Taylor UUCP

Version 1.04

Ian Lance Taylor <ian@airs.com>

Taylor UUCP 1.04

This is the documentation for the Taylor UUCP package, version 1.04. The programs were written by Ian Lance Taylor. The author can be reached at `<ian@airs.com>`, or '`c/o Cygnus Support, 4th Floor, Building 200, 1 Kendall Square, Cambridge, MA, 02139`'.

There is a mailing list for discussion of the package. To join (or get off) the list, send mail to `<taylor-uucp-request@gnu.ai.mit.edu>`. Mail to this address is answered by a person, not a program. When joining the list, please give the address at which you wish to receive mail; do not rely on the message headers. To send a message to the list, send it to `<taylor-uucp@gnu.ai.mit.edu>`.

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.

Here is some propaganda from the Free Software Foundation. If you find this stuff offensive or annoying, remember that you probably did not spend any money to get this code. I did not write this code to make life easier for developers of UUCP packages, I wrote it to help end users, and I believe that these are the most appropriate conditions for distribution.

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`'.

`uux`

The `uux` program is used to request a program to be executed 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.

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.

`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.

`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.

`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`.

`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.

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.

`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.

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 to run, 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.

uuconv

The uuconv program can be used to convert UUCP configuration files from one support format to another. This can be useful for administrators converting from an older UUCP. Taylor UUCP is able to read and use old configuration file formats, but some new features can not be selected using the old formats.

uusched

The `uusched` script is just 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.

`tstuu`

The `tstuu` program is a test harness for the UUCP package, which will help ensure that it has been configured and compiled correctly. It does not test everything, however. It only runs on Unix systems which support Berkeley style pseudo-terminals or STREAMS style pseudo-terminals. It can be useful when initially installing Taylor UUCP.

2 Taylor UUCP Overall Installation

These are the installation instructions for the Taylor UUCP package.

2.1 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 3 [Configuration Files], page 19. 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 dialer file (see Section 3.8 [dial File], page 55), 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 dialer file; see Section 3.8 [dial File], page 55. You will have to invoke the dialer program via a shell script, since an exit code of 0 is required to recognize success.

The `uuconv` 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 3 [Configuration Files], page 19.

You must also decide what sort of spool directory you want to use. If you will only be using these programs, I recommend `SPOOLDIR_TAYLOR`; otherwise select the spool directory corresponding to your existing UUCP package. The details of the spool directory choices are described at somewhat tedious length in `unix/spool.c`.

2.2 Compiling Taylor UUCP

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 '`conf.h`' based on '`conf.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`'.

   '`INSTALLDATA`'
   The program to run to install UUCP data files, such as the man pages and the info pages. If this is not set, then if `configure` finds the BSD `install` program, it will set this to '`install -c -m 644`'; otherwise, it will use '`cp`'.

   Suppose you want to set the environment variable '`CC`' to '`rcc`'. If you are using `sh` or `bash`, 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 wierd system, you may have to configure the package by hand. To do this, copy the file '`conf.h.in`' to '`conf.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 means nothing for uucp, but 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 '`-posix`' executes rmail, which in turn executes sendmail, the later is run under 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 '`-posix`', instead I run `gcc` with '`-D_POSIX_SOURCE`', and add '`-lcposix`' to '`LIBS`'.

4. You should verify that `configure` worked correctly by checking '`conf.h`' and the instances of '`Makefile`'.
5. Edit '`policy.h`' for your local system. The comments should explain the various choices. The default values are intended to be reasonable, so you may not have to make any changes.
6. 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 automatically determined at the moment). If you have any other problems there is probably a bug in the `configure` script.
7. 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 5.3 [Patches], page 83), or just ask for help.

2.3 Testing Taylor UUCP

This package is in use at many sites, and has been running at '`airs.com`' for over a year. However, it will doubtless fail in some situations. Do not rely on this code until you have proven to yourself that it will work.

You can use the `uuchk` program to test your configuration files. It will read them and print out a verbose description. This program should not be made setuid, because it will display passwords if it can read them.

If your system supports pseudo-terminals, and you compiled the code to support the new style of configuration files, 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`', at least at the moment; the STREAMS based code was contributed by Marc Boucher).

To run `tstuu`, just type '`tstuu`' with no arguments while logged in to the compilation directory (since it runs '`./uucp`', '`./uux`' and '`./uucico`'). It 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. It 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 does not seem to work under SunOS 4.1.1. This seems to be due to a bug in the implementation of ptys.

The 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`'. You can test `uuxqt` by running the command '`./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 3.5.4 [Debugging Levels], page 34). 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 `tstuu` passes, or you can't run it for some reason or other, move on to testing with some other system. Set up the configuration files (see Chapter 3 [Configuration Files], page 19), or use an existing configuration. 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.

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 under the `debug` command (see Section 3.5.4 [Debugging Levels], page 34). 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 configuration file can override the name 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`') will prevent too many attempts from occurring in rapid succession.

Again, 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, and a sample of the debugging file showing the problem. General questions such as "why doesn't uucico dial out" are impossible to answer without much more information.

2.4 Installing Taylor UUCP

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`'.

Simply installing the executable files is not enough, however. You must also arrange for them to be used correctly.

2.4.1 Running uucico

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 there may be time restrictions which prevent the call at that time (perhaps because telephone rates are high) (see Section 3.6.3.1 [When to Call], page 38). 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 works get transferred between the systems withing a reasonable time period, you should arrange to periodically invoke `uucico`.

These periodic invocations are normally caused 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`'.

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 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.

The file can be easily created using the '`touch`' command. The name of a poll file currently depends on the type of spool directory you are using, as set in '`policy.h`'. If you are using `SPOOLDIR_TAYLOR` (the default), put something like this in your '`crontab`' file:

```
touch /usr/spool/uucp/sys/C./C.A0000
```

In this example *sys* is the system you wish to call, and '`/usr/spool/uucp`' is your UUCP spool directory. If you are using `SPOOLDIR_HDB`, use

```
touch /usr/spool/uucp/sys/C.sysA0000
```

For example, I use the following crontab entries locally:

```
45 * * * * /bin/echo /usr/lib/uucp/uucico -r1 | /bin/su uucpa
```

```
    40 4,10,15 * * * touch /usr/spool/uucp/uunet/C./C.A0000
```

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 check to call '`uunet`'.

2.4.2 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 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 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`', 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`'.

News 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` when sending mail via UUCP. If you simply replace any existing version of `uux` with the Taylor UUCP version, this will probably happen automatically. However, if both versions exist 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.

Receiving mail or news via UUCP

As noted in [Sending mail or news], page 15, mail is sent by requesting a remote execution of `rmail`. To receive mail, then, all that is necessary is for UUCP to invoke `rmail` itself.

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 3.6.7 [Miscellaneous (sys)], page 50).

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.

2.4.3 Trimming UUCP Log Files

You should also periodically trim the log files, as they will otherwise continue to grow without limit. The names of the log files are set in '`policy.h`', and may be overridden in the configuration file (see Section 3.5 [config File], page 30). By default they are are '`/usr/spool/uucp/Log`' and '`/usr/spool/uucp/Stats`'.

You may find the `savelog` program in the '`contrib`' directory may be of use. There is a manual page for it in '`contrib`' as well.

2.5 TCP together with Taylor UUCP

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.

If you are using the new style configuration files, see Chapter 3 [Configuration Files], page 19. Basically, you can just add the line '`port type tcp`' to the entry in the system configuration file. By default UUCP will get the port number by looking up '`uucp`' in '`/etc/services`'; if '`uucp`' is not found, 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.

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

Basically, you must define a port, either using the port file (see Section 3.7 [port File], page 51) 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 3.5.2 [Configuration File Names], page 32).

Of course, you can get a similar effect by using the BSD `uucpd` program.

You can also have `inetd` start up `uucico` with the '`-l`' switch, which will cause it to prompt with '`login:`' and '`Password:`' and check the results against the UUCP (not the system) password file. This may be used in place of `uucpd`.

3 Taylor UUCP Configuration Files

This chapter describes the configuration files accepted by the Taylor UUCP package if compiled with `HAVE_TAYLOR_CONFIG` defined 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`' option; this can be useful when testing a new configuration. When you use the '`-I`' option the programs will revoke any setuid privileges.

3.1 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.

3.2 Examples of Configuration Files

All the configuration commands are explained in the following sections. However, I'll start by giving a few examples of configuration files. For a more complete description of any of the commands used here see the appropriate section of this chapter. There are also sample configuration files in the '`sample`' subdirectory of the distribution.

3.2.1 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 3.5 [config File], page 30.

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 3.6 [sys File], page 35.

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
```

3.2.2 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 3.6 [sys File], page 35.

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 3.5.2 [Configuration File Names], page 32).

`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 '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 baud 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
```

3.2.3 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/IP 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 requesting 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 3.7 [port File], page 51). 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 3.8 [dial File], page 55).

```
# 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
```

3.3 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 consist of the single word '`Never`', which does not match any time, or a single word with a name defined in a previous `timetable` command (see Section 3.5.1 [Miscellaneous (config)], page 30).

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.

3.4 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 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.

`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, although of course a shell script could be used as an interface.

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 path 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. It is also easy to invoke '`/bin/sh`' directly.

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 3.6.3.3 [Logging In], page 40.

3.5 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.

3.5.1 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 3.6.7 [Miscellaneous (sys)], page 50.

`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.

`unknown` *string* ...

The *string* and subsequent arguments are treated as though they appeared in the system file (see Section 3.6 [sys File], page 35). 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.

`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`.

`timetable` *string string*

The `timetable` defines a timetable that may be used in subsequently appearing time strings; Section 3.3 [Time Strings], page 26. 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.

3.5.2 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 3.6 [sys File], page 35. 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 3.7 [port File], page 51. 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 3.8 [dial File], page 55. 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 3.6.3.3 [Logging In], page 40), 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. 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`). 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 3.6.4 [Accepting a Call], page 42). These password files are optional, although one must exist if `uucico` is to present its own login prompts. 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.

3.5.3 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 '`conf.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 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` variable in '`policy.h`'). After the first debugging message has been written, 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.

3.5.4 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 3.6.7 [Miscellaneous (sys)], page 50).

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

3.6 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 3.5.2 [Configuration File Names], page 32.

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

3.6.1 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 system 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 3.2.3 [Gateway Example], page 23) 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.

3.6.2 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 3.6.1 [Defaults and Alternates], page 36). An optional argument may be used to name the alternate. This name will be put 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 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.

3.6.3 Calling Out

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

3.6.3.1 When to Call

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

> Specify when the system may be called. The first argument is a time string; see Section 3.3 [Time Strings], page 26. 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 3.3 [Time Strings], page 26). 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` 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 as described under the `time` command. 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.

3.6.3.2 Placing the Call

`baud` *number*
`speed` *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 baud rate until an unlocked port is found. The ports are defined in the port file. If both `baud` and `port` commands appear, both are used when selecting a port. To allow calls at more than one baud rate, the `alternate` command must be used (see Section 3.6.1 [Defaults and Alternates], page 36). If this command does not appear, there is no default; the baud rate may be specified in the port file, but if it is not then the natural baud rate of the port will be used (whatever that means on the system). Specifying an explicit baud rate of 0 will request the natural baud rate of the port, overriding any default baud rate 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 `baud` 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 baud rate.

`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 3.7 [port File], page 51). 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 `baud` 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 3.4 [Chat Scripts], page 27). The `dialer-sequence` command in the port file may override this address (see Section 3.7 [port File], page 51).

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

3.6.3.3 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. Chat scripts are explained in Section 3.4 [Chat Scripts], page 27.

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 3.5.2 [Configuration File Names], page 32). 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 3.5.2 [Configuration File Names], page 32). There is no default.

3.6.4 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 3.6.1 [Defaults and Alternates], page 36) 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 3.4 [Chat Scripts], page 27) that is run whenever the local system is called by the system being defined. The chat script defined by the `chat` command (see Section 3.6.3.3 [Logging In], page 40), 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. See Section 3.6.3.3 [Logging In], page 40 for additional escape sequence which may be used besides those defined for all chat scripts. There is no default called chat script. If the called chat script fails, the incoming call will be aborted.

3.6.5 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.

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 3.7 [port File], page 51).

The '`g`' protocol is robust, but requires an eight-bit connection.

The '`G`' protocol is the System V Release 4 version of the '`g`' protocol.

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.

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.

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 3.7 [port File], page 51), 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 31 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-window`

If this is between 1 and 31 inclusive, the window 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.

`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.

The ‘`g`’ and ‘`G`’ 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, which is the only packet size supported by many older UUCP packages. Some UUCP packages will even dump core if a larger packet size is requested.

`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`' protocol 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 3.4 [Chat Scripts], page 27). 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 protocol parameters are reset to their default values after each call.

3.6.6 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*

: Equivalent to specifying both '`call-transfer` *boolean*' '`called-transfer` *boolean*'.

`call-local-size` *number string*

: The *string* is a time string (see Section 3.3 [Time Strings], page 26). 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 inform this package of 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.

3.6.7 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` *string*

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 3.5.4 [Debugging Levels], page 34) 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. This 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'. The default is 'abnormal,chat,handshake'; to turn off these default entries, you must use 'max-remote-debug none' followed by other max-remote-debug commands specifying the settings you want.

3.6.8 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
```

3.7 The Port Configuration File

The port files may be used to name and describe ports. Any commands in the file before the first port command specify defaults for all ports in the file. 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.

`protocol` *string*

Specify a list of protocols to use for this port. This is just like the corresponding command for a system (see Section 3.6.5 [Protocol Selection], page 43). 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 3.6.5 [Protocol Selection], page 43). 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`'.

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

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

The argument indicates whether the port supports carrier. If it does not, carrier will never be required on this port, regardless of what the modem chat script indicates. The default is true.

`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 dialer file. There is no default. Some sort of dialer information must be specified to call out on a modem.

`dialer` *string* ... `[ modem only ]`

Execute a dialer command. 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 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, only the `complete` and `abort` chat scripts from the first dialer specified are used, and only the protocol parameters from the first dialer are used.

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.

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 3.6.3.2 [Placing the Call], page 39). Escape sequences in the address are expanded as they are for chat script expect strings (see Section 3.4 [Chat Scripts], page 27). 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 modem.

`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 3.4 [Chat Scripts], page 27).

3.8 The Dialer Configuration File

The dialer configuration files define dialers. 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. See Section 3.4 [Chat Scripts], page 27 for full details on chat scripts.

: Taylor UUCP 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 3.5.2 [Configuration File Names], page 32) 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*

If the argument is true, 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 not to 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 3.4 [Chat Scripts], page 27) 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 3.4 [Chat Scripts], page 27) 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 3.6.5 [Protocol Selection], page 43. 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.

3.9 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.

By default Taylor UUCP provides few mechanisms to secure local users of the system from each other. You can allow 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 3.5.2 [Configuration File Names], page 32) 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 3.7 [port File], page 51) you might invoke '`uucico -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 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.

4 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 UUCP code works.

This chapter is also, unfortunately, somewhat out of date, although I believe that is incomplete rather than inaccurate. I post this information to the newsgroups '`comp.mail.uucp`' and '`news.answers`' each month; if you want to write code based on this information, please get the most recent copy.

Most of the discussion covers the protocols used by all UUCP packages, not just Taylor UUCP. Any information specific to Taylor UUCP is indicated as such. There are some pointers to the actual functions in the Taylor UUCP source code, for those who are extremely interested in actual UUCP implementation.

4.1 UUCP Grades

Modern UUCP packages support grades for each command. The grades generally range from '`A`' (the highest) to '`Z`' followed by '`a`' to '`z`'. Taylor UUCP also supports '`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. 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 uses a different spool directory format, which allows up to fourteen characters to be used for each system name. The Taylor UUCP spool directory format is configurable. The new Taylor spool directory format permits system names to be as long as file names; the maximum length of a file name depends on the particular Unix file system being used.

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 creates command files in the function `zsysdep_spool_commands`. The file name is constructed by the function `zsfile_name`, which knows about all the different types of spool directories supported by Taylor UUCP. The Taylor UUCP sequence number can contain any alphanumeric character; the next sequence number is determined by the function `fscmd_seq`.

I do not know how command grades are handled in 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 only 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 (see Section 3.6.3.1 [When to Call], page 38) to achieve the same effect (and supports the above format when reading '`Systems`' or '`L.sys`').

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 'Z' and news at grade 'd'. The example above would allow mail to be received at any time, but would only permit news to be transferred at night.

4.2 UUCP Lock File Format

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.

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 encoded the process ID as a four byte binary number. The order of the bytes was 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). I have also seen a third type of UUCP lock file which did not contain the process ID at all.

The name of the lock file is generally "LCK.." followed by the base name of the device. For example, to lock /dev/ttyd0 the file LCK..ttyd0 would be created. There are various exceptions. 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 forms the lock file name using the major and minor device numbers rather than the device name (this is pretty sensible if you think about it).

Taylor UUCP can be configured to use various different types of locking. The actual locking code is in the function `fsdo_lock`.

4.3 The Common 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.

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 '`uucico`'.

4.3.1 Initial Handshake

All messages in the initial handshake begin with a '`^P`' (a byte with the octal value \020) and end with a null byte (\000).

Taylor UUCP implements the initial handshake for the calling machine in `fdo_call`, and for the called machine in `faccept_call`.

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: '`\020S`*hostname options*`\000`'

The *hostname* is the UUCP name of the calling machine. The following *options* may appear (or there may be none):

'`-Q`*seq*' 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.

'`-x`*level*' Requests the called system to set its debugging level to the specified value. This is not supported by all systems. Taylor UUCP currently never generates this switch. When it sees it, it restricts the value according to `max-remote-debug` (see Section 3.6.7 [Miscellaneous (sys)], page 50).

'`-p`*grade*'

'`-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=`'. Taylor UUCP supports both.

'`-R`' Indicates that the calling UUCP understands how to restart failed file transmissions. Supported only by System V Release 4 UUCP.

'`-U`*limit*' 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 by System V Release 4 UUCP. Taylor UUCP understands this option, but never generates it.

'`-N`' Indicates that the calling UUCP understands the Taylor UUCP size limiting extensions. Supported only by Taylor UUCP.

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`' The calling UUCP is acceptable, it specified '`-N`', and the called UUCP also understands the Taylor UUCP size limiting extensions. Supported only by Taylor UUCP.

'`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. Note that 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.

If the response is '`ROK`', the following options are supported by System V Release 4 UUCP.

'-R' The called UUCP knows how to restart failed file transmissions.

'-U*limit*' 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.

'-x*level*' I'm told that this is sometimes sent by SVR4 UUCP, but I'm not sure exactly what it 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: '\020P*protocols*\000'

The P is a literal character. 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. For example, the called UUCP might send '\020Pgf\000'.

caller: '\020U*protocol*\000'

The U is a literal character. 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. Taylor UUCP provides the protocol command which may be used either for a system (see Section 3.6.5 [Protocol Selection], page 43) or a port (see Section 3.7 [port File], page 51).

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.

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.

4.3.2 File Requests

The master will send one of four commands: '`S`', '`R`', '`X`' or '`H`'.

Any file name referred to below is either an absolute pathname beginning with '`/`', a public directory pathname beginning with '`~/`', a pathname relative to a user's home directory beginning with '`~`*user*`/`', or a spool directory file name. File names in the spool directory are not pathnames, but instead are converted to pathnames 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).

Taylor UUCP chooses which request to send next in the function `fuucp`. This is also where Taylor UUCP processes incoming commands from the remote system.

4.3.2.1 S Request

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. Taylor UUCP handles the '`S`' request in the file '`send.c`'.

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* is a dummy string, normally ‘`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 (Taylor UUCP does the latter).

notify This field is only used if the ‘`n`’ option appears in *options*. Otherwise, it may not appear, or it may be the string ‘`dummy`’, or it may simply be a pair of double quotes. If the ‘`n`’ option is specified, then when the transfer is successfully completed the slave will send mail to *notify*, which must be a legal mailing address on the slave.

size This field is only present when doing size negotiation, either with Taylor UUCP or SVR4 UUCP. It is the size of the file in bytes. SVR4 UUCP sends the size in base 16 as 0x... while Taylor UUCP sends the size as a decimal integer (a later version of Taylor UUCP will probably change to the SVR4 behaviour).

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 SVR4 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.

If the slave responds with '`SY`', a file transfer begins. When the file transfer is complete, the slave sends a '`C`' command response. Taylor UUCP generates this confirmation in `fprecfile_confirm` and checks it in `fpsendfile_confirm`.

'`CY`' The file transfer was successful.

'`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.

4.3.2.2 R Request

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 implements file transfer restart in this case. Taylor UUCP implements the '`R`' request in the file '`rec.c`'.

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, this was specified in the '`-U`' option during the initial handshake).

The slave then responds with an '`R`' command response.

'`RY` *mode*' — The slave is willing to send the file, and file transfer begins. *mode* is the octal mode of the file on the slave. The master uses this to set the mode of the file on the master's system just as the slave does the *mode* argument in the send command (see Section 4.3.2.1 [S Request], page 67).

'`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 will be cleared up in a later release of Taylor UUCP).

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. Taylor UUCP sends this confirmation in `fprecfile_confirm` and checks it in `fpsendfile_confirm`.

'`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.

4.3.2.3 X Request

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*'. Taylor UUCP handles the 'X' request in the file '`xcmd.c`'.

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. Taylor UUCP ignores the options field.

The slave then responds with an X command response.

'`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.

4.3.2.4 H Request

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, including Taylor UUCP, 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.

4.3.3 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: `\020OOOOOO\000`

called: `\020OOOOOOO\000`

That is, the calling UUCP sends six letter O's and the called UUCP replies with seven letter O's. Some UUCP packages always send six O's.

4.4 The 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 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.

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 (see Section 4.11 [Documentation References], page 80). A detailed examination of the source code would also be profitable.

The Taylor UUCP code to handle the ‘g’ protocol is in the file ‘`protg.c`’. There are a number of functions; the most important ones are `fgstart`, `fgsend_control`, `fgsenddata`, and `fgprocess_data`.

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 must data follows the six byte header. The amount of data is $2\hat{}k+4$. 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 packet 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 complete byte 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; the 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 packets, 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. Taylor UUCP will recognize it but not generate 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). Some UUCP implementations can handle any packet size up to that specified; some can only handled exactly the size specified. Taylor UUCP will always accept any packet size.

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)` (`^` denotes exclusive or, as in C), and *check* is the result of the following routine 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 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. The window size is sent twice, with both the INITA and the INITC packets.

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.

4.5 The 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 the '`f`' protocol originated in BSD versions of UUCP. It was originally intended for transmission over X.25 PAD links.

The Taylor UUCP code for the '`f`' protocol is in '`protf.c`'.

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.

4.6 The UUCP ‘t’ Protocol

The ‘t’ protocol is intended for TCP links. 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.

The Taylor UUCP code for the '`t`' protocol is in '`prott.c`'.

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.

4.7 The 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 TCP.

The '`e`' protocol originated in versions of HDB UUCP.

The Taylor UUCP code for the '`e`' protocol is in '`prote.c`'.

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.

4.8 The UUCP 'x' Protocol

I believe that the '`x`' protocol was intended for use over X.25 virtual circuits. It relies on a write of zero bytes being read as zero bytes without stopping communication. I have heard that it does not work correctly. If someone would care to fill this in more, I would be grateful. Taylor UUCP does not implement the '`x`' protocol.

4.9 The UUCP 'd' Protocol

This is apparently used for DataKit connections, and relies on a write of zero bytes being read as zero bytes, much as the '`x`' protocol does. I don't really know anything else about it. Taylor UUCP does not implement the '`d`' protocol.

4.10 The UUCP 'G' Protocol

The '`G`' protocol is apparently simply the '`g`' protocol, except that it is known to support all possible window and packet sizes. It was introduced by SVR4 UUCP; the SVR4 implementation of the '`g`' protocol is apparently fixed at a packet size of 64 and a window size of 7. Taylor UUCP does not recognize the '`G`' protocol. It does support all window and packet sizes for the '`g`' protocol.

4.11 Documentation References

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.

Some information is originally due to a Usenet article by Chuck Wegrzyn. 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.

5 Hacking Taylor UUCP

This chapter provides the briefest of guides to the Taylor UUCP source code itself.

5.1 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 files '`tcp.c`', '`tli.c`' and '`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 is in use on a 16 bit segmented system with no function prototypes, so I'm certain that all casts to long and pointers are done when necessary.

5.2 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`’.

5.3 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.

6 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, at least for now) 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 document. Doug Evans contributed the zmodem protocol. 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)
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)
Gregory Bond <gnb@bby.com.au>
Marc Boucher <marc@CAM.ORG>
dean@coplex.com (Dean Brooks)
dave@dlb.com (Dave Buck)
gordon@sneaky.lonestar.org (Gordon Burditt)
mib@gnu.ai.mit.edu (Michael I Bushnell)
Brian Campbell <brianc@quantum.on.ca>
Andrew A. Chernov <ache@astral.msk.su>
mafc!frank@bach.helios.de (Frank Conrad)
Ed Carp <erc@apple.com>
verbus@westmark.westmark.com (Verbus M. Counts)
cbmvax!snark.thyrsus.com!cowan (John Cowan)
```

```
Bob Cunningham <bob@soest.hawaii.edu>
hubert@arakis.fdn.org (Hubert Delahaye)
denny@dakota.alisa.com (Bob Denny)
ssd@nevets.oau.org (Steven S. Dick)
gert@greenie.gold.sub.org (Gert Doering)
Hans-Dieter Doll <hd2@Insel.DE>
Andrew Evans <andrew@airs.com>
dje@cygnus.com (Doug Evans)
Marc Evans <marc@synergytics.com>
kksys!kegworks!lfahnoe@cs.umn.edu (Larry Fahnoe)
fenner@jazz.psu.edu (Bill Fenner)
"David J. Fiander" <golem!david@news.lsuc.on.ca>
Thomas Fischer <batman@olorin.dark.sub.org>
erik@eab.retix.com (Erik Forsberg)
Lele Gaifax <piggy@idea.sublink.org>
Peter.Galbavy@micromuse.co.uk
hunter@phoenix.pub.uu.oz.au (James Gardiner [hunter])
Terry Gardner <cphpcom!tjg01>
ol@infopro.spb.su (Oleg Girko)
jimmy@tokyo07.info.com (Jim Gottlieb)
ryan@cs.umb.edu (Daniel R. Guilderson)
greg@gagme.chi.il.us (Gregory Gulik)
Richard H. Gumpertz <rhg@cps.com>
Michael Haberler <mah@parrot.prv.univie.ac.at>
jh@moon.nbn.com (John Harkin)
guy@auspex.auspex.com (Guy Harris)
Petri Helenius <pete@fidata.fi>
Bob Hemedinger <bob@dalek.mwc.com>
Andrew Herbert <andrew@werple.pub.uu.oz.au>
Peter Honeyman <honey@citi.umich.edu>
pmcgw!personal-media.co.jp!ishikawa (Chiaki Ishikawa)
bei@dogface.austin.tx.us (Bob Izenberg)
Rob Janssen <cmgit!rob@relay.nluug.nl>
harvee!esj (Eric S Johansson)
Alan Judge <aj@dec4ie.IEunet.ie>
chris@cj_net.in-berlin.de (Christof Junge)
tron@Veritas.COM (Ronald S. Karr)
Brendan Kehoe <brendan@cs.widener.edu>
kersing@nlmug.nl.mugnet.org (Jac Kersing)
rob@pact.nl (Rob Kurver)
kent@sparky.IMD.Sterling.COM (Kent Landfield)
lebaron@inrs-telecom.uquebec.ca  (Gregory LeBaron)
karl@sugar.NeoSoft.Com (Karl Lehenbauer)
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)
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>
Giles D Malet <shrdlu!gdm@provar.kwnet.on.ca>
mem@mv.MV.COM (Mark E. Mallett)
pepe@dit.upm.es (Jose A. Manas)
martelli@cadlab.sublink.org (Alex Martelli)
Yanek Martinson <yanek@mthvax.cs.miami.edu>
jm@aristote.univ-paris8.fr (Jean Mehat)
les@chinet.chi.il.us (Leslie Mikesell)
mmitchel@digi.lonestar.org (Mitch Mitchell)
rmohr@infoac.rmi.de (Rupert Mohr)
ianm@icsbelf.co.uk (Ian Moran)
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
nolan@helios.unl.edu (Michael Nolan)
david nugent <david@csource.oz.au>
Petri Ojala <ojala@funet.fi>
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>
dplatt@ntg.com (Dave Platt)
eldorado@tharr.UUCP (Mark Powell)
Mark Powell <mark@inet-uk.co.uk>
pozar@kumr.lns.com (Tim Pozar)
putsch@uicc.com (Jeff Putsch)
Jarmo Raiha <jarmo@ksvltd.FI>
Scott Reynolds <scott@clmqt.marquette.Mi.US>
mcr@Sandelman.OCUnix.On.Ca (Michael Richardson)
arnold@cc.gatech.edu (Arnold Robbins)
ross@sun490.fdu.edu (Jeff Ross)
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)
rsalz@bbn.com (Rich Salz)
sojurn!mike@hobbes.cert.sei.cmu.edu (Mike Sangrey)
Nickolay Saukh <nms@ussr.EU.net>
Eric Schnoebelen <eric@cirr.com>
scott@geom.umn.edu
Igor V. Semenyuk <iga@argrd0.argonaut.su>
uunet!gold.sub.org!root (Christian Seyb)
s4mjs!mjs@nirvo.nirvonics.com (M. J. Shannon Jr.)
peter@ficc.ferranti.com (Peter da Silva)
```

```
frumious!pat (Patrick Smith)
roscom!monty@bu.edu (Monty Solomon)
Harlan Stenn <harlan@mumps.pfcs.com>
Ralf Stephan <ralf@ark.abg.sub.org>
chs@antic.apu.fi (Hannu Strang)
ralf@reswi.ruhr.de (Ralf E. Stranzenbach)
Shigeya Suzuki <shigeya@dink.foretune.co.jp>
swiers@plains.NoDak.edu
Oleg Tabarovsky <olg@olghome.pccentre.msk.su>
John Theus <john@theus.rain.com>
Karsten Thygesen <karthy@dannug.dk>
Graham Toal <gtoal@pizzabox.demon.co.uk>
rmtodd@servalan.servalan.com (Richard Todd)
Len Tower <tower-prep@ai.mit.edu>
Mark Towfiq <justice!towfiq@Eingedi.Newton.MA.US>
mju@mudos.ann-arbor.mi.us (Marc Unangst)
Tomi Vainio <tomppa@fidata.fi>
vogel@omega.ssw.de (Andreas Vogel)
jv@mh.nl (Johan Vromans)
steve@nshore.org (Stephen J. Walick)
gerben@rna.indiv.nluug.nl (Gerben Wierda)
frnkmth!twwells.com!bill (T. William Wells)
Peter Wemm <Peter_Wemm@zeus.dialix.oz.au>
mauxci!eci386!woods@apple.com (Greg A. Woods)
Michael Yu.Yaroslavtsev <mike@yaranga.ipmce.su>
jon@console.ais.org (Jon Zeeff)
Matthias Zepf <agnus@amylnd.stgt.sub.org>
Eric Ziegast <uunet!ziegast>
```

Concept Index

Configuration File Index

Table of Contents