing a directory allows all files within the directory and all subdirectories to be sent. Directories may be excluded by preceding them with an exclamation point. For example:

```
local-send /usr/ftp !/usr/ftp/private ~
```

means that all files in '`/usr/ftp`' or the public directory may be sent, except those files in '`/usr/ftp/private`'. The list of directories is read from left to right, and the last directory to apply takes effect; this means that directories should be listed from top down. The default is the root directory (i.e., any file at all may be sent by local request).

`remote-send` *strings*
: Specifies that files in the named directories may be sent to the remote system when requested by the remote system. The default is '`~`'.

`local-receive` *strings*
: Specifies that files may be received into the named directories when requested by a local user. The default is '`~`'.

`remote-receive` *strings*
: Specifies that files may be received into the named directories when requested by the remote system. The default is '`~`'. On Unix, the remote system may only request that files be received into directories that are writeable by the world, regardless of how this is set.

`forward-to` *strings*
: Specifies a list of systems to which files may be forwarded. The remote system may forward files through the local system on to any of the systems in this list. The string '`ANY`' may be used to permit forwarding to any system. The default is to not permit forwarding to other systems. Note that if the remote system is permitted to execute the `uucp` command, it effectively has the ability to forward to any system.

`forward-from` *strings*
: Specifies a list of systems from which files may be forwarded. The remote system may request files via the local system from any of the systems in this list. The string '`ANY`' may be used to permit forwarding to any system. The default is to not permit forwarding from other systems. Note that if a remote system is permitted to execute the `uucp` command, it effectively has the ability to request files from any system.

`forward` *strings*
: Equivalent to specifying both '`forward-to` *strings*' and '`forward-from` *strings*'. This would normally be used rather than either of the more specific commands.

5.7.7 Miscellaneous sys File Commands

`sequence` *boolean*
: If *boolean* is true, then conversation sequencing is automatically used for the remote system, so that if somebody manages to spoof as the remote system, it will be detected the next time the remote system actually calls. This is false by default.

`command-path` *strings*
: Specifies the path (a list of whitespace separated directories) to be searched to locate commands to execute. This is only used for commands requested by `uux`, not for chat programs. The default is from '`policy.h`'.

`commands` *strings*
: The list of commands which the remote system is permitted to execute locally. For example: '`commands rnews rmail`'. If the value is '`ALL`' (case significant), all commands may be executed. The default is '`rnews rmail`'.

`free-space` *number*
: Specify the minimum amount of file system space (in bytes) to leave free after receiving a file. If the incoming file will not fit, it will be rejected. This initial rejection will only work when talking to another instance of this package, since older UUCP packages do not provide the file size of incoming files. Also, while a file is being received, `uucico` will periodically check the amount of free space. If it drops below the amount given by the `free-space` command, the file transfer will be aborted. The default amount of

space to leave free is from 'policy.h'. This file space checking may not work on all systems.

`pubdir` *string*
: Specifies the public directory that is used when '~' is specifed in a file transfer or a list of directories. This essentially overrides the public directory specified in the main configuration file for this system only. The default is the public directory specified in the main configuration file (which defaults to a value from 'policy.h').

`debug` *string* . . .
: Set additional debugging for calls to or from the system. This may be used to debug a connection with a specific system. It is particularly useful when debugging incoming calls, since debugging information will be generated whenever the call comes in. See the `debug` command in the main configuration file (see Section 5.6.4 [Debugging Levels], page 70) for more details. The debugging information specified here is in addition to that specified in the main configuration file or on the command line.

`max-remote-debug` *string* . . .
: When the system calls in, it may request that the debugging level be set to a certain value. The `max-remote-debug` command may be used to put a limit on the debugging level which the system may request, to avoid filling up the disk with debugging information. Only the debugging types named in the `max-remote-debug` command may be turned on by the remote system. To prohibit any debugging, use '`max-remote-debug none`'.

5.7.8 Default sys File Values

The following are used as default values for all systems; they can be considered as appearing before the start of the file.

```
time Never
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \L word: \P
chat-timeout 10
callback n
sequence n
request y
transfer y
local-send /
remote-send ~
local-receive ~
remove-receive ~
command-path [ from 'policy.h' ]
commands rnews rmail
max-remote-debug abnormal,chat,handshake
```

5.8 The Port Configuration File

The port files may be used to name and describe ports. By default there is a single port file, named '`port`' in the directory *newconfigdir*. This may be overridden by the `portfile` command in the main configuration file; see Section 5.6.2 [Configuration File Names], page 68.

Any commands in a port file before the first `port` command specify defaults for all ports in the file; however, since the `type` command must appear before all other commands for a port, the defaults are only useful if all ports in the file are of the same type (this restriction may be lifted in a later version). All commands after a `port` command up to the next `port` command then describe that port. There are different types of ports; each type supports its own set of commands. Each command indicates which types of ports support it. There may be many ports with the same name; if a system requests a port by name then each port with that name will be tried until an unlocked one is found.

`port` *string*
: Introduces and names a port.

`type` *string*
: Define the type of port. The default is '`modem`'. If this command appears, it must immediately follow the `port` command. The type defines what commands are subsequently allowed. Currently the types are:

'`modem`'
: For a modem hookup.

'`stdin`'
: For a connection through standard input and standard output, as when `uucico` is run as a login shell.

'`direct`'
: For a direct connection to another system.

'`tcp`'
: For a connection using TCP.

'`tli`'
: For a connection using TLI.

'`pipe`'
: For a connection through a pipe running another program.

`protocol` *string*
: Specify a list of protocols to use for this port. This is just like the corresponding command for a system (see Section 5.7.5 [Protocol Selection], page 79). A protocol list for a system takes precedence over a list for a port.

`protocol-parameter` *character strings* `[ any type ]`
: The same command as the `protocol-parameter` command used for systems (see Section 5.7.5 [Protocol Selection], page 79). This one takes precedence.

`seven-bit` *boolean* `[ any type ]`
: This is only used during protocol negotiation; if the argument is true, it forces the selection of a protocol which works across a seven-bit link. It does not prevent eight bit characters from being transmitted. The default is false.

`reliable` *boolean* `[ any type ]`
: This is only used during protocol negotiation; if the argument is false, it forces the selection of a protocol which works across an unreliable communication link. The default is true. It would be more common to specify this for a dialer rather than a port.

`half-duplex` *boolean* `[ any type ]`
: If the argument is true, it means that the port only supports half-duplex connections. This only affects bidirectional protocols, and causes them to not do bidirectional transfers.

`device` *string* `[ modem, direct and tli only ]`
: Names the device associated with this port. If the device is not named, the port name is taken as the device. Device names are system dependent. On Unix, a modem or direct connection might be something like '`/dev/ttyd0`'; a TLI port might be '`/dev/inet/tcp`'.

`speed` *number* `[modem and direct only ]`

`baud` *number* `[ modem and direct only ]`
: The speed this port runs at. If a system specifies a speed but no port name, then all ports which match the speed will be tried in order. If the speed is not specified here and is not specified by the system, the natural speed of the port will be used by default.

`speed-range` *number number* `[ modem only ]`

`baud-range` *number number* `[ modem only ]`
: Specify a range of speeds this port can run at. The first number is the minimum speed, the second number is the maximum speed. These numbers will be used when matching a system which specifies a desired speed. The simple `speed` (or `baud`) command is still used to determine the speed to run at if the system does not specify a speed. For example, the command '`speed-range 300 19200`' means that the port will match any system which uses a speed from 300 to 19200 baud (and will use the speed specified by the system); this could be combined with '`speed 2400`', which means that when this port is used with a system that does not specify a speed, the port will be used at 2400 baud.

`carrier` *boolean* `[ modem and direct only ]`
: The argument indicates whether the port supports carrier.

  If a modem port does not support carrier, the carrier detect signal will never be required on this port, regardless of what the modem chat script indicates. The default for a modem port is true.

If a direct port supports carrier, the port will be set to expect carrier whenever it is used. The default for a direct port is false.

`hardflow` *boolean* `[ modem and direct only ]`
: The argument indicates whether the port supports hardware flow control. If it does not, hardware flow control will not be turned on for this port. The default is true. Hardware flow control is only supported on some systems.

`dial-device` *string* `[ modem only ]`
: Dialing instructions should be output to the named device, rather than to the normal port device. The default is to output to the normal port device.

`dialer` *string* `[ modem only ]`
: Name a dialer to use. The information is looked up in the dial file. There is no default. Some sort of dialer information must be specified to call out on a modem.

`dialer` *string* `. . . [ modem only ]`
: If more than one string follows the `dialer` command, the strings are treated as a command that might appear in the dial file (see Section 5.9 [dial File], page 93). If a dialer is named (by using the first form of this command, described just above), these commands are ignored. They may be used to specify dialer information directly in simple situations without needing to go to a separate file. There is no default. Some sort of dialer information must be specified to call out on a modem.

`dialer-sequence` *strings* `[ modem or tcp or tli only ]`
: Name a sequence of dialers and tokens (phone numbers) to use. The first argument names a dialer, and the second argument names a token. The third argument names another dialer, and so on. If there are an odd number of arguments, the phone number specified with a `phone` command in the system file is used as the final token. The token is what is used for `\D` or `\T` in the dialer chat script. If the token in this string is `\D`, the system phone number will be used; if it is `\T`, the system phone number will be used after undergoing dialcodes translation. A missing final token is taken as `\D`.

  This command currently does not work if `dial-device` is specified; to handle this correctly will require a more systematic notion of chat scripts. Moreover, the `complete` and `abort` chat scripts, the protocol parameters, and the `carrier` and `dtr-toggle` commands are ignored for all but the first dialer.

  This command basically lets you specify a sequence of chat scripts to use. For example, the first dialer might get you to a local network and the second dialer might describe how to select a machine from the local network. This lets you break your dialing sequence into simple modules, and may make it easier to share dialer entries between machines.

  This command is to only way to use a chat script with a TCP port. This can be useful when using a modem which is accessed via TCP.

When this command is used with a TLI port, then if the first dialer is '`TLI`' or '`TLIS`' the first token is used as the address to connect to. If the first dialer is something else, or if there is no token, the address given by the `address` command is used (see Section 5.7.3.2 [Placing the Call], page 76). Escape sequences in the address are expanded as they are for chat script expect strings (see Section 5.5 [Chat Scripts], page 62). The different between '`TLI`' and '`TLIS`' is that the latter implies the command '`stream true`'. These contortions are all for HDB compatibility. Any subsequent dialers are treated as they are for a TCP port.

`lockname` *string* `[ modem and direct only ]`
: Give the name to use when locking this port. On Unix, this is the name of the file that will be created in the lock directory. It is used as is, so on Unix it should generally start with '`LCK..`'. For example, if a single port were named both '`/dev/ttycu0`' and '`/dev/tty0`' (perhaps with different characteristics keyed on the minor device number), then the command `lockname LCK..ttycu0` could be used to force the latter to use the same lock file name as the former.

`service` *string* `[ tcp only ]`
: Name the TCP port number to use. This may be a number. If not, it will be looked up in '`/etc/services`'. If this is not specified, the string '`uucp`' is looked up in '`/etc/services`'. If it is not found, port number 540 (the standard UUCP-over-TCP port number) will be used.

`push` *strings* `[ tli only ]`
: Give a list of modules to push on to the TLI stream.

`stream` *boolean* `[ tli only ]`
: If this is true, and the `push` command was not used, the '`tirdwr`' module is pushed on to the TLI stream.

`server-address` *string* `[ tli only ]`
: Give the address to use when running as a TLI server. Escape sequences in the address are expanded as they are for chat script expect strings (see Section 5.5 [Chat Scripts], page 62).

The string is passed directly to the TLI `t_bind` function. The value needed may depend upon your particular TLI implementation. Check the manual pages, and, if necessary, try writing some sample programs.

For AT&T 3B2 System V Release 3 using the Wollongong TCP/IP stack, which is probably typical, the format of TLI string is '`SSPPIIII`', where '`SS`' is the service number (for TCP, this is 2), '`PP`' is the TCP port number, and '`IIII`' is the Internet address. For example, to accept a connection from on port 540 from any interface, use '`server-address \x00\x02\x02\x1c\x00\x00\x00\x00`'. To only accept connections from a particular interface, replace the last four digits with the network address of the interface. (Thanks to Paul Pryor for the information in this paragraph).

`command` *strings* `[ pipe only ]`

Give the command, with arguments, to run when using a pipe port type. When a port of this type is used, the command is executed and `uucico` communicates with it over a pipe. This permits `uucico` or `cu` to communicate with another system which can only be reached through some unusual means. A sample use might be '`command /bin/rlogin -E -8 -l` *login system*'. The command is run with the full privileges of UUCP; it is responsible for maintaining security.

5.9 The Dialer Configuration File

The dialer configuration files define dialers. By default there is a single dialer file, named '`dial`' in the directory *newconfigdir*. This may be overridden by the `dialfile` command in the main configuration file; see Section 5.6.2 [Configuration File Names], page 68.

Any commands in the file before the first `dialer` command specify defaults for all the dialers in the file. All commands after a `dialer` command up to the next `dialer` command are associated with the named dialer.

`dialer` *string*

Introduces and names a dialer.

`chat` *strings*

`chat-timeout` *number*

`chat-fail` *string*

`chat-seven-bit` *boolean*

`chat-program` *strings*

Specify a chat script to be used to dial the phone. This chat script is used before the login chat script in the '`sys`' file, if any (see Section 5.7.3.3 [Logging In], page 77). For full details on chat scripts, see Section 5.5 [Chat Scripts], page 62.

The `uucico` daemon will sleep for one second between attempts to dial out on a modem. If your modem requires a longer wait period, you must start your chat script with delays ('`\d`' in a send string).

The chat script will be read from and sent to the port specified by the `dial-device` command for the port, if there is one.

The following escape addition escape sequences may appear in send strings:

`\D` send phone number without dialcode translation

`\T` send phone number with dialcode translation

`\M` do not require carrier

`\m` require carrier (fail if not present)

See the description of the dialcodes file (see Section 5.6.2 [Configuration File Names], page 68) for a description of dialcode translation. If the port does not support carrier, as set by the `carrier` command in the port file, `\M` and `\m` are ignored. If both the port and the dialer support carrier, as set by the `carrier` command in the port file and the `carrier` command in the dialer file, then every chat script implicitly begins with `\M` and ends with `\m`. There is no default chat script for dialers.

The following additional escape sequences may be used in `chat-program`:

`\D` phone number without dialcode translation

`\T` phone number with dialcode translation

If the program changes the port in any way (e.g., sets parity) the changes will be preserved during protocol negotiation, but once the protocol is selected it will change the port settings.

`dialtone` *string*
: A string to output when dialing the phone number which causes the modem to wait for a secondary dial tone. This is used to translate the `=` character in a phone number. The default is a comma.

`pause` *string*
: A string to output when dialing the phone number which causes the modem to wait for 1 second. This is used to translate the `-` character in a phone number. The default is a comma.

`carrier` *boolean*
: An argument of true means that the dialer supports the modem carrier signal. After the phone number is dialed, `uucico` will require that carrier be on. One some systems, it will be able to wait for it. If the argument is false, carrier will not be required. The default is true.

`carrier-wait` *number*
: If the port is supposed to wait for carrier, this may be used to indicate how many seconds to wait. The default is 60 seconds. Only some systems support waiting for carrier.

`dtr-toggle` *boolean boolean*
: If the first argument is true, then DTR is toggled before using the modem. This is only supported on some systems and some ports. The second *boolean* need not be present; if it is, and it is true, the program will sleep for 1 second after toggling DTR. The default is to not toggle DTR.

`complete-chat` *strings*

`complete-chat-timeout` *number*

`complete-chat-fail` *string*

`complete-chat-seven-bit` *boolean*

`complete-chat-program` *strings*

These commands define a chat script (see Section 5.5 [Chat Scripts], page 62) which is run when a call is finished normally. This allows the modem to be reset. There is no default. No additional escape sequences may be used.

`complete` *string*

This is a simple use of `complete-chat`. It is equivalent to `complete-chat ""` *string*; this has the effect of sending *string* to the modem when a call finishes normally.

`abort-chat` *strings*

`abort-chat-timeout` *number*

`abort-chat-fail` *string*

`abort-chat-seven-bit` *boolean*

`abort-chat-program` *strings*

These commands define a chat script (see Section 5.5 [Chat Scripts], page 62) to be run when a call is aborted. They may be used to interrupt and reset the modem. There is no default. No additional escape sequences may be used.

`abort` *string*

This is a simple use of `abort-chat`. It is equivalent to `abort-chat ""` *string*; this has the effect of sending *string* to the modem when a call is aborted.

`protocol-parameter` *character strings*

Set protocol parameters, just like the `protocol-parameter` command in the system configuration file or the port configuration file; see Section 5.7.5 [Protocol Selection], page 79. These parameters take precedence, then those for the port, then those for the system.

`seven-bit` *boolean*

This is only used during protocol negotiation; if it is true, it forces selection of a protocol which works across a seven-bit link. It does not prevent eight bit characters from being transmitted. The default is false. It would be more common to specify this for a port than for a dialer.

`reliable` *boolean*

This is only used during protocol negotiation; if it is false, it forces selection of a protocol which works across an unreliable communication link. The default is true.

`half-duplex` *boolean* `[ any type ]`
: If the argument is true, it means that the dialer only supports half-duplex connections. This only affects bidirectional protocols, and causes them to not do bidirectional transfers.

5.10 UUCP Over TCP

If your system has a Berkeley style socket library, or a System V style TLI interface library, you can compile the code to permit making connections over TCP. Specifying that a system should be reached via TCP is easy, but nonobvious.

5.10.1 Connecting to Another System Over TCP

If you are using the new style configuration files (see Chapter 5 [Configuration Files], page 53), add the line '`port type tcp`' to the entry in the '`sys`' file. By default UUCP will get the port number by looking up '`uucp`' in '`/etc/services`'; if the '`uucp`' service is not defined, port 540 will be used. You can set the port number to use with the command '`port service` *xxx*', where *xxx* can be either a number or a name to look up in '`/etc/services`'. You can specify the address of the remote host with '`address` *a.b.c*'; if you don't give an address, the remote system name will be used. You should give an explicit chat script for the system when you use TCP; the default chat script begins with a carriage return, which will not work with some UUCP TCP servers.

If you are using V2 configuration files, add a line like this to '`L.sys`':

```
sys Any TCP uucp host.domain chat-script
```

This will make an entry for system *sys*, to be called at any time, over TCP, using port number '`uucp`' (as found in '`/etc/services`'; this may be specified as a number), using remote host '*host.domain*', with some chat script.

If you are using HDB configuration files, add a line like this to Systems:

```
sys Any TCP - host.domain chat-script
```

and a line like this to '`Devices`':

```
TCP uucp - -
```

You only need one line in '`Devices`' regardless of how many systems you contact over TCP. This will make an entry for system *sys*, to be called at any time, over TCP, using port number '`uucp`' (as found in '`/etc/services`'; this may be specified as a number), using remote host '*host.domain*', with some chat script.

5.10.2 Running a TCP Server

The `uucico` daemon may be run as a TCP server. To use the default port number, which is a reserved port, `uucico` must be invoked by the superuser (or it must be set user ID to the superuser, but I don't recommend doing that).

You must define a port, either using the port file (see Section 5.8 [port File], page 89), if you are using the new configuration method, or with an entry in '`Devices`' if you are using HDB; there is no way to define a port using V2. If you are using HDB the port must be named '`TCP`'; a line as shown above will suffice. You can then start `uucico` as '`uucico -p TCP`' (after the '`-p`', name the port; in HDB it must be '`TCP`'). This will wait for incoming connections, and fork off a child for each one. Each connection will be prompted with '`login:`' and '`Password:`'; the results will be checked against the UUCP (not the system) password file (see Section 5.6.2 [Configuration File Names], page 68).

Another way to run a UUCP TCP server is to use the BSD `uucpd` program.

Yet another way to run a UUCP TCP server is to use `inetd`. Arrange for `inetd` to start up `uucico` with the '`-l`' switch. This will cause `uucico` to prompt with '`login:`' and '`Password:`' and check the results against the UUCP (not the system) password file (you may want to also use the '`-D`' switch to avoid a fork, which in this case is unnecessary).

5.11 Security

This discussion of UUCP security applies only to Unix. It is a bit cursory; suggestions for improvement are solicited.

UUCP is traditionally not very secure. Taylor UUCP addresses some security issues, but is still far from being a secure system.

If security is very important to you, then you should not permit any external access to your computer, including UUCP. Any opening to the outside world is a potential security risk.

When local users use UUCP to transfer files, Taylor UUCP can do little to secure them from each other. You can allow somewhat increased security by putting the owner of the UUCP programs (normally `uucp`) into a separate group; the use of this is explained in the following paragraphs, which refer to this separate group as `uucp-group`.

When the `uucp` program is invoked to copy a file to a remote system, it will, by default, copy the file into the UUCP spool directory. When the `uux` program is used, the '`-C`' switch must be used to copy the file into the UUCP spool directory. In any case, once the file has been copied into the spool directory, other local users will not be able to access it.

When a file is requested from a remote system, UUCP will only permit it to be placed in a directory which is writable by the requesting user. The directory must also be writable by UUCP. A local user can create a directory with a group of `uucp-group` and set the mode to permit group write access. This will allow the file be requested without permitting it to be viewed by any other user.

There is no provision for security for `uucp` requests (as opposed to `uux` requests) made by a user on a remote system. A file sent over by a remote request may only be placed in a directory which is world writable, and the file will be world readable and writable. This will permit any local user to destroy or replace the contents of the file. A file requested by a remote system must be world readable, and the directory it is in must be world readable. Any local user will be able to examine, although not necessarily modify, the file before it is sent.

There are some security holes and race conditions that apply to the above discussion which I will not elaborate on. They are not hidden from anybody who reads the source code, but they are somewhat technical and difficult (though scarcely impossible) to exploit. Suffice it to say that even under the best of conditions UUCP is not completely secure.

For many sites, security from remote sites is a more important consideration. Fortunately, Taylor UUCP does provide some support in this area.

The greatest security is provided by always dialing out to the other site. This prevents anybody from pretending to be the other site. Of course, only one side of the connection can do this.

If remote dialins must be permitted, then it is best if the dialin line is used only for UUCP. If this is the case, then you should create a call-in password file (see Section 5.6.2 [Configuration File Names], page 68) and let `uucico` do its own login prompting. For example, to let remote sites log in on a port named '`entry`' in the port file (see Section 5.8 [port File], page 89), you might invoke '`uucico -e -p entry`'. This would cause `uucico` to enter an endless loop of login prompts and daemon executions. The advantage of this approach is that even if remote users break into the system by guessing or learning the password, they will only be able to do whatever `uucico` permits them to do. They will not be able to start a shell on your system.

If remote users can dial in and log on to your system, then you have a security hazard more serious than that posed by UUCP. But then, you probably knew that already.

Once your system has connected with the remote UUCP, there is a fair amount of control you can exercise. You can use the `remote-send` and `remote-receive` commands to control the directories the remote UUCP can access. You can use the `request` command to prevent the remote UUCP from making any requests of your system at all; however, if you do this it will not even be able to send you mail or news. If you do permit remote requests, you should be careful to restrict what commands may be executed at the remote system's request. The default is `rmail` and `rnews`, which will suffice for most systems.

If different remote systems call in and they must be granted different privileges (perhaps some systems are within the same organization and some are not) then the `called-login` command should be used for each system to require that they use different login names. Otherwise, it would be simple for a remote system to use the `myname` command and pretend to be a different system. The `sequence` command can be used to detect when one system pretended to be another, but, since the sequence numbers must be reset manually after a failed handshake, this can sometimes be more trouble than it's worth.

6 UUCP Protocol Internals

This chapter describes how the various UUCP protocols work, and discusses some other internal UUCP issues.

This chapter is quite technical. You do not need to understand it, or even read it, in order to use Taylor UUCP. It is intended for people who are interested in how the UUCP code works.

The information in this chapter is posted monthly to the Usenet newsgroups '`comp.mail.uucp`', '`news.answers`', and '`comp.answers`'. The posting is available from any '`news.answers`' archive site, such as '`rtfm.mit.edu`'. If you plan to use this information to write a UUCP program, please make sure you get the most recent version of the posting, in case there have been any corrections.

6.1 UUCP Protocol Sources

> "Unix-to-Unix Copy Program," said PDP-1. "You will never find a more wretched hive of bugs and flamers. We must be cautious."
>
> —DECWars

I took a lot of the information from Jamie E. Hanrahan's paper in the Fall 1990 DECUS Symposium, and from *Managing UUCP and Usenet* by Tim O'Reilly and Grace Todino (with contributions by several other people). The latter includes most of the former, and is published by

```
O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472
```

It is currently in its tenth edition. The ISBN number is '`0-937175-93-5`'.

Some information is originally due to a Usenet article by Chuck Wegrzyn. The information on execution files comes partially from Peter Honeyman. The information on the '`g`' protocol comes partially from a paper by G.L. Chesson of Bell Laboratories, partially from Jamie E. Hanrahan's paper, and partially from source code by John Gilmore. The information on the '`f`' protocol comes from the source code by Piet Berteema. The information on the '`t`' protocol comes from the source code by Rick Adams. The information on the '`e`' protocol comes from a Usenet article by Matthias Urlichs. The information on the '`d`' protocol comes from Jonathan Clark, who also supplied information about QFT. The UUPlus information comes straight from Christopher J. Ambler, of UUPlus Development; it applies to version 1.52 and up of the shareware version of UUPlus Utilities, called FSUUCP 1.52, but referred to in this article as UUPlus.

Although there are few books about UUCP, there are many about networks and protocols in general. I recommend two non-technical books which describe the sorts of things that are available on the network: *The Whole Internet*, by Ed Krol, and *Zen and the Art of the Internet*, by Brendan P. Kehoe. Good technical discussions of networking issues can be found in *Internetworking with TCP/IP*, by Douglas E. Comer and David L. Stevens and in *Design and Validation of Computer Protocols* by Gerard J. Holzmann.

6.2 UUCP Grades

Modern UUCP packages support a priority grade for each command. The grades generally range from `A` (the highest) to `Z` followed by `a` to `z`. Some UUCP packages (including Taylor UUCP) also support `0` to `9` before `A`. Some UUCP packages may permit any ASCII character as a grade.

On Unix, these grades are encoded in the name of the command file created by `uucp` or `uux`. A command file name generally has the form '`C.nnnngssss`' where '`nnnn`' is the remote system name for which the command is queued, '`g`' is a single character grade, and '`ssss`' is a four character sequence number. For example, a command file created for the system '`airs`' at grade '`Z`' might be named '`C.airsZ2551`'.

The remote system name will be truncated to seven characters, to ensure that the command file name will fit in the 14 character file name limit of the traditional Unix file system. UUCP packages which have no other means of distinguishing which command files are intended for which systems thus require all systems they connect to to have names that are unique in the first seven characters. Some UUCP packages use a variant of this format which truncates the system name to six characters. HDB and Taylor UUCP use a different spool directory format, which allows up to fourteen characters to be used for each system name.

The sequence number in the command file name may be a decimal integer, or it may be a hexadecimal integer, or it may contain any alphanumeric character. Different UUCP packages are different. Taylor UUCP uses any alphanumeric character.

UUPlus Utilities (as FSUUCP, a shareware DOS based UUCP and news package) uses up to 8 characters for file names in the spool (this is a DOS file system limitation; actually, with the extension, 11 characters are available, but FSUUCP reserves that for future use). FSUUCP defaults mail to grade 'D', and news to grade 'N', except that when the grade of incoming mail can be determined, that grade is preserved if the mail is forwarded to another system. The default grades may be changed by editing the 'LIB/MAILRC' file for mail, or the 'UUPLUS.CFG' file for news.

UUPC/extended for DOS, OS/2 and Windows NT handles mail at grade 'C', news at grade 'd', and file transfers at grade 'n'. The UUPC/extended UUCP and RMAIL commands accept grades to override the default, the others do not.

I do not know how command grades are handled in other non-Unix UUCP packages.

Modern UUCP packages allow you to restrict file transfer by grade depending on the time of day. Typically this is done with a line in the 'Systems' (or 'L.sys') file like this:

```
airs Any/Z,Any2305-0855 ...
```

This allows grades 'Z' and above to be transferred at any time. Lower grades may only be transferred at night. I believe that this grade restriction applies to local commands as well as to remote commands, but I am not sure. It may only apply if the UUCP package places the call, not if it is called by the remote system.

Taylor UUCP can use the timegrade and call-timegrade commands to achieve the same effect. See Section 5.7.3.1 [When to Call], page 74. It supports the above format when reading 'Systems' or 'L.sys'.

UUPC/extended provides the symmetricgrades option to announce the current grade in effect when calling the remote system.

UUPlus allows specification of the highest grade accepted on a per-call basis with the '-g' option in UUCICO.

This sort of grade restriction is most useful if you know what grades are being used at the remote site. The default grades used depend on the UUCP package. Generally uucp and uux have different defaults. A particular grade can be specified with the '-g' option to uucp or uux. For example, to request execution of 'rnews' on 'airs' with grade 'd', you might use something like

```
uux -gd - airs!rnews < article
```

Uunet queues up mail at grade `C`, but increases the grade based on the size. News is queued at grade `d`, and file transfers at grade `n`. The example above would allow mail (below some large size) to be received at any time, but would only permit news to be transferred at night.

6.3 UUCP Lock Files

This discussion applies only to Unix. I have no idea how UUCP locks ports on other systems.

UUCP creates files to lock serial ports and systems. On most, if not all, systems, these same lock files are also used by `cu` to coordinate access to serial ports. On some systems `getty` also uses these lock files, often under the name `uugetty`.

The lock file normally contains the process ID of the locking process. This makes it easy to determine whether a lock is still valid. The algorithm is to create a temporary file and then link it to the name that must be locked. If the link fails because a file with that name already exists, the existing file is read to get the process ID. If the process still exists, the lock attempt fails. Otherwise the lock file is deleted and the locking algorithm is retried.

Older UUCP packages put the lock files in the main UUCP spool directory, `/usr/spool/uucp`. HDB UUCP generally puts the lock files in a directory of their own, usually `/usr/spool/locks` or `/etc/locks`.

The original UUCP lock file format encodes the process ID as a four byte binary number. The order of the bytes is host-dependent. HDB UUCP stores the process ID as a ten byte ASCII decimal number, with a trailing newline. For example, if process 1570 holds a lock file, it would contain the eleven characters space, space, space, space, space, space, one, five, seven, zero, newline. Some versions of UUCP add a second line indicating which program created the lock (`uucp`, `cu`, or `getty/uugetty`). I have also seen a third type of UUCP lock file which does not contain the process ID at all.

The name of the lock file is traditionally `LCK..` followed by the base name of the device. For example, to lock `/dev/ttyd0` the file `LCK..ttyd0` would be created. On SCO Unix, the lock file name is always forced to lower case even if the device name has upper case letters.

System V Release 4 UUCP names the lock file using the major and minor device numbers rather than the device name. The file is named `LK.`*XXX*`.`*YYY*`.`*ZZZ*, where *XXX*, *YYY* and *ZZZ* are all three digit decimal numbers. *XXX* is the major device number of the device holding the directory holding the device file (e.g., `/dev`). *YYY* is the major device number of the device file itself. *ZZZ* is the minor device number of the device file itself. If `s` holds the result of passing the device to the stat system call (e.g., `stat ("/dev/ttyd0", &s)`), the following line of C code will print out the corresponding lock file name:

```
printf ("LK.%03d.%03d.%03d", major (s.st_dev),
        major (s.st_rdev), minor (s.st_rdev));
```

The advantage of this system is that even if there are several links to the same device, they will all use the same lock file name.

When two or more instances of `uuxqt` are executing, some sort of locking is needed to ensure that a single execution job is only started once. I don't know how most UUCP packages deal with this. Taylor UUCP uses a lock file for each execution job. The name of the lock file is the same as the name of the '`X.*`' file, except that the initial '`X`' is changed to an '`L`'. The lock file holds the process ID as described above.

6.4 Execution File Format

UUCP '`X.*`' files control program execution. They are created by `uux`. They are transferred between systems just like any other file. The `uuxqt` daemon reads them to figure out how to execute the job requested by `uux`.

An '`X.*`' file is simply a text file. The first character of each line is a command, and the remainder of the line supplies arguments. The following commands are defined:

'`C command`'
: This gives the command to execute, including the program and all arguments. For example, '`rmail ian@airs.com`'.

'`U user system`'
: This names the user who requested the command, and the system from which the request came.

'`I standard-input`'
: This names the file from which standard input is taken. If no standard input file is given, the standard input will probably be attached to '`/dev/null`'. If the standard input file is not from the system on which the execution is to occur, it will also appear in an '`F`' command.

'O standard-output [system]'
: This names the standard output file. The optional second argument names the system to which the file should be sent. If there is no second argument, the file should be created on the executing system.

'F required-file [filename-to-use]'
: The 'F' command can appear multiple times. Each 'F' command names a file which must exist before the execution can proceed. This will usually be a file which is transferred from the system on which `uux` was executed, but it can also be a file from the local system or some other system. If the file is not from the local system, then the command will usually name a file in the spool directory. If the optional second argument appears, then the file should be copied to the execution directory under that name. This is necessary for any file other than the standard input file. If the standard input file is not from the local system, it will appear in both an 'F' command and an 'I' command.

'R requestor-address'
: This is the address to which mail about the job should be sent. It is relative to the system named in the 'U' command. If the 'R' command does not appear, then mail is sent to the user named in the 'U' command.

'Z'
: This command takes no arguments. It means that a mail message should be sent if the command failed. This is the default behaviour for most modern UUCP packages, and for them the 'Z' command does not actually do anything.

'N'
: This command takes no arguments. It means that no mail message should be sent, even if the command failed.

'n'
: This command takes no arguments. It means that a mail message should be sent if the command succeeded. Normally a message is sent only if the command failed.

'B'
: This command takes no arguments. It means that the standard input should be returned with any error message. This can be useful in cases where the input would otherwise be lost.

'e'
: This command takes no arguments. It means that the command should be processed with '/bin/sh'. For some packages this is the default anyhow. Most packages will refuse to execute complex commands or commands containing wildcards, because of the security holes this opens.

'E'
: This command takes no arguments. It means that the command should be processed with the `execve` system call. For some packages this is the default anyhow.

'M status-file'
: This command means that instead of mailing a message, the message should be copied to the named file on the system named by the 'U' command.

'# comment'
This command is ignored, as is any other unrecognized command.

Here is an example. Given the following command executed on system test1

```
uux - test2!cat - test2!~ian/bar !qux '>~/gorp'
```

(this is only an example, as most UUCP systems will not permit the cat command to be executed) Taylor UUCP will produce something like the following 'X.' file:

```
U ian test1
F D.test1N003r qux
O /usr/spool/uucppublic test1
F D.test1N003s
I D.test1N003s
C cat - ~ian/bar qux
```

The standard input will be read into a file and then transferred to the file 'D.test1N003s' on system 'test2'. The file 'qux' will be transferred to 'D.test1N003r' on system 'test2'. When the command is executed, the latter file will be copied to the execution directory under the name 'qux'. Note that since the file '~ian/bar' is already on the execution system, no action need be taken for it. The standard output will be collected in a file, then copied to the directory '/usr/spool/uucppublic' on the system 'test1'.

6.5 UUCP Protocol

The UUCP protocol is a conversation between two UUCP packages. A UUCP conversation consists of three parts: an initial handshake, a series of file transfer requests, and a final handshake.

6.5.1 The Initial Handshake

Before the initial handshake, the caller will usually have logged in the called machine and somehow started the UUCP package there. On Unix this is normally done by setting the shell of the login name used to '/usr/lib/uucp/uucico'.

All messages in the initial handshake begin with a ^P (a byte with the octal value '\020') and end with a null byte ('\000'). A few systems end these messages with a line feed character ('\012') instead of a null byte; the examples below assume a null byte is being used.

Some options below are supported by QFT, which stands for Queued File Transfer, and is (or was) an internal Bell Labs version of UUCP.

Taylor UUCP size negotiation was introduced by Taylor UUCP, and is also supported by DOS based UUPlus and Amiga based wUUCP and UUCP-1.17.

The initial handshake goes as follows. It is begun by the called machine.

called: '`\020Shere=hostname\000`'
: The hostname is the UUCP name of the called machine. Older UUCP packages do not output it, and simply send '`\020Shere\000`'.

caller: '`\020Shostname options\000`'
: The hostname is the UUCP name of the calling machine. The following options may appear (or there may be none):

  '`-QSEQ`' Report sequence number for this conversation. The sequence number is stored at both sites, and incremented after each call. If there is a sequence number mismatch, something has gone wrong (somebody may have broken security by pretending to be one of the machines) and the call is denied. If the sequence number changes on one of the machines, perhaps because of an attempted breakin or because a disk backup was restored, the sequence numbers on the two machines must be reconciled manually.

  '`-xLEVEL`' Requests the called system to set its debugging level to the specified value. This is not supported by all systems.

  '`-pGRADE`'
  '`-vgrade=GRADE`'
  Requests the called system to only transfer files of the specified grade or higher. This is not supported by all systems. Some systems support '`-p`', some support '`-vgrade=`'. UUPlus allows either '`-p`' or '`-v`' to be specified on a per-system basis in the '`SYSTEMS`' file ('`gradechar`' option).

  '`-R`' Indicates that the calling UUCP understands how to restart failed file transmissions. Supported only by System V Release 4 UUCP, QFT, and Taylor UUCP.

  '`-ULIMIT`' Reports the ulimit value of the calling UUCP. The limit is specified as a base 16 number in C notation (e.g., '`-U0x1000000`'). This number is the number of 512 byte blocks in the largest file which the calling UUCP can create. The called UUCP may not transfer a file larger than this. Supported only by System V Release 4 UUCP, QFT and UUPlus. UUPlus reports the lesser of the available disk space on the spool directory drive and the ulimit variable in '`UUPLUS.CFG`'. Taylor UUCP understands this option, but does not generate it.

'`-N[NUMBER]`'
: Indicates that the calling UUCP understands the Taylor UUCP size negotiation extension. Not supported by traditional UUCP packages. Supported by UUPlus. The optional number is a bitmask of features supported by the calling UUCP, and is described below.

called: '`\020ROK\000`'

There are actually several possible responses.

'`ROK`'
: The calling UUCP is acceptable, and the handshake proceeds to the protocol negotiation. Some options may also appear; see below.

'`ROKN[NUMBER]`'
: The calling UUCP is acceptable, it specified '`-N`', and the called UUCP also understands the Taylor UUCP size limiting extensions. The optional number is a bitmask of features supported by the called UUCP, and is described below.

'`RLCK`'
: The called UUCP already has a lock for the calling UUCP, which normally indicates the two machines are already communicating.

'`RCB`'
: The called UUCP will call back. This may be used to avoid impostors (but only one machine out of each pair should call back, or no conversation will ever begin).

'`RBADSEQ`'
: The call sequence number is wrong (see the '`-Q`' discussion above).

'`RLOGIN`'
: The calling UUCP is using the wrong login name.

'`RYou are unknown to me`'
: The calling UUCP is not known to the called UUCP, and the called UUCP does not permit connections from unknown systems. Some versions of UUCP just drop the line rather than sending this message.

If the response is '`ROK`', the following options are supported by System V Release 4 UUCP and QFT.

'`-R`'
: The called UUCP knows how to restart failed file transmissions.

'`-ULIMIT`'
: Reports the ulimit value of the called UUCP. The limit is specified as a base 16 number in C notation. This number is the number of 512 byte blocks in the largest file which the called UUCP can create. The calling UUCP may not send a file larger than this. Also supported by UUPlus. Taylor UUCP understands this option, but does not generate it.

‘-xLEVEL’ I’m not sure just what this means. It may request the calling UUCP to set its debugging level to the specified value.

If the response is not ‘ROK’ (or ‘ROKN’) both sides hang up the phone, abandoning the call.

called: ‘\020Pprotocols\000’
: Note that the called UUCP outputs two strings in a row. The protocols string is a list of UUCP protocols supported by the caller. Each UUCP protocol has a single character name. These protocols are discussed in more detail later in this document. For example, the called UUCP might send ‘\020Pgf\000’.

caller: ‘\020Uprotocol\000’
: The calling UUCP selects which protocol to use out of the protocols offered by the called UUCP. If there are no mutually supported protocols, the calling UUCP sends ‘\020UN\000’ and both sides hang up the phone. Otherwise the calling UUCP sends something like ‘\020Ug\000’.

Most UUCP packages will consider each locally supported protocol in turn and select the first one supported by the called UUCP. With some versions of HDB UUCP, this can be modified by giving a list of protocols after the device name in the ‘Devices’ file or the ‘Systems’ file. For example, to select the ‘e’ protocol in ‘Systems’,

```
airs Any ACU,e ...
```

or in Devices,

```
ACU,e ttyXX ...
```

Taylor UUCP provides the protocol command which may be used either for a system (see Section 5.7.5 [Protocol Selection], page 79) or a port (see Section 5.8 [port File], page 89). UUPlus allows specification of the protocol string on a per-system basis in the ‘SYSTEMS’ file.

The optional number following a ‘-N’ sent by the calling system, or an ‘ROKN’ sent by the called system, is a bitmask of features supported by the UUCP package. The optional number was introduced in Taylor UUCP version 1.04. The number is sent as an octal number with a leading zero. The following bits are currently defined. A missing number should be taken as ‘011’.

‘01’ UUCP supports size negotiation.

‘02’ UUCP supports file restart.

‘04’ UUCP supports the ‘E’ command.

'010' UUCP requires the file size in the 'S' and 'R' commands to be in base 10. This bit is used by default if no number appears, but should not be explicitly sent.

'020' UUCP expects a dummy string between the notify field and the size field in an 'S' command. This is true of SVR4 UUCP. This bit should not be used.

After the protocol has been selected and the initial handshake has been completed, both sides turn on the selected protocol. For some protocols (notably 'g') a further handshake is done at this point.

6.5.2 UUCP Protocol Commands

Each protocol supports a method for sending a command to the remote system. This method is used to transmit a series of commands between the two UUCP packages. At all times, one package is the master and the other is the slave. Initially, the calling UUCP is the master.

If a protocol error occurs during the exchange of commands, both sides move immediately to the final handshake.

The master will send one of five commands: 'S', 'R', 'X', 'E', or 'H'.

Any file name referred to below is either an absolute file name beginning with '/', a public directory file name beginning with '~/', a file name relative to a user's home directory beginning with '~*USER*/', or a spool directory file name. File names in the spool directory are not absolute, but instead are converted to file names within the spool directory by UUCP. They always begin with 'C.' (for a command file created by uucp or uux), 'D.' (for a data file created by uucp, uux or by an execution, or received from another system for an execution), or 'X.' (for an execution file created by uux or received from another system).

6.5.2.1 The S Command

master: 'S *from to user* -*options temp mode notify size*'

The 'S' and the '-' are literal characters. This is a request by the master to send a file to the slave.

from The name of the file to send. If the 'C' option does not appear in *options*, the master will actually open and send this file. Otherwise the file has been copied to the spool directory, where it is named *temp*. The slave ignores this field unless *to* is a directory, in which case the basename of *from* will be used as the file name. If *from* is a spool directory filename, it must be a data file created for or by an execution, and must begin with '`D.`'.

to The name to give the file on the slave. If this field names a directory the file is placed within that directory with the basename of *from*. A name ending in '`/`' is taken to be a directory even if one does not already exist with that name. If *to* begins with '`X.`', an execution file will be created on the slave. Otherwise, if *to* begins with '`D.`' it names a data file to be used by some execution file. Otherwise, *to* should not be in the spool directory.

user The name of the user who requested the transfer.

options A list of options to control the transfer. The following options are defined (all options are single characters):

'`C`' The file has been copied to the spool directory (the master should use *temp* rather than *from*).

'`c`' The file has not been copied to the spool directory (this is the default).

'`d`' The slave should create directories as necessary (this is the default).

'`f`' The slave should not create directories if necessary, but should fail the transfer instead.

'`m`' The master should send mail to *user* when the transfer is complete.

'`n`' The slave should send mail to *notify* when the transfer is complete.

temp If the '`C`' option appears in *options*, this names the file to be sent. Otherwise if *from* is in the spool directory, *temp* is the same as *from*. Otherwise *temp* may be a dummy string, such as '`D.0`'. After the transfer has been succesfully completed, the master will delete the file *temp*.

mode This is an octal number giving the mode of the file on the master. If the file is not in the spool directory, the slave will always create it with mode 0666, except that if (*mode* & 0111) is not zero (the file is executable), the slave will create the file with mode 0777. If the file is in the spool directory, some UUCP packages will use the algorithm above and some will always create the file with mode 0600. This field is ignored by UUPlus, since it is meaningless on DOS; UUPlus uses 0666 for outgoing files.

notify This field may not be present, and in any case is only meaningful if the '`n`' option appears in *options*. If the '`n`' option appears, then, when the transfer is successfully completed, the slave will send mail to *notify*, which must be a legal mailing address on the slave. If a *size* field will appear but the 'n' option does not appear, *notify* will always be present, typically as the string 'dummy' or simply a pair of double quotes.

size This field is only present when doing Taylor UUCP or SVR4 UUCP size negotiation. It is the size of the file in bytes. Taylor UUCP version 1.03 sends the size as a decimal integer, while versions 1.04 and up, and all other UUCP packages that support size negotiation, send the size in base 16 with a leading 0x.

The slave then responds with an 'S' command response.

'SY *start*' The slave is willing to accept the file, and file transfer begins. The *start* field will only be present when using file restart. It specifies the byte offset into the file at which to start sending. If this is a new file, *start* will be 0x0.

'SN2' The slave denies permission to transfer the file. This can mean that the destination directory may not be accessed, or that no requests are permitted. It implies that the file transfer will never succeed.

'SN4' The slave is unable to create the necessary temporary file. This implies that the file transfer might succeed later.

'SN6' This is only used by Taylor UUCP size negotiation. It means that the slave considers the file too large to transfer at the moment, but it may be possible to transfer it at some other time.

'SN7' This is only used by Taylor UUCP size negotiation. It means that the slave considers the file too large to ever transfer.

'SN8' This is only used by Taylor UUCP. It means that the file was already received in a previous conversation. This can happen if the receive acknowledgement was lost after it was sent by the receiver but before it was received by the sender.

'SN9' This is only used by Taylor UUCP (versions 1.05 and up) and UUPlus (versions 2.0 and up). It means that the remote system was unable to open another channel (see the discussion of the 'i' protocol for more information about channels). This implies that the file transfer might succeed later.

'SN10' This is reportedly used by SVR4 UUCP to mean that the file size is too large.

If the slave responds with 'SY', a file transfer begins. When the file transfer is complete, the slave sends a 'C' command response.

`CY` The file transfer was successful.

`CYM` The file transfer was successful, and the slave wishes to become the master; the master should send an `H` command, described below.

`CN5` The temporary file could not be moved into the final location. This implies that the file transfer will never succeed.

After the `C` command response has been received (in the `SY` case) or immediately (in an `SN` case) the master will send another command.

6.5.2.2 The R Command

master: `R` *from to user* `-`*options size*

The `R` and the `-` are literal characters. This is a request by the master to receive a file from the slave. I do not know how SVR4 UUCP or QFT implement file transfer restart in this case.

from This is the name of the file on the slave which the master wishes to receive. It must not be in the spool directory, and it may not contain any wildcards.

to This is the name of the file to create on the master. I do not believe that it can be a directory. It may only be in the spool directory if this file is being requested to support an execution either on the master or on some system other than the slave.

user The name of the user who requested the transfer.

options A list of options to control the transfer. The following options are defined (all options are single characters):

`d` The master should create directories as necessary (this is the default).

`f` The master should not create directories if necessary, but should fail the transfer instead.

`m` The master should send mail to *user* when the transfer is complete.

size This only appears if Taylor UUCP size negotiation is being used. It specifies the largest file which the master is prepared to accept (when using SVR4 UUCP or QFT, this was specified in the `-U` option during the initial handshake).

The slave then responds with an '`R`' command response. UUPlus does not support '`R`' requests, and always responds with '`RN2`'.

'`RY` *mode* `[`*size*`]`'
: The slave is willing to send the file, and file transfer begins. The *mode* argument is the octal mode of the file on the slave. The master treats this just as the slave does the *mode* argument in the send command, q.v. I am told that SVR4 UUCP sends a trailing *size* argument. For some versions of BSD UUCP, the *mode* argument may have a trailing '`M`' character (e.g., '`RY 0666M`'). This means that the slave wishes to become the master.

'`RN2`'
: The slave is not willing to send the file, either because it is not permitted or because the file does not exist. This implies that the file request will never succeed.

'`RN6`'
: This is only used by Taylor UUCP size negotiation. It means that the file is too large to send, either because of the size limit specifies by the master or because the slave considers it too large. The file transfer might succeed later, or it might not (this may be cleared up in a later release of Taylor UUCP).

'`RN9`'
: This is only used by Taylor UUCP (versions 1.05 and up) and FSUUCP (versions 1.5 and up). It means that the remote system was unable to open another channel (see the discussion of the '`i`' protocol for more information about channels). This implies that the file transfer might succeed later.

If the slave responds with '`RY`', a file transfer begins. When the file transfer is complete, the master sends a '`C`' command. The slave pretty much ignores this, although it may log it.

'`CY`'
: The file transfer was successful.

'`CN5`'
: The temporary file could not be moved into the final location.

After the '`C`' command response has been sent (in the '`RY`' case) or immediately (in an '`RN`' case) the master will send another command.

6.5.2.3 The X Command

master: '`X` *from to user* `-`*options*'
: The '`X`' and the '`-`' are literal characters. This is a request by the master to, in essence, execute uucp on the slave. The slave should execute '`uucp` *from to*'.

from
: This is the name of the file or files on the slave which the master wishes to transfer. Any wildcards are expanded on the slave. If the master is

requesting that the files be transferred to itself, the request would normally contain wildcard characters, since otherwise an 'R' command would suffice. The master can also use this command to request that the slave transfer files to a third system.

to This is the name of the file or directory to which the files should be transferred. This will normally use a UUCP name. For example, if the master wishes to receive the files itself, it would use '`master!path`'.

user The name of the user who requested the transfer.

options A list of options to control the transfer. It is not clear which, if any, options are supported by most UUCP packages.

The slave then responds with an 'X' command response. FSUUCP does not support 'X' requests, and always responds with 'XN'.

'XY' The request was accepted, and the appropriate file transfer commands have been queued up for later processing.

'XN' The request was denied. No particular reason is given.

In either case, the master will then send another command.

6.5.2.4 The E Command

master: '`E` *from to user* `-`*options temp mode notify size command*'

The 'E' command is only supported by Taylor UUCP 1.04 and up. It is used to make an execution request without requiring a separate 'X.*' file. See Section 6.4 [Execution File Format], page 105. It is only used when the command to be executed requires a single input file which is passed to it as standard input.

All the fields have the same meaning as they do for an 'S' command, except for *options* and *command*.

options A list of options to control the transfer. The following options are defined (all options are single characters):

- 'C' The file has been copied to the spool directory (the master should use *temp* rather than *from*).
- 'c' The file has not been copied to the spool directory (this is the default).
- 'N' No mail message should be sent, even if the command fails. This is the equivalent of the 'N' command in an 'X.*' file.

'Z' A mail message should be sent if the command fails (this is generally the default in any case). This is the equivalent of the 'Z' command in an 'X.*' file.

'R' Mail messages about the execution should be sent to the address in the *notify* field. This is the equivalent of the 'R' command in an 'X.*' file.

'e' The execution should be done with '/bin/sh'. This is the equivalent of the 'e' command in an 'X.*' file.

command The command which should be executed. This is the equivalent of the 'C' command in an 'X.*' file.

The slave then responds with an 'E' command response. These are the same as the 'S' command responses, but the initial character is 'E' rather than 'S'.

If the slave responds with 'EY', the file transfer begins. When the file transfer is complete, the slave sends a 'C' command response, just as for the 'S' command. After a successful file transfer, the slave is responsible for arranging for the command to be executed. The transferred file is passed as standard input, as though it were named in the 'I' and 'F' commands of an 'X.*' file.

After the 'C' command response has been received (in the 'EY' case) or immediately (in an 'EN' case) the master will send another command.

6.5.2.5 The H Command

master: 'H'

This is used by the master to hang up the connection. The slave will respond with an 'H' command response.

'HY' The slave agrees to hang up the connection. In this case the master sends another 'HY' command. In some UUCP packages the slave will then send a third 'HY' command. At this point the protocol is shut down, and the final handshake is begun.

'HN' The slave does not agree to hang up. In this case the master and the slave exchange roles. The next command will be sent by the former slave, which is the new master. The roles may be reversed several times during a single connection.

6.5.3 The Final Handshake

After the protocol has been shut down, the final handshake is performed. This handshake has no real purpose, and some UUCP packages simply drop the connection rather than do it (in fact, some will drop the connection immediately after both sides agree to hangup, without even closing down the protocol).

caller: '\020000000\000'

called: '\0200000000\000'

That is, the calling UUCP sends six '0' characters and the called UUCP replies with seven '0' characters. Some UUCP packages always send six '0' characters.

6.6 UUCP 'g' Protocol

The 'g' protocol is a packet based flow controlled error correcting protocol that requires an eight bit clear connection. It is the original UUCP protocol, and is supported by all UUCP implementations. Many implementations of it are only able to support small window and packet sizes, specifically a window size of 3 and a packet size of 64 bytes, but the protocol itself can support up to a window size of 7 and a packet size of 4096 bytes. Complaints about the inefficiency of the 'g' protocol generally refer to specific implementations, rather than to the correctly implemented protocol.

The 'g' protocol was originally designed for general packet drivers, and thus contains some features that are not used by UUCP, including an alternate data channel and the ability to renegotiate packet and window sizes during the communication session.

The 'g' protocol is spoofed by many Telebit modems. When spoofing is in effect, each Telebit modem uses the 'g' protocol to communicate with the attached computer, but the data between the modems is sent using a Telebit proprietary error correcting protocol. This allows for very high throughput over the Telebit connection, which, because it is half-duplex, would not normally be able to handle the 'g' protocol very well at all. When a Telebit is spoofing the 'g' protocol, it forces the packet size to be 64 bytes and the window size to be 3.

This discussion of the 'g' protocol explains how it works, but does not discuss useful error handling techniques. Some discussion of this can be found in Jamie E. Hanrahan's paper, cited above (see Section 6.1 [UUCP Protocol Sources], page 101).

All 'g' protocol communication is done with packets. Each packet begins with a six byte header. Control packets consist only of the header. Data packets contain additional data.

The header is as follows:

'\020' Every packet begins with a ^P.

k (1 <= k <= 9)
: The k value is always 9 for a control packet. For a data packet, the k value indicates how much data follows the six byte header. The amount of data is 2^{k+4}. Thus a k value of 1 means 32 data bytes and a k value of 8 means 4096 data bytes. The k value for a data packet must be between 1 and 8 inclusive.

checksum low byte
checksum high byte
: The checksum value is described below.

control byte
: The control byte indicates the type of packet, and is described below.

xor byte This byte is the xor of k, the checksum low byte, the checksum high byte and the control byte (i.e., the second, third, fourth and fifth header bytes). It is used to ensure that the header data is valid.

The control byte in the header is composed of three bit fields, referred to here as *tt* (two bits), *xxx* (three bits) and *yyy* (three bits). The control is *ttxxxyyy*, or `(`*tt* `<< 6) + (`*xxx* `<< 3) +` *yyy*.

The *TT* field takes on the following values:

'0' This is a control packet. In this case the k byte in the header must be 9. The *xxx* field indicates the type of control packet; these types are described below.

'1' This is an alternate data channel packet. This is not used by UUCP.

'2' This is a data packet, and the entire contents of the attached data field (whose length is given by the k byte in the header) are valid. The *xxx* and *yyy* fields are described below.

'3' This is a short data packet. Let the length of the data field (as given by the k byte in the header) be *l*. Let the first byte in the data field be *b1*. If *b1* is less than 128 (if the most significant bit of *b1* is 0), then there are *l* `-` *b1* valid bytes of data in the data field, beginning with the second byte. If *b1* `>= 128`, let *b2* be the second byte in the data field. Then there are *l* `- ((`*b1* `& 0x7f) + (`*b2* `<< 7))` valid bytes of data in the data field, beginning with the third byte. In all cases *l* bytes of data are sent (and all data bytes participate in the checksum calculation) but some of the trailing bytes may be dropped by the receiver. The *xxx* and *yyy* fields are described below.

In a data packet (short or not) the *xxx* field gives the sequence number of the packet. Thus sequence numbers can range from 0 to 7, inclusive. The *yyy* field gives the sequence number of the last correctly received packet.

Each communication direction uses a window which indicates how many unacknowledged packets may be transmitted before waiting for an acknowledgement. The window may range from 1 to 7, and may be different in each direction. For example, if the window is 3 and the last packet acknowledged was packet number 6, packet numbers 7, 0 and 1 may be sent but the sender must wait for an acknowledgement before sending packet number 2. This acknowledgement could come as the *yyy* field of a data packet, or as the *yyy* field of a '`RJ`' or '`RR`' control packet (described below).

Each packet must be transmitted in order (the sender may not skip sequence numbers). Each packet must be acknowledged, and each packet must be acknowledged in order.

In a control packet, the *xxx* field takes on the following values:

1 '`CLOSE`'
: The connection should be closed immediately. This is typically sent when one side has seen too many errors and wants to give up. It is also sent when shutting down the protocol. If an unexpected '`CLOSE`' packet is received, a '`CLOSE`' packet should be sent in reply and the '`g`' protocol should halt, causing UUCP to enter the final handshake.

2 '`RJ`' or '`NAK`'
: The last packet was not received correctly. The *yyy* field contains the sequence number of the last correctly received packet.

3 '`SRJ`'
: Selective reject. The *yyy* field contains the sequence number of a packet that was not received correctly, and should be retransmitted. This is not used by UUCP, and most implementations will not recognize it.

4 '`RR`' or '`ACK`'
: Packet acknowledgement. The *yyy* field contains the sequence number of the last correctly received packet.

5 '`INITC`'
: Third initialization packet. The *yyy* field contains the maximum window size to use.

6 '`INITB`'
: Second initialization packet. The *yyy* field contains the packet size to use. It requests a size of 2^{yyy+5}. Note that this is not the same coding used for the *k* byte in the packet

header (it is 1 less). Most UUCP implementations that request a packet size larger than 64 bytes can handle any packet size up to that specified.

7 'INITA' First initialization packet. The *yyy* field contains the maximum window size to use.

To compute the checksum, call the control byte (the fifth byte in the header) *c*.

The checksum of a control packet is simply `0xaaaa -` *c*.

The checksum of a data packet is `0xaaaa - (`*check* `^` *c*`)`, where `^` denotes exclusive or, and *check* is the result of the following routine as run on the contents of the data field (every byte in the data field participates in the checksum, even for a short data packet). Below is the routine used by an early version of Taylor UUCP; it is a slightly modified version of a routine which John Gilmore patched from G.L. Chesson's original paper. The `z` argument points to the data and the `c` argument indicates how much data there is.

```
int
igchecksum (z, c)
     register const char *z;
     register int c;
{
  register unsigned int ichk1, ichk2;

  ichk1 = 0xffff;
  ichk2 = 0;

  do
    {
      register unsigned int b;

      /* Rotate ichk1 left.  */
      if ((ichk1 & 0x8000) == 0)
        ichk1 <<= 1;
      else
        {
          ichk1 <<= 1;
          ++ichk1;
        }

      /* Add the next character to ichk1.  */
      b = *z++ & 0xff;
      ichk1 += b;

      /* Add ichk1 xor the character position in the buffer counting from
         the back to ichk2.  */
      ichk2 += ichk1 ^ c;
```

```
      /* If the character was zero, or adding it to ichk1 caused an
         overflow, xor ichk2 to ichk1.  */
      if (b == 0 || (ichk1 & 0xffff) < b)
        ichk1 ^= ichk2;
    }
  while (--c > 0);

  return ichk1 & 0xffff;
}
```

When the 'g' protocol is started, the calling UUCP sends an 'INITA' control packet with the window size it wishes the called UUCP to use. The called UUCP responds with an 'INITA' packet with the window size it wishes the calling UUCP to use. Pairs of 'INITB' and 'INITC' packets are then similarly exchanged. When these exchanges are completed, the protocol is considered to have been started.

Note that the window and packet sizes are not a negotiation. Each system announces the window and packet size which the other system should use. It is possible that different window and packet sizes will be used in each direction. The protocol works this way on the theory that each system knows how much data it can accept without getting overrun. Therefore, each system tells the other how much data to send before waiting for an acknowledgement.

When a UUCP package transmits a command, it sends one or more data packets. All the data packets will normally be complete, although some UUCP packages may send the last one as a short packet. The command string is sent with a trailing null byte, to let the receiving package know when the command is finished. Some UUCP packages require the last byte of the last packet sent to be null, even if the command ends earlier in the packet. Some packages may require all the trailing bytes in the last packet to be null, but I have not confirmed this.

When a UUCP package sends a file, it will send a sequence of data packets. The end of the file is signalled by a short data packet containing zero valid bytes (it will normally be preceeded by a short data packet containing the last few bytes in the file).

Note that the sequence numbers cover the entire communication session, including both command and file data.

When the protocol is shut down, each UUCP package sends a 'CLOSE' control packet.

6.7 UUCP 'f' Protocol

The 'f' protocol is a seven bit protocol which checksums an entire file at a time. It only uses the characters between '\040' and '\176' (ASCII space and ~) inclusive, as well as the carriage return character. It can be very efficient for transferring text only data, but it is very inefficient at transferring eight bit data (such as compressed news). It is not flow controlled, and the checksum is fairly insecure over large files, so using it over a serial connection requires handshaking (XON/XOFF can be used) and error correcting modems. Some people think it should not be used even under those circumstances.

I believe that the 'f' protocol originated in BSD versions of UUCP. It was originally intended for transmission over X.25 PAD links.

The 'f' protocol has no startup or finish protocol. However, both sides typically sleep for a couple of seconds before starting up, because they switch the terminal into XON/XOFF mode and want to allow the changes to settle before beginning transmission.

When a UUCP package transmits a command, it simply sends a string terminated by a carriage return.

When a UUCP package transmits a file, each byte *b* of the file is translated according to the following table:

```
   0 <= b <=  037: 0172, b + 0100 (0100 to 0137)
 040 <= b <= 0171:       b        ( 040 to 0171)
0172 <= b <= 0177: 0173, b - 0100 ( 072 to  077)
0200 <= b <= 0237: 0174, b - 0100 (0100 to 0137)
0240 <= b <= 0371: 0175, b - 0200 ( 040 to 0171)
0372 <= b <= 0377: 0176, b - 0300 ( 072 to  077)
```

That is, a byte between '\040' and '\171' inclusive is transmitted as is, and all other bytes are prefixed and modified as shown.

When all the file data is sent, a seven byte sequence is sent: two bytes of '\176' followed by four ASCII bytes of the checksum as printed in base 16 followed by a carriage return. For example, if the checksum was 0x1234, this would be sent: '\176\1761234\r'.

The checksum is initialized to 0xffff. For each byte that is sent it is modified as follows (where *b* is the byte before it has been transformed as described above):

```
/* Rotate the checksum left.  */
if ((ichk & 0x8000) == 0)
  ichk <<= 1;
else
  {
    ichk <<= 1;
    ++ichk;
  }

/* Add the next byte into the checksum.  */
ichk += b;
```

When the receiving UUCP sees the checksum, it compares it against its own calculated checksum and replies with a single character followed by a carriage return.

'G' The file was received correctly.

'R' The checksum did not match, and the file should be resent from the beginning.

'Q' The checksum did not match, but too many retries have occurred and the communication session should be abandoned.

The sending UUCP checks the returned character and acts accordingly.

6.8 UUCP 't' Protocol

The 't' protocol is intended for use on links which provide reliable end-to-end connections, such as TCP. It does no error checking or flow control, and requires an eight bit clear channel.

I believe the 't' protocol originated in BSD versions of UUCP.

When a UUCP package transmits a command, it first gets the length of the command string, *c*. It then sends `((c / 512) + 1) * 512` bytes (the smallest multiple of 512 which can hold *c* bytes plus a null byte) consisting of the command string itself followed by trailing null bytes.

When a UUCP package sends a file, it sends it in blocks. Each block contains at most 1024 bytes of data. Each block consists of four bytes containing the amount of data in binary (most significant byte first, the same format as used by the Unix function `htonl`) followed by that amount of data. The end of the file is signalled by a block containing zero bytes of data.

6.9 UUCP 'e' Protocol

The 'e' protocol is similar to the 't' protocol. It does no flow control or error checking and is intended for use over networks providing reliable end-to-end connections, such as TCP.

The 'e' protocol originated in versions of HDB UUCP.

When a UUCP package transmits a command, it simply sends the command as an ASCII string terminated by a null byte.

When a UUCP package transmits a file, it sends the complete size of the file as an ASCII decimal number. The ASCII string is padded out to 20 bytes with null bytes (i.e. if the file is 1000 bytes long, it sends '`1000\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0`'). It then sends the entire file.

6.10 UUCP 'G' Protocol

The 'G' protocol is used by SVR4 UUCP. It is identical to the 'g' protocol, except that it is possible to modify the window and packet sizes. The SVR4 implementation of the 'g' protocol reportedly is fixed at a packet size of 64 and a window size of 7. Supposedly SVR4 chose to implement a new protocol using a new letter to avoid any potential incompatibilities when using different packet or window sizes.

Most implementations of the 'g' protocol that accept packets larger than 64 bytes will also accept packets smaller than whatever they requested in the '`INITB`' packet. The SVR4 'G' implementation is an exception; it will only accept packets of precisely the size it requests in the INITB packet.

6.11 UUCP 'i' Protocol

The 'i' protocol was written by Ian Lance Taylor (who also wrote this manual). It was first used by Taylor UUCP version 1.04.

It is a sliding window packet protocol, like the 'g' protocol, but it supports bidirectional transfers (i.e., file transfers in both directions simultaneously). It requires an eight bit clear connection. Several ideas for the protocol were taken from the paper *A High-Throughput Message Transport System* by P. Lauder. I don't know where the paper was published, but the author's e-mail address is `piers@cs.su.oz.au`. The 'i' protocol does not adopt his main idea, which is to dispense with windows entirely. This is because some links still do require flow control and, more importantly, because using windows sets a limit to the amount of data which the protocol must be able to resend upon request. To reduce the costs of window acknowledgements, the protocol uses a large window and only requires an ack at the halfway point.

Each packet starts with a six byte header, optionally followed by data bytes with a four byte checksum. There are currently five defined packet types (`DATA`, `SYNC`, `ACK`, `NAK`, `SPOS`, `CLOSE`) which are described below. Although any packet type may include data, any data provided with an `ACK`, `NAK` or `CLOSE` packet is ignored.

Every `DATA`, `SPOS` and `CLOSE` packet has a sequence number. The sequence numbers are independent for each side. The first packet sent by each side is always number 1. Each packet is numbered one greater than the previous packet, modulo 32.

Every packet has a local channel number and a remote channel number. For all packets at least one channel number is zero. When a UUCP command is sent to the remote system, it is assigned a non-zero local channel number. All packets associated with that UUCP command sent by the local system are given the selected local channel number. All associated packets sent by the remote system are given the selected number as the remote channel number. This permits each UUCP command to be uniquely identified by the channel number on the originating system, and therefore each UUCP package can associate all file data and UUCP command responses with the appropriate command. This is a requirement for bidirectional UUCP transfers.

The protocol maintains a single global file position, which starts at 0. For each incoming packet, any associated data is considered to occur at the current file position, and the file position is incremented by the amount of data contained. The exception is a packet of type `SPOS`, which is used to change the file position. The reason for keeping track of the file position is described below.

The header is as follows:

`\007`
: Every packet begins with `^G`.

`(`*packet* `<< 3) +` *locchan*
: The five bit packet number combined with the three bit local channel number. `DATA`, `SPOS` and `CLOSE` packets use the packet sequence number for the *packet* field. `NAK` packet types use the *packet* field for the sequence number to be resent. `ACK` and `SYNC` do not use the *packet* field, and generally leave it set to 0. Packets which are not associated with a UUCP command from the local system use a local channel number of 0.

`(`*ack* `<< 3) +` *remchan*
: The five bit packet acknowledgement combined with the three bit remote channel number. The packet acknowledgement is the number of the last packet successfully received; it is used by all packet types. Packets which are not sent in response to a UUCP command from the remote system use a remote channel number of 0.

`(`*type* `<< 5) + (`*caller* `<< 4) +` *len1*
: The three bit packet type combined with the one bit packet direction combined with the upper four bits of the data length. The packet direction bit is always 1 for packets sent by the calling UUCP, and 0 for packets sent by the called UUCP. This prevents confusion caused by echoed packets.

len2
: The lower eight bits of the data length. The twelve bits of data length permit packets ranging in size from 0 to 4095 bytes.

check
: The exclusive or of the second through fifth bytes of the header. This provides an additional check that the header is valid.

If the data length is non-zero, the packet is immediately followed by the specified number of data bytes. The data bytes are followed by a four byte CRC 32 checksum, with the most significant byte first. The CRC is calculated over the contents of the data field.

The defined packet types are as follows:

0 '`DATA`'
: This is a plain data packet.

1 '`SYNC`'
: '`SYNC`' packets are exchanged when the protocol is initialized, and are described further below. '`SYNC`' packets do not carry sequence numbers (that is, the *packet* field is ignored).

2 '`ACK`'
: This is an acknowledgement packet. Since '`DATA`' packets also carry packet acknowledgements, '`ACK`' packets are only used when one side has no data to send. '`ACK`' packets do not carry sequence numbers.

3 '`NAK`'
: This is a negative acknowledgement. This is sent when a packet is received incorrectly, and means that the packet number appearing in the *packet* field must be resent. '`NAK`' packets do not carry sequence numbers (the *packet* field is already used).

4 '`SPOS`'
: This packet changes the file position. The packet contains four bytes of data holding the file position, most significant byte first. The next packet received will be considered to be at the named file position.

5 '`CLOSE`'
: When the protocol is shut down, each side sends a '`CLOSE`' packet. This packet does have a sequence number, which could be used to ensure that all packets were correctly

received (this is not needed by UUCP, however, which uses the higher level 'H' command with an 'HY' response).

When the protocol starts up, both systems send a 'SYNC' packet. The 'SYNC' packet includes at least three bytes of data. The first two bytes are the maximum packet size the remote system should send, most significant byte first. The third byte is the window size the remote system should use. The remote system may send packets of any size up to the maximum. If there is a fourth byte, it is the number of channels the remote system may use (this must be between 1 and 7, inclusive). Additional data bytes may be defined in the future.

The window size is the number of packets that may be sent before a packet is acknowledged. There is no requirement that every packet be acknowledged; any acknowledgement is considered to acknowledge all packets through the number given. In the current implementation, if one side has no data to send, it sends an 'ACK' when half the window is received.

Note that the 'NAK' packet corresponds to the unused 'g' protocol 'SRJ' packet type, rather than to the 'RJ' packet type. When a 'NAK' is received, only the named packet should be resent, not any subsequent packets.

Note that if both sides have data to send, but a packet is lost, it is perfectly reasonable for one side to continue sending packets, all of which will acknowledge the last packet correctly received, while the system whose packet was lost will be unable to send a new packet because the send window will be full. In this circumstance, neither side will time out and one side of the communication will be effectively shut down for a while. Therefore, any system with outstanding unacknowledged packets should arrange to time out and resend a packet even if data is being received.

Commands are sent as a sequence of data packets with a non-zero local channel number. The last data packet for a command includes a trailing null byte (normally a command will fit in a single data packet). Files are sent as a sequence of data packets ending with one of length zero.

The channel numbers permit a more efficient implementation of the UUCP file send command. Rather than send the command and then wait for the 'SY' response before sending the file, the file data is sent beginning immediately after the 'S' command is sent. If an 'SN' response is received, the file send is aborted, and a final data packet of length zero is sent to indicate that the channel number may be reused. If an 'SY' reponse with a file position indicator is received, the file send adjusts to the file position; this is why the protocol maintains a global file position.

Note that the use of channel numbers means that each UUCP system may send commands and file data simultaneously. Moreover, each UUCP system may send multiple files at the same time, using the channel number to disambiguate the data. Sending a file before receiving an acknowledgement for the previous file helps to eliminate the round trip delays inherent in other UUCP protocols.

6.12 UUCP 'j' Protocol

The 'j' protocol is a variant of the 'i' protocol. It was also written by Ian Lance Taylor, and first appeared in Taylor UUCP version 1.04.

The 'j' protocol is a version of the 'i' protocol designed for communication links which intercept a few characters, such as XON or XOFF. It is not efficient to use it on a link which intercepts many characters, such as a seven bit link. The 'j' protocol performs no error correction or detection; that is presumed to be the responsibility of the 'i' protocol.

When the 'j' protocol starts up, each system sends a printable ASCII string indicating which characters it wants to avoid using. The string begins with the ASCII character ^ (octal 136) and ends with the ASCII character ~ (octal 176). After sending this string, each system looks for the corresponding string from the remote system. The strings are composed of escape sequences: '\ooo', where 'o' is an octal digit. For example, sending the string '^\021\023~' means that the ASCII XON and XOFF characters should be avoided. The union of the characters described in both strings (the string which is sent and the string which is received) is the set of characters which must be avoided in this conversation. Avoiding a printable ASCII character (octal 040 to octal 176, inclusive) is not permitted.

After the exchange of characters to avoid, the normal 'i' protocol start up is done, and the rest of the conversation uses the normal 'i' protocol. However, each 'i' protocol packet is wrapped to become a 'j' protocol packet.

Each 'j' protocol packet consists of a seven byte header, followed by data bytes, followed by index bytes, followed by a one byte trailer. The packet header looks like this:

^
: Every packet begins with the ASCII character ^, octal 136.

high
low
: These two characters give the total number of bytes in the packet. Both *high* and *low* are printable ASCII characters. The length of the packet is `(`*high* `- 040) * 0100 + (`*low* `- 040)`, where `040 <=` *high* `< 0177` and `040 <=` *low* `< 0140`. This permits a length of 6079 bytes, but there is a further restriction on packet size described below.

`=` The ASCII character `=`, octal 075.

data-high
data-low These two characters give the total number of data bytes in the packet. The encoding is as described for *high* and *low*. The number of data bytes is the size of the '`i`' protocol packet wrapped inside this '`j`' protocol packet.

`@` The ASCII character `@`, octal 100.

The header is followed by the number of data bytes given in *data-high* and *data-low*. These data bytes are the '`i`' protocol packet which is being wrapped in the '`j`' protocol packet. However, each character in the '`i`' protocol packet which the '`j`' protocol must avoid is transformed into a printable ASCII character (recall that avoiding a printable ASCII character is not permitted). Two index bytes are used for each character which must be transformed.

The index bytes immediately follow the data bytes. The index bytes are created in pairs. Each pair of index bytes encodes the location of a character in the '`i`' protocol packet which was transformed to become a printable ASCII character. Each pair of index bytes also encodes the precise transformation which was performed.

When the sender finds a character which must be avoided, it will transform it using one or two operations. If the character is 0200 or greater, it will subtract 0200. If the resulting character is less than 020, or is equal to 0177, it will xor by 020. The result is a printable ASCII character.

The zero based byte index of the character within the '`i`' protocol packet is determined. This index is turned into a two byte printable ASCII index, *index-high* and *index-low*, such that the index is `(`*index-high* `- 040) * 040 + (`*index-low* `- 040)`. *index-low* is restricted such that `040 <=` *index-low* `< 0100`. *index-high* is not permitted to be 0176, so `040 <=` *index-high* `< 0176`. *index-low* is then modified to encode the transformation:

- If the character transformation only had to subtract 0200, then *index-low* is used as is.
- If the character transformation only had to xor by 020, then 040 is added to *index-low*.
- If both operations had to be performed, then 0100 is added to *index-low*. However, if the value of *index-low* was initially 077, then adding 0100 would result in 0177, which is not a printable ASCII character. For that special case, *index-high* is set to 0176, and *index-low* is set to the original value of *index-high*.

The receiver decodes the index bytes as follows (this is the reverse of the operations performed by the sender, presented here for additional clarity):

- The first byte in the index is *index-high*, and the second is *index-low*.
- If `040 <=` *index-high* `< 0176`, the index refers to the data byte at position `(`*index-high* `- 040) * 040 +` *index-low* `% 040`.
- If `040 <=` *index-low* `< 0100`, then 0200 must be added to indexed byte.
- If `0100 <=` *index-low* `< 0140`, then 020 must be xor'ed to the indexed byte.
- If `0140 <=` *index-low* `< 0177`, then 0200 must be added to the indexed byte, and 020 must be xor'ed to the indexed byte.
- If *index-high* `== 0176`, the index refers to the data byte at position `(`*index-low* `- 040) * 040 + 037`. 0200 must be added to the indexed byte, and 020 must be xor'ed to the indexed byte.

This means the largest '`i`' protocol packet which may be wrapped inside a '`j`' protocol packet is `(0175 - 040) * 040 + (077 - 040) == 3007` bytes.

The final character in a '`j`' protocol packet, following the index bytes, is the ASCII character ~ (octal 176).

The motivation behind using an indexing scheme, rather than escape characters, is to avoid data movement. The sender may simply add a header and a trailer to the '`i`' protocol packet. Once the receiver has loaded the '`j`' protocol packet, it may scan the index bytes, transforming the data bytes, and then pass the data bytes directly on to the '`i`' protocol routine.

6.13 UUCP 'x' Protocol

The '`x`' protocol is used in Europe (and probably elsewhere) with machines that contain an builtin X.25 card and can send eight bit data transparently across X.25 circuits, without interference from the X.28 or X.29 layers. The protocol sends packets of 512 bytes, and relies on a write of zero bytes being read as zero bytes without stopping communication. It first appeared in the original System V UUCP implementation.

6.14 UUCP 'y' Protocol

The '`y`' protocol was developed by Jorge Cwik for use in FX UUCICO, a PC uucico program. It is designed for communication lines which handle error correction and flow control. It requires an eight bit clean connection. It performs error detection, but not error correction: when an error is detected, the line is dropped. It is a streaming protocol, like the '`f`' protocol; there are no packet acknowledgements, so the protocol is efficient over a half-duplex communication line such as PEP.

Every packet contains a six byte header:

sequence low byte
sequence high byte
: A two byte sequence number, in little endian order. The first sequence number is 0. Since the first packet is always a sync packet (described below) the sequence number of the first data packet is always 1. Each system counts sequence numbers independently.

length low byte
length high byte
: A two byte data length, in little endian order. If the high bit of the sixteen bit field is clear, this is the number of data bytes which follow the six byte header. If the high bit is set, there is no data, and the length field is a type of control packet.

checksum low byte
checksum high byte
: A two byte checksum, in little endian order. The checksum is computed over the data bytes. The checksum algorithm is described below. If there are no data bytes, the checksum is sent as 0.

When the protocol starts up, each side must send a sync packet. This is a packet with a normal six byte header followed by data. The sequence number of the sync packet should be 0. Currently at least four bytes of data must be sent with the sync packet. Additional bytes should be ignored. They are defined as follows:

version
: The version number of the protocol. Currently this must be 1. Larger numbers should be ignored; it is the responsibility of the newer version to accommodate the older one.

packet size
: The maximum data length to use divided by 256. This is sent as a single byte. The maximum data length permitted is 32768, which would be sent as 128. Customarily both systems will use the same maximum data length, the lower of the two requested.

flags low byte
flags high byte
: Two bytes of flags. None are currently defined. These bytes should be sent as 0, and ignored by the receiver.

A length field with the high bit set is a control packet. The following control packet types are defined:

0xfffe 'YPKT_ACK'
: Acknowledges correct receipt of a file.

0xfffd 'YPKT_ERR'
: Indicates an incorrect checksum.

0xfffc 'YPKT_BAD'
: Indicates a bad sequence number, an invalid length, or some other error.

If a control packet other than 'YPKT_ACK' is received, the connection is dropped. If a checksum error is detected for a received packet, a 'YPKT_ERR' control packet is sent, and the connection is dropped. If a packet is received out of sequence, a 'YPKT_BAD' control packet is sent, and the connection is dropped.

The checksum is initialized to 0xffff. For each data byte in a packet it is modified as follows (where *b* is the byte before it has been transformed as described above):

```
/* Rotate the checksum left.  */
if ((ichk & 0x8000) == 0)
  ichk <<= 1;
else
  {
    ichk <<= 1;
    ++ichk;
  }

/* Add the next byte into the checksum.  */
ichk += b;
```

This is the same algorithm as that used by the 'f' protocol.

A command is sent as a sequence of data packets followed by a null byte. In the normal case, a command will fit into a single packet. The packet should be exactly the length of the command plus a null byte. If the command is too long, more packets are sent as required.

A file is sent as a sequence of data packets, ending with a zero length packet. The data packets may be of any length greater than zero and less than or equal to the maximum permitted packet size specified in the initial sync packet.

After the zero length packet ending a file transfer has been received, the receiving system sends a 'YPKT_ACK' control packet. The sending system waits for the 'YPKT_ACK' control packet before continuing; this wait should be done with a large timeout, since there may be a considerable amount of data buffered on the communication path.

6.15 UUCP 'd' Protocol

The '`d`' protocol is apparently used for DataKit muxhost (not RS-232) connections. No file size is sent. When a file has been completely transferred, a write of zero bytes is done; this must be read as zero bytes on the other end.

6.16 UUCP 'h' Protocol

The '`h`' protocol is apparently used in some places with HST modems. It does no error checking, and is not that different from the '`t`' protocol. I don't know the details.

6.17 UUCP 'v' Protocol

The '`v`' protocol is used by UUPC/extended, a PC UUCP program. It is simply a version of the '`g`' protocol which supports packets of any size, and also supports sending packets of different sizes during the same conversation. There are many '`g`' protocol implementations which support both, but there are also many which do not. Using '`v`' ensures that everything is supported.

7 Hacking Taylor UUCP

This chapter provides the briefest of guides to the Taylor UUCP source code itself.

7.1 System Dependence

The code is carefully segregated into a system independent portion and a system dependent portion. The system dependent code is in the '`unix`' subdirectory, and also in the file '`sysh.unx`' (also known as '`sysdep.h`').

With the right configuration parameters, the system independent code calls only ANSI C functions. Some of the less common ANSI C functions are also provided in the '`lib`' directory. The replacement function `strtol` in '`lib/strtol.c`' assumes that the characters `A` to `F` and `a` to `f` appear in strictly sequential order. The function `igradecmp` in '`uuconf/grdcmp.c`' assumes that the upper and lower case letters appear in order. Both assumptions are true for ASCII and EBCDIC, but neither is guaranteed by ANSI C. Disregarding these caveats, I believe that the system independent portion of the code is strictly conforming.

That's not too exciting, since all the work is done in the system dependent code. I think that this code can conform to POSIX 1003.1, given the right compilation parameters. I'm a bit less certain about this, though.

The code has been used on a 16 bit segmented system with no function prototypes, so I'm fairly certain that all casts to long and pointers are done when necessary.

7.2 Naming Conventions

I use a modified Hungarian naming convention for my variables and functions. As with all naming conventions, the code is rather opaque if you are not familiar with it, but becomes clear and easy to use with time.

The first character indicates the type of the variable (or function return value). Sometimes additional characters are used. I use the following type prefixes:

'`a`' array; the next character is the type of an element 'b' byte or character

'c' count of something

'e' stdio FILE *

'f' boolean

'i' generic integer

'l' double

'o' file descriptor (as returned by open, creat, etc.)

'p' generic pointer

'q' pointer to structure

's' structure

'u' void (function return values only)

'z' character string

A generic pointer (`p`) is sometimes a `void *`, sometimes a function pointer in which case the prefix is pf, and sometimes a pointer to another type, in which case the next character is the type to which it points (pf is overloaded).

An array of strings (`char *[]`) would be named `az` (array of string). If this array were passed to a function, the function parameter would be named `paz` (pointer to array of string).

Note that the variable name prefixes do not necessarily indicate the type of the variable. For example, a variable prefixed with `i` may be int, long or short. Similarly, a variable prefixed with `b` may be a char or an int; for example, the return value of `getchar` would be caught in an int variable prefixed with `b`.

For a non-local variable (extern or file static), the first character after the type prefix is capitalized.

Most static variables and functions use another letter after the type prefix to indicate which module they come from. This is to help distinguish different names in the debugger. For example, all static functions in '`protg.c`', the '`g`' protocol source code, use a module prefix of '`g`'. This isn't too useful, as a number of modules use a module prefix of '`s`'.

7.3 Patches

I am always grateful for any patches sent in. Much of the flexibility and portability of the code is due to other people. Please do not hesitate to send me any changes you have found necessary or useful.

When sending a patch, please send the output of the Unix `diff` program invoked with the '`-c`' option (if you have the GNU version of `diff`, use the '`-p`' option). Always invoke `diff` with the original file first and the modified file second.

If your `diff` does not support '`-c`' (or you don't have `diff`), send a complete copy of the modified file (if you have just changed a single function, you can just send the new version of the function). In particular, please do not send `diff` output without the '`-c`' option, as it is useless.

If you have made a number of changes, it is very convenient for me if you send each change as a separate mail message. Sometimes I will think that one change is useful but another one is not. If they are in different messages it is much easier for me to apply one but not the other.

I rarely apply the patches directly. Instead I work my way through the hunks and apply each one separately. This ensures that the naming remains consistent, and that I understand all the code.

If you can not follow all these rules, then don't. But if you do, it makes it more likely that I will incorporate your changes. I am not paid for my UUCP work, and my available time is unfortunately very restricted. The package is important to me, and I do what I can, but I can not do all that I would like, much less all that everybody else would like.

Finally, please do not be offended if I do not reply to messages for some time, even a few weeks. I am often behind on my mail, and if I think your message deserves a considered reply I will often put it aside until I have time to deal with it.

8 Acknowledgements

This is a list of people who gave help or suggestions while I was working on the Taylor UUCP project. Appearance on this list does not constitute endorsement of the program, particularly since some of the comments were criticisms. I've probably left some people off, and I apologize for any oversight; it does not mean your contribution was unappreciated.

First of all, I would like to thank the people at Infinity Development Systems (formerly AIRS, which lives on in the domain name) for permitting me to use their computers and 'uunet' access. I would also like to thank Richard Stallman <rms@gnu.ai.mit.edu> for founding the Free Software Foundation, and John Gilmore <gnu@cygnus.com> for writing the initial version of gnuucp which was a direct inspiration for this somewhat larger project. Chip Salzenberg <chip@tct.com> has contributed many patches. François Pinard <pinard@iro.umontreal.ca> tirelessly tested the code and suggested many improvements. He also put together the initial version of this manual. Doug Evans contributed the zmodem protocol. Marc Boucher <marc@CAM.ORG> contributed the code supporting the pipe port type. Jorge Cwik jorge@laser.satlink.net contributed the 'y' protocol code. Finally, Verbus M. Counts <verbus@westmark.com> and Centel Federal Systems, Inc., deserve special thanks, since they actually paid me money to port this code to System III.

In alphabetical order:

```
"Earle F. Ake - SAIC" <ake@Dayton.SAIC.COM>
mra@searchtech.com (Michael Almond)
cambler@zeus.calpoly.edu (Christopher J. Ambler)
Brian W. Antoine <briana@tau-ceti.isc-br.com>
jantypas@soft21.s21.com (John Antypas)
james@bigtex.cactus.org (James Van Artsdalen)
jima@netcom.com (Jim Avera)
nba@sysware.DK (Niels Baggesen)
uunet!hotmomma!sdb (Scott Ballantyne)
Zacharias Beckman <zac@dolphin.com>
mike@mbsun.ann-arbor.mi.us (Mike Bernson)
bob@usixth.sublink.org (Roberto Biancardi)
statsci!scott@coco.ms.washington.edu (Scott Blachowicz)
bag%wood2.cs.kiev.ua@relay.ussr.eu.net (Andrey G Blochintsev)
spider@Orb.Nashua.NH.US (Spider Boardman)
Gregory Bond <gnb@bby.com.au>
Marc Boucher <marc@CAM.ORG>
Ard van Breemen <ard@cstmel.hobby.nl>
dean@coplex.com (Dean Brooks)
jbrow@radical.com (Jim Brownfield)
dave@dlb.com (Dave Buck)
gordon@sneaky.lonestar.org (Gordon Burditt)
dburr@sbphy.physics.ucsb.edu (Donald Burr)
```

mib@gnu.ai.mit.edu (Michael I Bushnell)
Brian Campbell <brianc@quantum.on.ca>
Andrew A. Chernov <ache@astral.msk.su>
jhc@iscp.bellcore.com (Jonathan Clark)
mafc!frank@bach.helios.de (Frank Conrad)
Ed Carp <erc@apple.com>
mpc@mbs.linet.org (Mark Clements)
verbus@westmark.westmark.com (Verbus M. Counts)
cbmvax!snark.thyrsus.com!cowan (John Cowan)
Bob Cunningham <bob@soest.hawaii.edu>
jorge@laser.satlink.net (Jorge Cwik)
kdburg@incoahe.hanse.de (Klaus Dahlenburg)
Damon <d@exnet.co.uk>
celit!billd@UCSD.EDU (Bill Davidson)
hubert@arakis.fdn.org (Hubert Delahaye)
markd@bushwire.apana.org.au (Mark Delany)
Allen Delaney <allen@brc.ubc.ca>
Gerriet M. Denkmann gerriet@hazel.north.de
denny@dakota.alisa.com (Bob Denny)
Drew Derbyshire <ahd@kew.com>
ssd@nevets.oau.org (Steven S. Dick)
gert@greenie.gold.sub.org (Gert Doering)
gemini@geminix.in-berlin.de (Uwe Doering)
Hans-Dieter Doll <hd2@Insel.DE>
deane@deane.teleride.on.ca (Dean Edmonds)
Mark W. Eichin <eichin@cygnus.com>
erik@pdnfido.fidonet.org
Andrew Evans <andrew@airs.com>
dje@cygnus.com (Doug Evans)
Marc Evans <marc@synergytics.com>
Dan Everhart <dan@dyndata.com>
kksys!kegworks!lfahnoe@cs.umn.edu (Larry Fahnoe)
Matthew Farwell <dylan@ibmpcug.co.uk>
fenner@jazz.psu.edu (Bill Fenner)
jaf@inference.com (Jose A. Fernandez)
"David J. Fiander" <golem!david@news.lsuc.on.ca>
Thomas Fischer <batman@olorin.dark.sub.org>
Mister Flash <flash@sam.imash.ras.ru>
louis@marco.de (Ju"rgen Fluk)
erik@eab.retix.com (Erik Forsberg)
andy@scp.caltech.edu (Andy Fyfe)
Lele Gaifax <piggy@idea.sublink.org>
Peter.Galbavy@micromuse.co.uk
hunter@phoenix.pub.uu.oz.au (James Gardiner [hunter])
Terry Gardner <cphpcom!tjg01>
dgilbert@gamiga.guelphnet.dweomer.org (David Gilbert)
ol@infopro.spb.su (Oleg Girko)
jimmy@tokyo07.info.com (Jim Gottlieb)
Benoit Grange <ben@fizz.fdn.org>
elg@elgamy.jpunix.com (Eric Lee Green)

ryan@cs.umb.edu (Daniel R. Guilderson)
greg@gagme.chi.il.us (Gregory Gulik)
Richard H. Gumpertz <rhg@cps.com>
Scott Guthridge <scooter@cube.rain.com>
Michael Haberler <mah@parrot.prv.univie.ac.at>
Daniel Hagerty <hag@eddie.mit.edu>
jh@moon.nbn.com (John Harkin)
guy@auspex.auspex.com (Guy Harris)
hsw1@papa.attmail.com (Stephen Harris)
Petri Helenius <pete@fidata.fi>
gabe@edi.com (B. Gabriel Helou)
Bob Hemedinger <bob@dalek.mwc.com>
Andrew Herbert <andrew@werple.pub.uu.oz.au>
kherron@ms.uky.edu (Kenneth Herron)
Peter Honeyman <honey@citi.umich.edu>
jhood@smoke.marlboro.vt.us (John Hood)
Mike Ipatow <mip@fido.itc.e-burg.su>
Bill Irwin <bill@twg.bc.ca>
pmcgw!personal-media.co.jp!ishikawa (Chiaki Ishikawa)
ai@easy.in-chemnitz.de (Andreas Israel)
iverson@lionheart.com (Tim Iverson)
bei@dogface.austin.tx.us (Bob Izenberg)
djamiga!djjames@fsd.com (D.J.James)
Rob Janssen <cmgit!rob@relay.nluug.nl>
harvee!esj (Eric S Johansson)
Kevin Johnson <kjj@pondscum.phx.mcd.mot.com>
rj@rainbow.in-berlin.de (Robert Joop)
Alan Judge <aj@dec4ie.IEunet.ie>
chris@cj_net.in-berlin.de (Christof Junge)
Romain Kang <romain@pyramid.com>
tron@Veritas.COM (Ronald S. Karr)
Brendan Kehoe <brendan@cs.widener.edu>
warlock@csuchico.edu (John Kennedy)
kersing@nlmug.nl.mugnet.org (Jac Kersing)
ok@daveg.PFM-Mainz.de (Olaf Kirch)
Gabor Kiss <kissg@sztaki.hu>
gero@gkminix.han.de (Gero Kuhlmann)
rob@pact.nl (Rob Kurver)
"C.A. Lademann" <cal@zls.gtn.com>
kent@sparky.IMD.Sterling.COM (Kent Landfield)
Tin Le <tin@saigon.com>
lebaron@inrs-telecom.uquebec.ca (Gregory LeBaron)
karl@sugar.NeoSoft.Com (Karl Lehenbauer)
alex@hal.rhein-main.de (Alexander Lehmann)
merlyn@digibd.com (Merlyn LeRoy)
clewis@ferret.ocunix.on.ca (Chris Lewis)
gdonl@ssi1.com (Don Lewis)
libove@libove.det.dec.com (Jay Vassos-Libove)
bruce%blilly@Broadcast.Sony.COM (Bruce Lilly)
Godfrey van der Linden <Godfrey_van_der_Linden@NeXT.COM>

Ted Lindgreen <tlindgreen@encore.nl>
andrew@cubetech.com (Andrew Loewenstern)
"Arne Ludwig" <arne@rrzbu.hanse.de>
Matthew Lyle <matt@mips.mitek.com>
djm@eng.umd.edu (David J. MacKenzie)
John R MacMillan <chance!john@sq.sq.com>
jum@helios.de (Jens-Uwe Mager)
Giles D Malet <shrdlu!gdm@provar.kwnet.on.ca>
mem@mv.MV.COM (Mark E. Mallett)
pepe@dit.upm.es (Jose A. Manas)
peter@xpoint.ruessel.sub.org (Peter Mandrella)
martelli@cadlab.sublink.org (Alex Martelli)
W Christopher Martin <wcm@geek.ca.geac.com>
Yanek Martinson <yanek@mthvax.cs.miami.edu>
thomasm@mechti.wupper.de (Thomas Mechtersheimer)
jm@aristote.univ-paris8.fr (Jean Mehat)
me@halfab.freiburg.sub.org (Udo Meyer)
les@chinet.chi.il.us (Leslie Mikesell)
bug@cyberdex.cuug.ab.ca (Trever Miller)
mmitchel@digi.lonestar.org (Mitch Mitchell)
Emmanuel Mogenet <mgix@krainte.jpn.thomson-di.fr>
rmohr@infoac.rmi.de (Rupert Mohr)
Jason Molenda <molenda@sequent.com>
ianm@icsbelf.co.uk (Ian Moran)
jmorriso@bogomips.ee.ubc.ca (John Paul Morrison)
brian@ilinx.wimsey.bc.ca (Brian J. Murrell)
service@infohh.rmi.de (Dirk Musstopf)
lyndon@cs.athabascau.ca (Lyndon Nerenberg)
rolf@saans.north.de (Rolf Nerstheimer)
tom@smart.bo.open.de (Thomas Neumann)
mnichols@pacesetter.com
Richard E. Nickle <trystro!rick@Think.COM>
stephan@sunlab.ka.sub.org (Stephan Niemz)
nolan@helios.unl.edu (Michael Nolan)
david nugent <david@csource.oz.au>
Jim O'Connor <jim@bahamut.fsc.com>
kevin%kosman.uucp@nrc.com (Kevin O'Gorman)
Petri Ojala <ojala@funet.fi>
oneill@cs.ulowell.edu (Brian 'Doc' O'Neill)
Stephen.Page@prg.oxford.ac.uk
abekas!dragoman!mikep@decwrl.dec.com (Mike Park)
Tim Peiffer peiffer@cs.umn.edu
don@blkhole.resun.com (Don Phillips)
"Mark Pizzolato 415-369-9366" <mark@infocomm.com>
John Plate <plate@infotek.dk>
dplatt@ntg.com (Dave Platt)
eldorado@tharr.UUCP (Mark Powell)
Mark Powell <mark@inet-uk.co.uk>
pozar@kumr.lns.com (Tim Pozar)
joey@tessi.UUCP (Joey Pruett)

Paul Pryor ptp@fallschurch-acirs2.army.mil
putsch@uicc.com (Jeff Putsch)
ar@nvmr.robin.de (Andreas Raab)
Jarmo Raiha <jarmo@ksvltd.FI>
James Revell <revell@uunet.uu.net>
Scott Reynolds <scott@clmqt.marquette.Mi.US>
mcr@Sandelman.OCUnix.On.Ca (Michael Richardson)
Kenji Rikitake <kenji@rcac.astem.or.jp>
arnold@cc.gatech.edu (Arnold Robbins)
steve@Nyongwa.cam.org (Steve M. Robbins)
Ollivier Robert <Ollivier.Robert@keltia.frmug.fr.net>
Serge Robyns <sr@denkart.be>
Lawrence E. Rosenman <ler@lerami.lerctr.org>
Jeff Ross <jeff@wisdom.bubble.org>
Aleksey P. Rudnev <alex@kiae.su>
"Heiko W.Rupp" <hwr@pilhuhn.ka.sub.org>
wolfgang@wsrcc.com (Wolfgang S. Rupprecht)
tbr@tfic.bc.ca (Tom Rushworth)
jsacco@ssl.com (Joseph E. Sacco)
rsalz@bbn.com (Rich Salz)
sojurn!mike@hobbes.cert.sei.cmu.edu (Mike Sangrey)
Nickolay Saukh <nms@ussr.EU.net>
heiko@lotte.sax.de (Heiko Schlittermann)
Eric Schnoebelen <eric@cirr.com>
russell@alpha3.ersys.edmonton.ab.ca (Russell Schulz)
scott@geom.umn.edu
Igor V. Semenyuk <iga@argrd0.argonaut.su>
Christopher Sawtell <chris@gerty.equinox.gen.nz>
schuler@bds.sub.org (Bernd Schuler)
uunet!gold.sub.org!root (Christian Seyb)
s4mjs!mjs@nirvo.nirvonics.com (M. J. Shannon Jr.)
shields@tembel.org (Michael Shields)
peter@ficc.ferranti.com (Peter da Silva)
vince@victrola.sea.wa.us (Vince Skahan)
frumious!pat (Patrick Smith)
roscom!monty@bu.edu (Monty Solomon)
sommerfeld@orchard.medford.ma.us (Bill Sommerfeld)
Julian Stacey <stacey@guug.de>
evesg@etlrips.etl.go.jp (Gjoen Stein)
Harlan Stenn <harlan@mumps.pfcs.com>
Ralf Stephan <ralf@ark.abg.sub.org>
johannes@titan.westfalen.de (Johannes Stille)
chs@antic.apu.fi (Hannu Strang)
ralf@reswi.ruhr.de (Ralf E. Stranzenbach)
sullivan@Mathcom.com (S. Sullivan)
Shigeya Suzuki <shigeya@dink.foretune.co.jp>
kls@ditka.Chicago.COM (Karl Swartz)
swiers@plains.NoDak.edu
Oleg Tabarovsky <olg@olghome.pccentre.msk.su>
ikeda@honey.misystems.co.jp (Takatoshi Ikeda)

```
John Theus <john@theus.rain.com>
rd@aii.com (Bob Thrush)
ppKarsten Thygesen <karthy@dannug.dk>
Graham Toal <gtoal@pizzabox.demon.co.uk>
rmtodd@servalan.servalan.com (Richard Todd)
Martin Tomes <mt00@controls.eurotherm.co.uk>
Len Tower <tower-prep@ai.mit.edu>
Mark Towfiq <justice!towfiq@Eingedi.Newton.MA.US>
mju@mudos.ann-arbor.mi.us (Marc Unangst)
Matthias Urlichs <urlichs@smurf.noris.de>
Tomi Vainio <tomppa@fidata.fi>
a3@a3.xs4all.nl (Adri Verhoef)
Andrew Vignaux <ajv@ferrari.datamark.co.nz>
vogel@omega.ssw.de (Andreas Vogel)
Dima Volodin <dvv@hq.demos.su>
jos@bull.nl (Jos Vos)
jv@nl.net (Johan Vromans)
David Vrona <dave@sashimi.wwa.com>
Marcel.Waldvogel@nice.usergroup.ethz.ch (Marcel Waldvogel)
steve@nshore.org (Stephen J. Walick)
syd@dsinc.dsi.com (Syd Weinstein)
gerben@rna.indiv.nluug.nl (Gerben Wierda)
jbw@cs.bu.edu (Joe Wells)
frnkmth!twwells.com!bill (T. William Wells)
Peter Wemm <Peter_Wemm@zeus.dialix.oz.au>
mauxci!eci386!woods@apple.com (Greg A. Woods)
John.Woods@proteon.com (John Woods)
Michael Yu.Yaroslavtsev <mike@yaranga.ipmce.su>
Alexei K. Yushin <root@july.elis.crimea.ua>
jon@console.ais.org (Jon Zeeff)
Matthias Zepf <agnus@amylnd.stgt.sub.org>
Eric Ziegast <uunet!ziegast>
```

Concept Index

Configuration File Index

F

H

L

M

N

P

R

S

T

U

V

Table of Contents