UUCP, protocol, FAQ

- Detailed description of the 'y' protocol.
- Mention the name uuxqt uses for lock files.

This article was written by Ian Lance Taylor '*ian@airs.com*' and I may even update it periodically. Please send me mail about suggestions or inaccuracies.

This article describes how the various UUCP protocols work, and discusses some other internal UUCP issues. It does not describe how to configure UUCP, nor how to solve UUCP connection problems, nor how to deal with UUCP mail. I do not know of any FAQ postings on these topics. There are some documents on the net describing UUCP configuration, but I can not keep an up to date list here; try using archie. If you haven't read the 'news.announce.newusers' articles, read them.

This article is in digest format. Some newsreaders will be able to break it apart into separate articles. Please don't ask me how to do this, though.

This article covers the following topics. If questions about one of these topics is posted to 'comp.mail.uucp', please send mail to the poster referring her or him to this FAQ. There is no reason to post a followup, as most of us know the answer already.

UUCP Protocol Sources Alarm in Debugging Output UUCP Grades UUCP Lock Files Execution File Format UUCP Protocol UUCP 'g' Protocol UUCP 'f' Protocol UUCP 't' Protocol UUCP 'e' Protocol UUCP 'G' Protocol UUCP 'i' Protocol UUCP 'j' Protocol UUCP 'x' Protocol UUCP 'y' Protocol UUCP 'd' Protocol UUCP 'h' Protocol UUCP 'v' Protocol Thanks

Contents

1 UUCP Protocol Sources

UUCP Protocol Sources

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

I took a lot of the information from Jamie E. Hanrahan’s paper in the Fall 1990 DECUS Symposium, and from ‘Managing UUCP and Usenet’ by Tim O’Reilly and Grace Todino (with contributions by several other people). The latter includes most of the former, and is published by O’Reilly & Associates, Inc. 103 Morris Street, Suite A Sebastopol, CA 95472 It is currently in its tenth edition. The ISBN number is ‘0-937175-93-5’.

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

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

2 Alarm in Debugging Output

Alarm in Debugging Output

The debugging output of many versions of UUCP will include messages like 'alarm 1' or 'pkcget: alarm 1'. Taylor UUCP does not use the word 'alarm', but will instead log messages like 'Timed out waiting for packet'.

These types of messages mean that the UUCP package has timed out while waiting for some sort of response from the remote system. If it happens consistently when trying to transfer a particular file, then the most likely problem is that one of the modems will not transmit the XON or XOFF characters. Several UUCP protocols require an eight bit clean connection, which means that the modems must treat XON or XOFF as normal data characters, not as flow control signals. This should always be checked first.

Other possible problems are that the modems have simply dropped their connection, or perhaps on one side or the other the serial buffer is overflowing and dropping characters. Another possibility is that the UUCP packages disagree about some aspect of the UUCP protocol, which is uncommon but does happen occasionally.

Using the information in the following sections, you should be able to figure out what type of data your UUCP was expecting to receive. This may give some indication as to exactly what the problem is. It is difficult to be more specific, since there are many possiblities.

3 UUCP Grades

UUCP Grades

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

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

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

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

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

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

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

Modern UUCP packages allow you to restrict file transfer by grade depending on the time of day. Typically this is done with a line in the 'Systems' (or 'L.sys') file like this: airs Any/Z,Any2305-0855 ... This allows grades 'Z' and above to be transferred at any time. Lower grades may only be transferred at night. I believe that this grade restriction applies to local commands as well as to remote commands, but I am not sure. It may only apply if the UUCP package places the call, not if it is called by the remote system.

Taylor UUCP can use the 'timegrade' and 'call-timegrade' commands to achieve the same effect. It supports the above format when reading 'Systems' or 'L.sys'.

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

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

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

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

4 UUCP Lock Files

UUCP Lock Files

This discussion applies only to Unix. I have no idea how UUCP locks ports on other systems.

UUCP creates files to lock serial ports and systems. On most, if not all, systems, these same lock files are also used by 'cu' to coordinate access to serial ports. On some systems 'getty' also uses these lock files, often under the name 'uugetty'.

The lock file normally contains the process ID of the locking process. This makes it easy to determine whether a lock is still valid. The algorithm is to create a temporary file and then link it to the name that must be locked. If the link fails because a file with that name already exists, the existing file is read to get the process ID. If the process still exists, the lock attempt fails. Otherwise the lock file is deleted and the locking algorithm is retried.

Older UUCP packages put the lock files in the main UUCP spool directory, '/usr/spool/uucp'. HDB UUCP generally puts the lock files in a directory of their own, usually '/usr/spool/locks' or '/etc/locks'.

The original UUCP lock file format encodes the process ID as a four byte binary number. The order of the bytes is host-dependent. HDB UUCP stores the process ID as a ten byte ASCII decimal number, with a trailing newline. For example, if process 1570 holds a lock file, it would contain the eleven characters space, space, space, space, space, space, one, five, seven, zero, newline. Some versions of UUCP add a second line indicating which program created the lock ('uucp', 'cu', or 'getty/uugetty'). I have also seen a third type of UUCP lock file which does not contain the process ID at all.

The name of the lock file is traditionally 'LCK..' followed by the base name of the device. For example, to lock '/dev/ttyd0' the file 'LCK..ttyd0' would be created. On SCO Unix, the lock file name is always forced to lower case even if the device name has upper case letters.

System V Release 4 UUCP names the lock file using the major and minor device numbers rather than the device name. The file is named 'LK.XXX.YYY.ZZZ', where XXX, YYY and ZZZ are all three digit decimal numbers. XXX is the major device number of the device holding the directory holding the device file (e.g., '/dev'). YYY is the major device number of the device file itself. ZZZ is the minor device number of the device file itself. If 's' holds the result of passing the device to the stat system call (e.g., 'stat ("/dev/ttyd0",)'), the following line of C code will print out the corresponding lock file name: printf ("LK.%03d.%03d.%03d", major (s.st_dev), major (s.st_rdev), minor (s.st_rdev)); The advantage of this system is that even if there are several links to the same device, they will all use the same lock file name.

When two or more instances of 'uuxqt' are executing, some sort of locking is needed to ensure that a single execution job is only started once. I don't know how most UUCP packages deal with this. Taylor UUCP uses a lock file for each execution job. The name of the lock file is the same as the name of the 'X.*' file, except that the initial 'X' is changed to an 'L'. The lock file holds the process ID as described above.

5 Execution File Format

Execution File Format

UUCP 'X.*' files control program execution. They are created by 'uux'. They are transferred between systems just like any other file. The 'uuxqt' daemon reads them to figure out how to execute the job requested by 'uux'.

An 'X.*' file is simply a text file. The first character of each line is a command, and the remainder of the line supplies arguments. The following commands are defined:

'C command' This gives the command to execute, including the program and all arguments. For example, 'rmail ian@airs.com'.

'U user system' This names the user who requested the command, and the system from which the request came.

'I standard-input' This names the file from which standard input is taken. If no standard input file is given, the standard input will probably be attached to '/dev/null'. If the standard input file is not from the system on which the execution is to occur, it will also appear in an 'F' command.

'O standard-output [system]' This names the standard output file. The optional second argument names the system to which the file should be sent. If there is no second argument, the file should be created on the executing system.

'F required-file [filename-to-use]' The 'F' command can appear multiple times. Each 'F' command names a file which must exist before the execution can proceed. This will usually be a file which is transferred from the system on which 'uux' was executed, but it can also be a file from the local system or some other system. If the file is not from the local system, then the command will usually name a file in the spool directory. If the optional second argument appears, then the file should be copied to the execution directory under that name. This is necessary for any file other than the standard input file. If the standard input file is not from the local system, it will appear in both an 'F' command and an 'I' command.

'R requestor-address' This is the address to which mail about the job should be sent. It is relative to the system named in the 'U' command. If the 'R' command does not appear, then mail is sent to the user named in the 'U' command.

'Z' This command takes no arguments. It means that a mail message should be sent if the command failed. This is the default behaviour for most modern UUCP packages, and for them the 'Z' command does not actually do anything.

'N' This command takes no arguments. It means that no mail message should be sent, even if the command failed.

'n' This command takes no arguments. It means that a mail message should be sent if the command succeeded. Normally a message is sent only if the command failed.

'B' This command takes no arguments. It means that the standard input should be returned with any error message. This can be useful in cases where the input would otherwise be lost.

'e' This command takes no arguments. It means that the command should be processed with '/bin/sh'. For some packages this is the default anyhow. Most packages will refuse to execute complex commands or commands containing wildcards, because of the security holes this opens.

'E' This command takes no arguments. It means that the command should be processed with the 'execve' system call. For some packages this is the default anyhow.

'M status-file' This command means that instead of mailing a message, the message should be copied to the named file on the system named by the 'U' command.

'# comment' This command is ignored, as is any other unrecognized command.

Here is an example. Given the following command executed on system test1 uux - test2!cat - test2! ian/bar !qux '> /gorp' (this is only an example, as most UUCP systems will not permit the cat command to be executed) Taylor UUCP will produce something like the following 'X.' file: U ian test1 F D.test1N003r qux O /usr/spool/uucppublic test1 F D.test1N003s I D.test1N003s C cat - ian/bar qux The standard input will be read into a file and then transferred to the file 'D.test1N003s' on system 'test2'. The file 'qux' will be transferred to 'D.test1N003r' on system 'test2'. When the command is executed, the latter file will be copied to the execution directory under the name 'qux'. Note that since the file ' ian/bar' is already on the execution system, no action need be taken for it. The standard output will be collected in a file, then copied to the directory '/usr/spool/uucppublic' on the system 'test1'.

6 UUCP Protocol

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.

The Initial Handshake

Before the initial handshake, the caller will usually have logged in the called machine and somehow started the UUCP package there. On Unix this is normally done by setting the shell of the login name used to '/usr/lib/uucp/uucico'.

All messages in the initial handshake begin with a '^P' (a byte with the octal value '\020') and end with a null byte ('\000'). A few systems end these messages with a line feed character ('\012') instead of a null byte; the examples below assume a null byte is being used.

Some options below are supported by QFT, which stands for Queued File Transfer, and is (or was) an internal Bell Labs version of UUCP.

Taylor UUCP size negotiation was introduced by Taylor UUCP, and is also supported by DOS based UUPlus and Amiga based wUUCP and UUCP-1.17.

The initial handshake goes as follows. It is begun by the called machine.

called: '\020Shere=hostname\000' The hostname is the UUCP name of the called machine. Older UUCP packages do not output it, and simply send '\020Shere\000'.

caller: '\020Shostname options\000' The hostname is the UUCP name of the calling machine. The following options may appear (or there may be none):

'-QSEQ' Report sequence number for this conversation. The sequence number is stored at both sites, and incremented after each call. If there is a sequence number mismatch, something has gone wrong (somebody may have broken security by pretending to be one of the machines) and the call is denied. If the sequence number changes on one of the machines, perhaps because of an attempted breakin or because a disk backup was restored, the sequence numbers on the two machines must be reconciled manually.

'-xLEVEL' Requests the called system to set its debugging level to the specified value. This is not supported by all systems.

'-pGRADE' '-vgrade=GRADE' Requests the called system to only transfer files of the specified grade or higher. This is not supported by all systems. Some systems support '-p', some support '-vgrade='. UUPlus allows either '-p' or '-v' to be specified on a per-system basis in the 'SYSTEMS' file ('gradechar' option).

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

'-ULIMIT' Reports the ulimit value of the calling UUCP. The limit is specified as a base 16 number in C notation (e.g., '-U0x1000000'). This number is the number of 512 byte blocks in the largest file which the calling UUCP can create. The called UUCP may not transfer a file larger than this. Supported only by System V Release 4 UUCP, QFT and UUPlus. UUPlus reports the lesser of the available disk space on the spool directory drive and the ulimit variable in 'UUPLUS.CFG'. Taylor UUCP understands this option, but does not generate it.

'-N[NUMBER]' Indicates that the calling UUCP understands the Taylor UUCP size negotiation extension. Not supported by traditional UUCP packages. Supported by UUPlus. The optional number is a bitmask of features supported by the calling UUCP, and is described below.

called: '\020ROK\000' There are actually several possible responses. 'ROK' The calling UUCP is acceptable, and the handshake proceeds to the protocol negotiation. Some options may also appear; see below.

'ROKN[NUMBER]' The calling UUCP is acceptable, it specified '-N', and the called UUCP also understands the Taylor UUCP size limiting extensions. The optional number is a bitmask of features supported by the called UUCP, and is described below.

'RLCK' The called UUCP already has a lock for the calling UUCP, which normally indicates the two machines are already communicating.

'RCB' The called UUCP will call back. This may be used to avoid impostors (but only one machine out of each pair should call back, or no conversation will ever begin).

'RBADSEQ' The call sequence number is wrong (see the '-Q' discussion above).

'RLOGIN' The calling UUCP is using the wrong login name.

'RYou are unknown to me' The calling UUCP is not known to the called UUCP, and the called UUCP does not permit connections from unknown systems. Some versions of UUCP just drop the line rather than sending this message.

If the response is 'ROK', the following options are supported by System V Release 4 UUCP and QFT. '-R' The called UUCP knows how to restart failed file transmissions.

'-ULIMIT' Reports the ulimit value of the called UUCP. The limit is specified as a base 16 number in C notation. This number is the number of 512 byte blocks in the largest file which the called UUCP can create. The calling UUCP may not send a file larger than this. Also supported by UUPlus. Taylor UUCP understands this option, but does not generate it.

'-xLEVEL' I'm not sure just what this means. It may request the calling UUCP to set its debugging level to the specified value.

If the response is not 'ROK' (or 'ROKN') both sides hang up the phone, abandoning the call.

called: '\020Pprotocols\000' Note that the called UUCP outputs two strings in a row. The protocols string is a list of UUCP protocols supported by the caller. Each UUCP protocol has a single character name. These protocols are discussed in more detail later in this document. For example, the called UUCP might send '\020Pgf\000'.

caller: '\020Uprotocol\000' The calling UUCP selects which protocol to use out of the protocols offered by the called UUCP. If there are no mutually supported protocols, the calling UUCP sends '\020UN\000' and both sides hang up the phone. Otherwise the calling UUCP sends something like '\020Ug\000'.

Most UUCP packages will consider each locally supported protocol in turn and select the first one supported by the called UUCP. With some versions of HDB UUCP, this can be modified by giving a list of protocols after the device name in the 'Devices' file or the 'Systems' file. For example, to select the 'e' protocol in 'Systems', airs Any ACU,e ... or in Devices, ACU,e ttyXX ... Taylor UUCP provides the 'protocol' command which may be used either for a system or a port. UUPlus allows specification of the protocol string on a per-system basis in the 'SYSTEMS' file.

The optional number following a '-N' sent by the calling system, or an 'ROKN' sent by the called system, is a bitmask of features supported by the UUCP package. The optional number was introduced in Taylor UUCP version 1.04. The number is sent as an octal number with a leading zero. The following bits are currently defined. A missing number should be taken as '011'.

'01' UUCP supports size negotiation.

'02' UUCP supports file restart.

'04' UUCP supports the 'E' command.

'010' UUCP requires the file size in the 'S' and 'R' commands to be in base 10. This bit is used by default if no number appears, but should not be explicitly sent.

'020' UUCP expects a dummy string between the notify field and the size field in an 'S' command. This is true of SVR4 UUCP. This bit should not be used.

After the protocol has been selected and the initial handshake has been completed, both sides turn on the selected protocol. For some protocols (notably 'g') a further handshake is done at this point.

UUCP Protocol Commands

Each protocol supports a method for sending a command to the remote system. This method is used to transmit a series of commands between the two UUCP packages. At all times, one package is the master and the other is the slave. Initially, the calling UUCP is the master.

If a protocol error occurs during the exchange of commands, both sides move immediately to the final handshake.

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

Any file name referred to below is either an absolute file name beginning with '/', a public directory file name beginning with ' /', a file name relative to a user's home directory beginning with ' USER/', or a spool directory file name. File names in the spool directory are not absolute, but instead are converted to file names within the spool directory by UUCP. They always begin with 'C.' (for a command file created by 'uucp' or 'uux'), 'D.' (for a data file created by 'uucp', 'uux' or by an execution, or received from another system for an execution), or 'X.' (for an execution file created by 'uux' or received from another system).

The S Command ....

master: 'S FROM TO USER -OPTIONS TEMP MODE NOTIFY SIZE' The 'S' and the '-' are literal characters. This is a request by the master to send a file to the slave.

FROM The name of the file to send. If the 'C' option does not appear in OPTIONS, the master will actually open and send this file. Otherwise the file has been copied to the spool directory, where it is named TEMP. The slave ignores this field unless TO is a directory, in which case the basename of FROM will be used as the file name. If FROM is a spool directory filename, it must be a data file created for or by an execution, and must begin with 'D.'.

TO The name to give the file on the slave. If this field names a directory the file is placed within that directory with the basename of FROM. A name ending in '/' is taken to be a directory even if one does not already exist with that name. If TO begins with 'X.', an execution file will be created on the slave. Otherwise, if TO begins with 'D.' it names a data file to be used by some execution file. Otherwise, TO should not be in the spool directory.

USER The name of the user who requested the transfer.

OPTIONS A list of options to control the transfer. The following options are defined (all options are single characters): 'C' The file has been copied to the spool directory (the master should use TEMP rather than FROM).

'c' The file has not been copied to the spool directory (this is the default).

'd' The slave should create directories as necessary (this is the default).

'f' The slave should not create directories if necessary, but should fail the transfer instead.

'm' The master should send mail to USER when the transfer is complete.

'n' The slave should send mail to NOTIFY when the transfer is complete.

TEMP If the 'C' option appears in OPTIONS, this names the file to be sent. Otherwise if FROM is in the spool directory, TEMP is the same as FROM. Otherwise TEMP may be a dummy string, such as 'D.0'. After the transfer has been succesfully completed, the master will delete the file TEMP.

MODE This is an octal number giving the mode of the file on the master. If the file is not in the spool directory, the slave will always create it with mode 0666, except that if (MODE & 0111) is not zero (the file is executable), the slave will create the file with mode 0777. If the file is in the spool directory, some UUCP packages will use the algorithm above and some will always create the file with mode 0600. This field is ignored by UUPlus, since it is meaningless on DOS; UUPlus uses 0666 for outgoing files.

NOTIFY This field may not be present, and in any case is only meaningful if the 'n' option appears in OPTIONS. If the 'n' option appears, then, when the transfer is successfully completed, the slave will send mail to NOTIFY, which must be a legal mailing address on the slave. If a SIZE field will appear but the 'n' option does not appear, NOTIFY will always be present, typically as the string 'dummy' or simply a pair of double quotes.

SIZE This field is only present when doing Taylor UUCP or SVR4 UUCP size negotiation. It is the size of the file in bytes. Taylor UUCP version 1.03 sends the size as a decimal integer, while versions 1.04 and up, and all other UUCP packages that support size negotiation, send the size in base 16 with a leading 0x.

The slave then responds with an 'S' command response.

'SY START' The slave is willing to accept the file, and file transfer begins. The START field will only be present when using file restart. It specifies the byte offset into the file at which to start sending. If this is a new file, START will be 0x0.

'SN2' The slave denies permission to transfer the file. This can mean that the destination directory may not be accessed, or that no requests are permitted. It implies that the file transfer will never succeed.

'SN4' The slave is unable to create the necessary temporary file. This implies that the file transfer might succeed later.

'SN6' This is only used by Taylor UUCP size negotiation. It means that the slave considers the file too large to transfer at the moment, but it may be possible to transfer it at some other time.

'SN7' This is only used by Taylor UUCP size negotiation. It means that the slave considers the file too large to ever transfer.

'SN8' This is only used by Taylor UUCP. It means that the file was already received in a previous conversation. This can happen if the receive acknowledgement was lost after it was sent by the receiver but before it was received by the sender.

'SN9' This is only used by Taylor UUCP (versions 1.05 and up) and UUPlus (versions 2.0 and up). It means that the remote system was unable to open another channel (see the discussion of the 'i' protocol for more information about channels). This implies that the file transfer might succeed later.

'SN10' This is reportedly used by SVR4 UUCP to mean that the file size is too large.

If the slave responds with 'SY', a file transfer begins. When the file transfer is complete, the slave sends a

'C' command response.

'CY' The file transfer was successful.

'CYM' The file transfer was successful, and the slave wishes to become the master; the master should send an 'H' command, described below.

'CN5' The temporary file could not be moved into the final location. This implies that the file transfer will never succeed.

After the 'C' command response has been received (in the 'SY' case) or immediately (in an 'SN' case) the master will send another command.

The R Command ....

master: 'R FROM TO USER -OPTIONS SIZE' The 'R' and the '-' are literal characters. This is a request by the master to receive a file from the slave. I do not know how SVR4 UUCP or QFT implement file transfer restart in this case.

FROM This is the name of the file on the slave which the master wishes to receive. It must not be in the spool directory, and it may not contain any wildcards.

TO This is the name of the file to create on the master. I do not believe that it can be a directory. It may only be in the spool directory if this file is being requested to support an execution either on the master or on some system other than the slave.

USER The name of the user who requested the transfer.

OPTIONS A list of options to control the transfer. The following options are defined (all options are single characters): 'd' The master should create directories as necessary (this is the default).

'f' The master should not create directories if necessary, but should fail the transfer instead.

'm' The master should send mail to USER when the transfer is complete.

SIZE This only appears if Taylor UUCP size negotiation is being used. It specifies the largest file which the master is prepared to accept (when using SVR4 UUCP or QFT, this was specified in the '-U' option during the initial handshake).

The slave then responds with an 'R' command response. UUPlus does not support 'R' requests, and always responds with 'RN2'.

'RY MODE [SIZE]' The slave is willing to send the file, and file transfer begins. The MODE argument is the octal mode of the file on the slave. The master treats this just as the slave does the MODE argument in the send command, q.v. I am told that SVR4 UUCP sends a trailing SIZE argument. For some versions of BSD UUCP, the MODE argument may have a trailing 'M' character (e.g., 'RY 0666M'). This means that the slave wishes to become the master.

'RN2' The slave is not willing to send the file, either because it is not permitted or because the file does not exist. This implies that the file request will never succeed.

'RN6' This is only used by Taylor UUCP size negotiation. It means that the file is too large to send, either because of the size limit specifies by the master or because the slave considers it too large. The file transfer might succeed later, or it might not (this may be cleared up in a later release of Taylor UUCP).

'RN9' This is only used by Taylor UUCP (versions 1.05 and up) and FSUUCP (versions 1.5 and up). It means that the remote system was unable to open another channel (see the discussion of the 'i' protocol for more information about channels). This implies that the file transfer might succeed later.

If the slave responds with 'RY', a file transfer begins. When the file transfer is complete, the master sends a 'C' command. The slave pretty much ignores this, although it may log it.

'CY' The file transfer was successful.

'CN5' The temporary file could not be moved into the final location.

After the 'C' command response has been sent (in the 'RY' case) or immediately (in an 'RN' case) the master will send another command.

The X Command ....

master: 'X FROM TO USER -OPTIONS' The 'X' and the '-' are literal characters. This is a request by the master to, in essence, execute uucp on the slave. The slave should execute 'uucp FROM TO'.

FROM This is the name of the file or files on the slave which the master wishes to transfer. Any wildcards are expanded on the slave. If the master is requesting that the files be transferred to itself, the request would normally contain wildcard characters, since otherwise an 'R' command would suffice. The master can also use this command to request that the slave transfer files to a third system.

TO This is the name of the file or directory to which the files should be transferred. This will normally use a UUCP name. For example, if the master wishes to receive the files itself, it would use 'master!path'.

USER The name of the user who requested the transfer.

OPTIONS A list of options to control the transfer. It is not clear which, if any, options are supported by most UUCP packages.

The slave then responds with an 'X' command response. FSUUCP does not support 'X' requests, and always responds with 'XN'.

'XY' The request was accepted, and the appropriate file transfer commands have been queued up for later processing.

'XN' The request was denied. No particular reason is given.

In either case, the master will then send another command.

The E Command ....

master: 'E FROM TO USER -OPTIONS TEMP MODE NOTIFY SIZE COMMAND' The 'E' command is only supported by Taylor UUCP 1.04 and up. It is used to make an execution request without requiring a separate 'X.*' file. It is only used when the command to be executed requires a single input file which is passed to it as standard input.

All the fields have the same meaning as they do for an 'S' command, except for OPTIONS and COMMAND.

OPTIONS A list of options to control the transfer. The following options are defined (all options are single characters): 'C' The file has been copied to the spool directory (the master should use TEMP rather than FROM).

'c' The file has not been copied to the spool directory (this is the default).

'N' No mail message should be sent, even if the command fails. This is the equivalent of the 'N' command in an 'X.*' file.

'Z' A mail message should be sent if the command fails (this is generally the default in any case). This is the equivalent of the 'Z' command in an 'X.*' file.

'R' Mail messages about the execution should be sent to the address in the NOTIFY field. This is the equivalent of the 'R' command in an 'X.*' file.

'e' The execution should be done with '/bin/sh'. This is the equivalent of the 'e' command in an 'X.*' file.

COMMAND The command which should be executed. This is the equivalent of the 'C' command in an 'X.*' file.

The slave then responds with an 'E' command response. These are the same as the 'S' command responses, but the initial character is 'E' rather than 'S'.

If the slave responds with 'EY', the file transfer begins. When the file transfer is complete, the slave sends a 'C' command response, just as for the 'S' command. After a successful file transfer, the slave is responsible for arranging for the command to be executed. The transferred file is passed as standard input, as though it were named in the 'I' and 'F' commands of an 'X.*' file.

After the 'C' command response has been received (in the 'EY' case) or immediately (in an 'EN' case) the master will send another command.

The H Command ....

master: 'H' This is used by the master to hang up the connection. The slave will respond with an 'H' command response.

'HY' The slave agrees to hang up the connection. In this case the master sends another 'HY' command. In some UUCP packages the slave will then send a third 'HY' command. At this point the protocol is shut down, and the final handshake is begun.

'HN' The slave does not agree to hang up. In this case the master and the slave exchange roles. The next command will be sent by the former slave, which is the new master. The roles may be reversed several times during a single connection.

The Final Handshake

After the protocol has been shut down, the final handshake is performed. This handshake has no real purpose, and some UUCP packages simply drop the connection rather than do it (in fact, some will drop the connection immediately after both sides agree to hangup, without even closing down the protocol).

caller: '\020OOOOOO\000' called: '\020OOOOOOO\000' That is, the calling UUCP sends six 'O' characters and the called UUCP replies with seven 'O' characters. Some UUCP packages always send six 'O' characters.

7 UUCP 'g' Protocol

UUCP 'g' Protocol

The 'g' protocol is a packet based flow controlled error correcting protocol that requires an eight bit clear connection. It is the original UUCP protocol, and is supported by all UUCP implementations. Many implementations of it are only able to support small window and packet sizes, specifically a window size of 3 and a packet size of 64 bytes, but the protocol itself can support up to a window size of 7 and a packet size of 4096 bytes. Complaints about the inefficiency of the 'g' protocol generally refer to specific implementations, rather than to the correctly implemented protocol.

The 'g' protocol was originally designed for general packet drivers, and thus contains some features that are not used by UUCP, including an alternate data channel and the ability to renegotiate packet and window sizes during the communication session.

The ‘g’ protocol is spoofed by many Telebit modems. When spoofing is in effect, each Telebit modem uses the ‘g’ protocol to communicate with the attached computer, but the data between the modems is sent using a Telebit proprietary error correcting protocol. This allows for very high throughput over the Telebit connection, which, because it is half-duplex, would not normally be able to handle the ‘g’ protocol very well at all. When a Telebit is spoofing the ‘g’ protocol, it forces the packet size to be 64 bytes and the window size to be 3.

This discussion of the ‘g’ protocol explains how it works, but does not discuss useful error handling techniques. Some discussion of this can be found in Jamie E. Hanrahan’s paper, cited above.

All ‘g’ protocol communication is done with packets. Each packet begins with a six byte header. Control packets consist only of the header. Data packets contain additional data.

The header is as follows:

‘\020’ Every packet begins with a ‘^P’.

K (1 <= K <= 9) The K value is always 9 for a control packet. For a data packet, the K value indicates how much data follows the six byte header. The amount of data is 2 ** (K + 4), where ** indicates exponentiation. Thus a K value of 1 means 32 data bytes and a K value of 8 means 4096 data bytes. The K value for a data packet must be between 1 and 8 inclusive.

checksum low byte checksum high byte The checksum value is described below.

control byte The control byte indicates the type of packet, and is described below.

xor byte This byte is the xor of K, the checksum low byte, the checksum high byte and the control byte (i.e., the second, third, fourth and fifth header bytes). It is used to ensure that the header data is valid.

The control byte in the header is composed of three bit fields, referred to here as TT (two bits), XXX (three bits) and YYY (three bits). The control is TTXXXYYY, or ‘(TT << 6) + (XXX << 3) + YYY’.

The TT field takes on the following values:

‘0’ This is a control packet. In this case the K byte in the header must be 9. The XXX field indicates the type of control packet; these types are described below.

‘1’ This is an alternate data channel packet. This is not used by UUCP.

‘2’ This is a data packet, and the entire contents of the attached data field (whose length is given by the K byte in the header) are valid. The XXX and YYY fields are described below.

‘3’ This is a short data packet. Let the length of the data field (as given by the K byte in the header) be L. Let the first byte in the data field be B1. If B1 is less than 128 (if the most significant bit of B1 is 0), then there are ‘L - B1’ valid bytes of data in the data field, beginning with the second byte. If ‘B1 >= 128’, let B2 be the second byte in the data field. Then there are ‘L - ((B1 & 0x7f) + (B2 << 7))’ valid bytes of data in the data field, beginning with the third byte. In all cases L bytes of data are sent (and all data bytes participate in the checksum calculation) but some of the trailing bytes may be dropped by the receiver. The XXX and YYY fields are described below.

In a data packet (short or not) the XXX field gives the sequence number of the packet. Thus sequence numbers can range from 0 to 7, inclusive. The YYY field gives the sequence number of the last correctly received packet.

Each communication direction uses a window which indicates how many unacknowledged packets may be transmitted before waiting for an acknowledgement. The window may range from 1 to 7, and may be different in each direction. For example, if the window is 3 and the last packet acknowledged was packet number 6, packet numbers 7, 0 and 1 may be sent but the sender must wait for an acknowledgement before sending packet number 2. This acknowledgement could come as the YYY field of a data packet, or as the YYY field of a ‘RJ’ or ‘RR’ control packet (described below).

Each packet must be transmitted in order (the sender may not skip sequence numbers). Each packet must be acknowledged, and each packet must be acknowledged in order.

In a control packet, the XXX field takes on the following values:

1 ‘CLOSE’ The connection should be closed immediately. This is typically sent when one side has seen too many errors and wants to give up. It is also sent when shutting down the protocol. If an unexpected ‘CLOSE’ packet is received, a ‘CLOSE’ packet should be sent in reply and the ‘g’ protocol should halt, causing UUCP to enter the final handshake.

2 ‘RJ’ or ‘NAK’ The last packet was not received correctly. The YYY field contains the sequence number of the last correctly received packet.

3 ‘SRJ’ Selective reject. The YYY field contains the sequence number of a packet that was not received correctly, and should be retransmitted. This is not used by UUCP, and most implementations will not recognize it.

4 ‘RR’ or ‘ACK’ Packet acknowledgement. The YYY field contains the sequence number of the last correctly received packet.

5 ‘INITC’ Third initialization packet. The YYY field contains the maximum window size to use.

6 ‘INITB’ Second initialization packet. The YYY field contains the packet size to use. It requests a size of 2 ** (YYY + 5). Note that this is not the same coding used for the K byte in the packet header (it is 1 less). Most UUCP implementations that request a packet size larger than 64 bytes can handle any packet size up to that specified.

7 ‘INITA’ First initialization packet. The YYY field contains the maximum window size to use.

To compute the checksum, call the control byte (the fifth byte in the header) C.

The checksum of a control packet is simply ‘0xaaaa - C’.

The checksum of a data packet is ‘0xaaaa - (CHECK ^ C)’, where ‘^’ denotes exclusive or, and CHECK is the result of the following routine as run on the contents of the data field (every byte in the data field participates in the checksum, even for a short data packet). Below is the routine used by an early version of Taylor UUCP; it is a slightly modified version of a routine which John Gilmore patched from G.L. Chesson’s original paper. The ‘z’ argument points to the data and the ‘c’ argument indicates how much data there is.

int igchecksum (z, c) register const char *z; register int c; { register unsigned int ichk1, ichk2;

ichk1 = 0xffff; ichk2 = 0;

do { register unsigned int b;

/* Rotate ichk1 left. */ if ((ichk1 & 0x8000) == 0) ichk1 <<= 1; else { ichk1 <<= 1; ++ichk1; }

/* Add the next character to ichk1. */ b = *z++ & 0xff; ichk1 += b;

/* Add ichk1 xor the character position in the buffer counting from the back to ichk2. */ ichk2 += ichk1 ^ c;

/* If the character was zero, or adding it to ichk1 caused an overflow, xor ichk2 to ichk1. */ if (b == 0 || (ichk1 & 0xffff) < b) ichk1 ˆ= ichk2; } while (–c > 0);

return ichk1 & 0xffff; }

When the ‘g’ protocol is started, the calling UUCP sends an ‘INITA’ control packet with the window size it wishes the called UUCP to use. The called UUCP responds with an ‘INITA’ packet with the window size it wishes the calling UUCP to use. Pairs of ‘INITB’ and ‘INITC’ packets are then similarly exchanged. When these exchanges are completed, the protocol is considered to have been started.

Note that the window and packet sizes are not a negotiation. Each system announces the window and packet size which the other system should use. It is possible that different window and packet sizes will be used in each direction. The protocol works this way on the theory that each system knows how much data it can accept without getting overrun. Therefore, each system tells the other how much data to send before waiting for an acknowledgement.

When a UUCP package transmits a command, it sends one or more data packets. All the data packets will normally be complete, although some UUCP packages may send the last one as a short packet. The command string is sent with a trailing null byte, to let the receiving package know when the command is finished. Some UUCP packages require the last byte of the last packet sent to be null, even if the command ends earlier in the packet. Some packages may require all the trailing bytes in the last packet to be null, but I have not confirmed this.

When a UUCP package sends a file, it will send a sequence of data packets. The end of the file is signalled by a short data packet containing zero valid bytes (it will normally be preceeded by a short data packet containing the last few bytes in the file).

Note that the sequence numbers cover the entire communication session, including both command and file data.

When the protocol is shut down, each UUCP package sends a ‘CLOSE’ control packet.

8 UUCP ‘f’ Protocol

UUCP ‘f’ Protocol

The ‘f’ protocol is a seven bit protocol which checksums an entire file at a time. It only uses the characters between ‘\040’ and ‘\176’ (ASCII ‘space’ and ‘ ’) inclusive, as well as the carriage return character. It can be very efficient for transferring text only data, but it is very inefficient at transferring eight bit data (such as compressed news). It is not flow controlled, and the checksum is fairly insecure over large files, so using it over a serial connection requires handshaking (XON/XOFF can be used) and error correcting modems. Some people think it should not be used even under those circumstances.

I believe that the ‘f’ protocol originated in BSD versions of UUCP. It was originally intended for transmission over X.25 PAD links.

The ‘f’ protocol has no startup or finish protocol. However, both sides typically sleep for a couple of seconds before starting up, because they switch the terminal into XON/XOFF mode and want to allow the changes to settle before beginning transmission.

When a UUCP package transmits a command, it simply sends a string terminated by a carriage return.

When a UUCP package transmits a file, each byte B of the file is translated according to the following table:

0 <= B <= 037: 0172, B + 0100 (0100 to 0137) 040 <= B <= 0171: B (040 to 0171) 0172 <= B <= 0177: 0173, B - 0100 (072 to 077) 0200 <= B <= 0237: 0174, B - 0100 (0100 to 0137) 0240 <= B <= 0371: 0175, B - 0200 (040 to 0171) 0372 <= B <= 0377: 0176, B - 0300 (072 to 077)

That is, a byte between ‘\040’ and ‘\171’ inclusive is transmitted as is, and all other bytes are prefixed and modified as shown.

When all the file data is sent, a seven byte sequence is sent: two bytes of ‘\176’ followed by four ASCII bytes of the checksum as printed in base 16 followed by a carriage return. For example, if the checksum was 0x1234, this would be sent: ‘\176\1761234\r’.

The checksum is initialized to 0xffff. For each byte that is sent it is modified as follows (where B is the byte before it has been transformed as described above):

/* Rotate the checksum left. */ if ((ichk & 0x8000) == 0) ichk <<= 1; else { ichk <<= 1; ++ichk; }

/* Add the next byte into the checksum. */ ichk += B;

When the receiving UUCP sees the checksum, it compares it against its own calculated checksum and replies with a single character followed by a carriage return.

‘G’ The file was received correctly.

‘R’ The checksum did not match, and the file should be resent from the beginning.

‘Q’ The checksum did not match, but too many retries have occurred and the communication session should be abandoned.

The sending UUCP checks the returned character and acts accordingly.

9 UUCP ‘t’ Protocol

UUCP ‘t’ Protocol

The ‘t’ protocol is intended for use on links which provide reliable end-to-end connections, such as TCP. It does no error checking or flow control, and requires an eight bit clear channel.

I believe the ‘t’ protocol originated in BSD versions of UUCP.

When a UUCP package transmits a command, it first gets the length of the command string, C. It then sends ‘((C / 512) + 1) * 512’ bytes (the smallest multiple of 512 which can hold C bytes plus a null byte) consisting of the command string itself followed by trailing null bytes.

When a UUCP package sends a file, it sends it in blocks. Each block contains at most 1024 bytes of data. Each block consists of four bytes containing the amount of data in binary (most significant byte first, the same format as used by the Unix function ‘htonl’) followed by that amount of data. The end of the file is signalled by a block containing zero bytes of data.

10 UUCP ‘e’ Protocol

UUCP ‘e’ Protocol

The ‘e’ protocol is similar to the ‘t’ protocol. It does no flow control or error checking and is intended for use over networks providing reliable end-to-end connections, such as TCP.

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

When a UUCP package transmits a command, it simply sends the command as an ASCII string terminated by a null byte.

When a UUCP package transmits a file, it sends the complete size of the file as an ASCII decimal number. The ASCII string is padded out to 20 bytes with null bytes (i.e. if the file is 1000 bytes long, it sends '1000\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0'). It then sends the entire file.

11 UUCP 'G' Protocol

UUCP 'G' Protocol

The 'G' protocol is used by SVR4 UUCP. It is identical to the 'g' protocol, except that it is possible to modify the window and packet sizes. The SVR4 implementation of the 'g' protocol reportedly is fixed at a packet size of 64 and a window size of 7. Supposedly SVR4 chose to implement a new protocol using a new letter to avoid any potential incompatibilities when using different packet or window sizes.

Most implementations of the 'g' protocol that accept packets larger than 64 bytes will also accept packets smaller than whatever they requested in the 'INITB' packet. The SVR4 'G' implementation is an exception; it will only accept packets of precisely the size it requests in the INITB packet.

12 UUCP 'i' Protocol

UUCP 'i' Protocol

The 'i' protocol was written by Ian Lance Taylor (who also wrote this FAQ). It was first used by Taylor UUCP version 1.04.

It is a sliding window packet protocol, like the 'g' protocol, but it supports bidirectional transfers (i.e., file transfers in both directions simultaneously). It requires an eight bit clear connection. Several ideas for the protocol were taken from the paper 'A High-Throughput Message Transport System' by P. Lauder. I don't know where the paper was published, but the author's e-mail address is 'piers@cs.su.oz.au'. The 'i' protocol does not adopt his main idea, which is to dispense with windows entirely. This is because some links still do require flow control and, more importantly, because using windows sets a limit to the amount of data which the protocol must be able to resend upon request. To reduce the costs of window acknowledgements, the protocol uses a large window and only requires an ack at the halfway point.

Each packet starts with a six byte header, optionally followed by data bytes with a four byte checksum. There are currently five defined packet types ('DATA', 'SYNC', 'ACK', 'NAK', 'SPOS', 'CLOSE') which are described below. Although any packet type may include data, any data provided with an 'ACK', 'NAK' or 'CLOSE' packet is ignored.

Every 'DATA', 'SPOS' and 'CLOSE' packet has a sequence number. The sequence numbers are independent for each side. The first packet sent by each side is always number 1. Each packet is numbered one greater than the previous packet, modulo 32.

Every packet has a local channel number and a remote channel number. For all packets at least one channel number is zero. When a UUCP command is sent to the remote system, it is assigned a non-zero local channel number. All packets associated with that UUCP command sent by the local system are given the selected local channel number. All associated packets sent by the remote system are given the selected number as the remote channel number. This permits each UUCP command to be uniquely identified by the channel number on the originating system, and therefore each UUCP package can associate all file data and UUCP command responses with the appropriate command. This is a requirement for bidirectional UUCP transfers.

The protocol maintains a single global file position, which starts at 0. For each incoming packet, any associated data is considered to occur at the current file position, and the file position is incremented by the amount of data contained. The exception is a packet of type ‘SPOS’, which is used to change the file position. The reason for keeping track of the file position is described below.

The header is as follows:

‘\007’ Every packet begins with ‘^G’.

‘(PACKET << 3) + LOCCHAN’ The five bit packet number combined with the three bit local channel number. ‘DATA’, ‘SPOS’ and ‘CLOSE’ packets use the packet sequence number for the PACKET field. ‘NAK’ packet types use the PACKET field for the sequence number to be resent. ‘ACK’ and ‘SYNC’ do not use the PACKET field, and generally leave it set to 0. Packets which are not associated with a UUCP command from the local system use a local channel number of 0.

‘(ACK << 3) + REMCHAN’ The five bit packet acknowledgement combined with the three bit remote channel number. The packet acknowledgement is the number of the last packet successfully received; it is used by all packet types. Packets which are not sent in response to a UUCP command from the remote system use a remote channel number of 0.

‘(TYPE << 5) + (CALLER << 4) + LEN1’ The three bit packet type combined with the one bit packet direction combined with the upper four bits of the data length. The packet direction bit is always 1 for packets sent by the calling UUCP, and 0 for packets sent by the called UUCP. This prevents confusion caused by echoed packets.

LEN2 The lower eight bits of the data length. The twelve bits of data length permit packets ranging in size from 0 to 4095 bytes.

CHECK The exclusive or of the second through fifth bytes of the header. This provides an additional check that the header is valid.

If the data length is non-zero, the packet is immediately followed by the specified number of data bytes. The data bytes are followed by a four byte CRC 32 checksum, with the most significant byte first. The CRC is calculated over the contents of the data field.

The defined packet types are as follows:

0 ‘DATA’ This is a plain data packet.

1 ‘SYNC’ ‘SYNC’ packets are exchanged when the protocol is initialized, and are described further below. ‘SYNC’ packets do not carry sequence numbers (that is, the PACKET field is ignored).

2 ‘ACK’ This is an acknowledgement packet. Since ‘DATA’ packets also carry packet acknowledgements, ‘ACK’ packets are only used when one side has no data to send. ‘ACK’ packets do not carry sequence numbers.

3 ‘NAK’ This is a negative acknowledgement. This is sent when a packet is received incorrectly, and means that the packet number appearing in the PACKET field must be resent. ‘NAK’ packets do not carry sequence numbers (the PACKET field is already used).

4 ‘SPOS’ This packet changes the file position. The packet contains four bytes of data holding the file

position, most significant byte first. The next packet received will be considered to be at the named file position.

5 'CLOSE' When the protocol is shut down, each side sends a 'CLOSE' packet. This packet does have a sequence number, which could be used to ensure that all packets were correctly received (this is not needed by UUCP, however, which uses the higher level 'H' command with an 'HY' response).

When the protocol starts up, both systems send a 'SYNC' packet. The 'SYNC' packet includes at least three bytes of data. The first two bytes are the maximum packet size the remote system should send, most significant byte first. The third byte is the window size the remote system should use. The remote system may send packets of any size up to the maximum. If there is a fourth byte, it is the number of channels the remote system may use (this must be between 1 and 7, inclusive). Additional data bytes may be defined in the future.

The window size is the number of packets that may be sent before a packet is acknowledged. There is no requirement that every packet be acknowledged; any acknowledgement is considered to acknowledge all packets through the number given. In the current implementation, if one side has no data to send, it sends an 'ACK' when half the window is received.

Note that the 'NAK' packet corresponds to the unused 'g' protocol 'SRJ' packet type, rather than to the 'RJ' packet type. When a 'NAK' is received, only the named packet should be resent, not any subsequent packets.

Note that if both sides have data to send, but a packet is lost, it is perfectly reasonable for one side to continue sending packets, all of which will acknowledge the last packet correctly received, while the system whose packet was lost will be unable to send a new packet because the send window will be full. In this circumstance, neither side will time out and one side of the communication will be effectively shut down for a while. Therefore, any system with outstanding unacknowledged packets should arrange to time out and resend a packet even if data is being received.

Commands are sent as a sequence of data packets with a non-zero local channel number. The last data packet for a command includes a trailing null byte (normally a command will fit in a single data packet). Files are sent as a sequence of data packets ending with one of length zero.

The channel numbers permit a more efficient implementation of the UUCP file send command. Rather than send the command and then wait for the 'SY' response before sending the file, the file data is sent beginning immediately after the 'S' command is sent. If an 'SN' response is received, the file send is aborted, and a final data packet of length zero is sent to indicate that the channel number may be reused. If an 'SY' reponse with a file position indicator is received, the file send adjusts to the file position; this is why the protocol maintains a global file position.

Note that the use of channel numbers means that each UUCP system may send commands and file data simultaneously. Moreover, each UUCP system may send multiple files at the same time, using the channel number to disambiguate the data. Sending a file before receiving an acknowledgement for the previous file helps to eliminate the round trip delays inherent in other UUCP protocols.

13 UUCP 'j' Protocol

UUCP 'j' Protocol

The 'j' protocol is a variant of the 'i' protocol. It was also written by Ian Lance Taylor, and first appeared in Taylor UUCP version 1.04.

The ‘j’ protocol is a version of the ‘i’ protocol designed for communication links which intercept a few characters, such as XON or XOFF. It is not efficient to use it on a link which intercepts many characters, such as a seven bit link. The ‘j’ protocol performs no error correction or detection; that is presumed to be the responsibility of the ‘i’ protocol.

When the ‘j’ protocol starts up, each system sends a printable ASCII string indicating which characters it wants to avoid using. The string begins with the ASCII character ‘^’ (octal 136) and ends with the ASCII character ‘ ’ (octal 176). After sending this string, each system looks for the corresponding string from the remote system. The strings are composed of escape sequences: ‘\ooo’, where ‘o’ is an octal digit. For example, sending the string ‘^\021\023 ’ means that the ASCII XON and XOFF characters should be avoided. The union of the characters described in both strings (the string which is sent and the string which is received) is the set of characters which must be avoided in this conversation. Avoiding a printable ASCII character (octal 040 to octal 176, inclusive) is not permitted.

After the exchange of characters to avoid, the normal ‘i’ protocol start up is done, and the rest of the conversation uses the normal ‘i’ protocol. However, each ‘i’ protocol packet is wrapped to become a ‘j’ protocol packet.

Each ‘j’ protocol packet consists of a seven byte header, followed by data bytes, followed by index bytes, followed by a one byte trailer. The packet header looks like this:

‘^’ Every packet begins with the ASCII character ‘^’, octal 136.

HIGH LOW These two characters give the total number of bytes in the packet. Both HIGH and LOW are printable ASCII characters. The length of the packet is ‘(HIGH - 040) * 0100 + (LOW - 040)’, where ‘040 <= HIGH < 0177’ and ‘040 <= LOW < 0140’. This permits a length of 6079 bytes, but there is a further restriction on packet size described below.

‘=’ The ASCII character ‘=’, octal 075.

DATA-HIGH DATA-LOW These two characters give the total number of data bytes in the packet. The encoding is as described for HIGH and LOW. The number of data bytes is the size of the ‘i’ protocol packet wrapped inside this ‘j’ protocol packet.

‘@’ The ASCII character ‘@’, octal 100.

The header is followed by the number of data bytes given in DATA-HIGH and DATA-LOW. These data bytes are the ‘i’ protocol packet which is being wrapped in the ‘j’ protocol packet. However, each character in the ‘i’ protocol packet which the ‘j’ protocol must avoid is transformed into a printable ASCII character (recall that avoiding a printable ASCII character is not permitted). Two index bytes are used for each character which must be transformed.

The index bytes immediately follow the data bytes. The index bytes are created in pairs. Each pair of index bytes encodes the location of a character in the ‘i’ protocol packet which was transformed to become a printable ASCII character. Each pair of index bytes also encodes the precise transformation which was performed.

When the sender finds a character which must be avoided, it will transform it using one or two operations. If the character is 0200 or greater, it will subtract 0200. If the resulting character is less than 020, or is equal to 0177, it will xor by 020. The result is a printable ASCII character.

The zero based byte index of the character within the ‘i’ protocol packet is determined. This index is turned into a two byte printable ASCII index, INDEX-HIGH and INDEX-LOW, such that the index is ‘(INDEX-

HIGH - 040) * 040 + (INDEX-LOW - 040)’. INDEX-LOW is restricted such that ‘040 <= INDEX-LOW < 0100’. INDEX-HIGH is not permitted to be 0176, so ‘040 <= INDEX-HIGH < 0176’. INDEX-LOW is then modified to encode the transformation:

* If the character transformation only had to subtract 0200, then INDEX-LOW is used as is.

* If the character transformation only had to xor by 020, then 040 is added to INDEX-LOW.

* If both operations had to be performed, then 0100 is added to INDEX-LOW. However, if the value of INDEX-LOW was initially 077, then adding 0100 would result in 0177, which is not a printable ASCII character. For that special case, INDEX-HIGH is set to 0176, and INDEX-LOW is set to the original value of INDEX-HIGH.

The receiver decodes the index bytes as follows (this is the reverse of the operations performed by the sender, presented here for additional clarity):

* The first byte in the index is INDEX-HIGH, and the second is INDEX-LOW.

* If ‘040 <= INDEX-HIGH < 0176’, the index refers to the data byte at position ‘(INDEX-HIGH - 040) * 040 + INDEX-LOW % 040’.

* If ‘040 <= INDEX-LOW < 0100’, then 0200 must be added to indexed byte.

* If ‘0100 <= INDEX-LOW < 0140’, then 020 must be xor’ed to the indexed byte.

* If ‘0140 <= INDEX-LOW < 0177’, then 0200 must be added to the indexed byte, and 020 must be xor’ed to the indexed byte.

* If ‘INDEX-HIGH == 0176’, the index refers to the data byte at position ‘(INDEX-LOW - 040) * 040 + 037’. 0200 must be added to the indexed byte, and 020 must be xor’ed to the indexed byte.

This means the largest ‘i’ protocol packet which may be wrapped inside a ‘j’ protocol packet is ‘(0175 - 040) * 040 + (077 - 040) == 3007’ bytes.

The final character in a ‘j’ protocol packet, following the index bytes, is the ASCII character ‘ ’ (octal 176).

The motivation behind using an indexing scheme, rather than escape characters, is to avoid data movement. The sender may simply add a header and a trailer to the ‘i’ protocol packet. Once the receiver has loaded the ‘j’ protocol packet, it may scan the index bytes, transforming the data bytes, and then pass the data bytes directly on to the ‘i’ protocol routine.

14 UUCP ‘x’ Protocol

UUCP ‘x’ Protocol

The ‘x’ protocol is used in Europe (and probably elsewhere) with machines that contain an builtin X.25 card and can send eight bit data transparently across X.25 circuits, without interference from the X.28 or X.29 layers. The protocol sends packets of 512 bytes, and relies on a write of zero bytes being read as zero bytes without stopping communication. It first appeared in the original System V UUCP implementation.

15 UUCP ‘y’ Protocol

UUCP ‘y’ Protocol

The ‘y’ protocol was developed by Jorge Cwik for use in FX UUCICO, a PC uucico program. It is designed for communication lines which handle error correction and flow control. It requires an eight bit clean connection. It performs error detection, but not error correction: when an error is detected, the line is dropped. It is a streaming protocol, like the ‘f’ protocol; there are no packet acknowledgements, so the protocol is efficient over a half-duplex communication line such as PEP.

Every packet contains a six byte header:

sequence low byte sequence high byte A two byte sequence number, in little endian order. The first sequence number is 0. Since the first packet is always a sync packet (described below) the sequence number of the first data packet is always 1. Each system counts sequence numbers independently.

length low byte length high byte A two byte data length, in little endian order. If the high bit of the sixteen bit field is clear, this is the number of data bytes which follow the six byte header. If the high bit is set, there is no data, and the length field is a type of control packet.

checksum low byte checksum high byte A two byte checksum, in little endian order. The checksum is computed over the data bytes. The checksum algorithm is described below. If there are no data bytes, the checksum is sent as 0.

When the protocol starts up, each side must send a sync packet. This is a packet with a normal six byte header followed by data. The sequence number of the sync packet should be 0. Currently at least four bytes of data must be sent with the sync packet. Additional bytes should be ignored. They are defined as follows:

version The version number of the protocol. Currently this must be 1. Larger numbers should be ignored; it is the responsibility of the newer version to accommodate the older one.

packet size The maximum data length to use divided by 256. This is sent as a single byte. The maximum data length permitted is 32768, which would be sent as 128. Customarily both systems will use the same maximum data length, the lower of the two requested.

flags low byte flags high byte Two bytes of flags. None are currently defined. These bytes should be sent as 0, and ignored by the receiver.

A length field with the high bit set is a control packet. The following control packet types are defined:

0xfffe ‘YPKT_ACK’ Acknowledges correct receipt of a file.

0xfffd ‘YPKT_ERR’ Indicates an incorrect checksum.

0xfffc ‘YPKT_BAD’ Indicates a bad sequence number, an invalid length, or some other error.

If a control packet other than ‘YPKT_ACK’ is received, the connection is dropped. If a checksum error is detected for a received packet, a ‘YPKT_ERR’ control packet is sent, and the connection is dropped. If a packet is received out of sequence, a ‘YPKT_BAD’ control packet is sent, and the connection is dropped.

The checksum is initialized to 0xffff. For each data byte in a packet it is modified as follows (where B is the byte before it has been transformed as described above):

/* Rotate the checksum left. */ if ((ichk & 0x8000) == 0) ichk <<= 1; else { ichk <<= 1; ++ichk; }

/* Add the next byte into the checksum. */ ichk += B;

This is the same algorithm as that used by the ‘f’ protocol.

A command is sent as a sequence of data packets followed by a null byte. In the normal case, a command will fit into a single packet. The packet should be exactly the length of the command plus a null byte. If the command is too long, more packets are sent as required.

A file is sent as a sequence of data packets, ending with a zero length packet. The data packets may be of any length greater than zero and less than or equal to the maximum permitted packet size specified in the initial sync packet.

After the zero length packet ending a file transfer has been received, the receiving system sends a ‘YPKT_ACK’ control packet. The sending system waits for the ‘YPKT_ACK’ control packet before continuing; this wait should be done with a large timeout, since there may be a considerable amount of data buffered on the communication path.

16 UUCP ‘d’ Protocol

UUCP ‘d’ Protocol

The ‘d’ protocol is apparently used for DataKit muxhost (not RS-232) connections. No file size is sent. When a file has been completely transferred, a write of zero bytes is done; this must be read as zero bytes on the other end.

17 UUCP ‘h’ Protocol

UUCP ‘h’ Protocol

The ‘h’ protocol is apparently used in some places with HST modems. It does no error checking, and is not that different from the ‘t’ protocol. I don’t know the details.

18 UUCP ‘v’ Protocol

UUCP ‘v’ Protocol

The ‘v’ protocol is used by UUPC/extended, a PC UUCP program. It is simply a version of the ‘g’ protocol which supports packets of any size, and also supports sending packets of different sizes during the same conversation. There are many ‘g’ protocol implementations which support both, but there are also many which do not. Using ‘v’ ensures that everything is supported.

19 Thanks

Besides the papers and information acknowledged at the top of this article, the following people have contributed help, advice, suggestions and information: Earle Ake 513-429-6500 *ake@Dayton.SAIC.COM* chris@uuplus.com (Christopher J. Ambler) jhc@iscp.bellcore.com (Jonathan Clark) jorge@laser.satlink.net (Jorge Cwik) celit!billd@UCSD.EDU (Bill Davidson) "Drew Derbyshire" *ahd@kew.com* erik@pdnfido.fidonet.org Matthew Farwell *dylan@ibmpcug.co.uk* dgilbert@gamiga.guelphnet.dweomer.org (David Gilbert) kherron@ms.uky.edu (Kenneth Herron) Mike Ipatow fido.itc.e-burg.su> Romain Kang *romain@pyramid.com* "Jonathan I. Kamens" *jik@GZA.COM* "David J. MacKenzie" *djm@eng.umd.edu* jum@helios.de (Jens-Uwe Mager) peter@xpoint.ruessel.sub.org (Peter Mandrella) david nugent *david@csource.oz.au* Stephen.Page@prg.oxford.ac.uk joey@tessi.UUCP (Joey Pruett) James Revell *revell@uunet.uu.net* Larry Rosenman *ler@lerami.lerctr.org* Rich Salz *rsalz@bbn.com*

evesg@etlrips.etl.go.jp (Gjoen Stein) kls@ditka.Chicago.COM (Karl Swartz) Dima Volodin *dvv@hq.demos.su* John.Woods@proteon.com (John Woods) jon@console.ais.org (Jon Zeeff) Eric Ziegast *ziegast@uunet.uu.net*