GNU sharutils, version 4.2c

A set of shell archiver utilities
Edition 4.2c, 10 October 2000

Jan Djärv
Francois Pinard

1 Introduction to this toolset

GNU uuencode and uudecode have an history which roots are lost in ages, and we will not even try to trace it. The current versions were brought into GNU by Ian Lance Taylor, and later modernized by Ulrich Drepper. GNU shar surely has a long history, too. All along this long road, numerous users contributed various improvements. The file 'THANKS' in the distribution, as far as we know, contain the names of all contributors we could identify, and for which email addresses are seemingly valid.

Please help us getting the history straight, for the following information is somewhat approximative. James Gosling wrote the public domain shar 1.x. William Davidsen rewrote it as shar 2.x. Warren Tucker implemented modifications and called it shar 3.x. Richard Gumpertz maintained it until 1990. Francois Pinard, from the public domain shar 3.49, made GNU shar 4.x, in 1994. Some modules and other code sections were freely borrowed from other GNU distributions, bringing this shar under the terms of the GNU General Public License.

The few wrapper scripts and the remsync program have been contributed more recently by Francois Pinard, just as an attempt for making this GNU sharutils toolset more useful.

Your feedback helps us to make a better and more portable product. Mail suggestions and bug reports (including documentation errors) for these programs to 'bug-gnu-utils@prep.ai.mit.edu'.

2 The basic `shar` utilities

GNU `shar` makes so-called shell archives out of many files, preparing them for transmission by electronic mail services. A *shell archive* is a collection of files that can be unpacked by `/bin/sh`. A wide range of features provide extensive flexibility in manufacturing shars and in specifying shar *smartness*. For example, `shar` may compress files, uuencode binary files, split long files and construct multi-part mailings, ensure correct unsharing order, and provide simplistic checksums. See Section 2.1 [shar invocation], page 2.

GNU `unshar` scans a set of mail messages looking for the start of shell archives. It will automatically strip off the mail headers and other introductory text. The archive bodies are then unpacked by a copy of the shell. `unshar` may also process files containing concatenated shell archives. See Section 2.2 [unshar invocation], page 7.

2.1 Invoking the `shar` program

The format of the `shar` command is one of:

```
shar [ option ] ... file ...
shar -S [ option ] ...
```

In the first form, the file list is given as command arguments. In the second form, the file list is read from standard input. The resulting archive is sent to standard output unless the `-o` option is given.

Options can be given in any order. Some options depend on each other: the `-o` option is required if the `-l` or `-L` option is used. The `-n` option is required if the `-a` option is used. Also see `-V` below.

Some options are special purpose:

`--help` Print a help summary on standard output, then immediately exits.

`--version`
Print the version number of the program on standard output, then immediately exits.

`-q`
`--quiet` Verbose *off* at `shar` time. Messages are usually issued on standard error to let the user follow the progress, while making the archives. This option inhibits these messages.

2.1.1 Selecting files

`-p`
`--intermix-type`
Allow positional parameter options. The options `-M`, `-B`, `-T`, `-z` and `-Z` may be embedded, and files to the right of the option will be processed in the specified mode. Without the `-p` option, embedded options would be interpreted as file names. See Section 2.1.4 [Stocking], page 4, for more information on these options.

`-S`
`--stdin-file-list`
Read list of files to be packed from the standard input rather than from the command line. Input must be one file name per line. This switch is especially useful when the command line will not hold the list of files to be packed. For example:

```
find . -type f -print | shar -S -o /tmp/big.shar
```

If `-p` is specified on the command line, then the options `-M`, `-B`, `-T`, `-z` and `-Z` may be included in the standard input (on a line separate from file names). The maximum number of lines of standard input, file names and options, may not exceed 1024.

2.1.2 Splitting output

`-o` *prefix*
`--output-prefix=`*prefix*
Save the archive to files '*prefix*`.01`' through '*prefix*`.`*nnn*' instead of standard output. This option *must* be used when the `-l` or the `-L` switches are used.

When *prefix* contains any '`%`' character, *prefix* is then interpreted as a `sprintf` format, which should be able to display a single decimal number. When *prefix* does not contain such a '`%`' character, the string '`.%02d`' is internally appended.

`-l` *size*
`--whole-size-limit=`*size*
Limit the output file size to *size* times 1024 bytes but don't split input files. This allows the recipient of the shell archives to unpack them in any order.

`-L` *size*
`--split-size-limit=`*size*
Limit output file size to *size* times 1024 bytes and split files if necessary. The archives created with this option must be unpacked in the correct order. If the recipient of the shell archives wants to put all of them in a single folder, she shall save them in the correct order for `unshar`, used with option `-e`, to unpack them all at once. See Section 2.2 [unshar invocation], page 7.

For people used to saving all the shell archives into a single mail folder, care must be taken to save them in the appropriate order. For those having the appropriate tools (like Masanobu Umeda's `rmailsort` package for GNU Emacs), shell archives can be saved in any order, then sorted by increasing date (or send time) before massive unpacking.

2.1.3 Controlling the shar headers

`-n` *name*
`--archive-name=`*name*
Name of archive to be included in the header of the shar files. Also see the `-a` switch further down.

`-s` *address*
`--submitter=`*address*

The `-s` option allows for overriding the email address for the submitter, for when the default is not appropriate. The automatically determined address looks like '*username*`@`*hostname*'.

`-a`
`--net-headers`

Allows automatic generation of headers:

```
Submitted-by: address
Archive-name: name/partnn
```

The *name* must be given with the `-n` switch. If name includes a '`/`', then '`/part`' isn't used. Thus '`-n xyzzy`' produces:

```
xyzzy/part01
xyzzy/part02
```

while '`-n xyzzy/patch`' produces:

```
xyzzy/patch01
xyzzy/patch02
```

and '`-n xyzzy/patch01.`' produces:

```
xyzzy/patch01.01
xyzzy/patch01.02
```

`-c`
`--cut-mark`

Start the shar with a cut line. A line saying '`Cut here`' is placed at the start of each output file.

2.1.4 Selecting how files are stocked

`-T`
`--text-files`

Treat all files as text, regardless of their contents.

`-B`
`--uuencode`

Treat all files as binary, use `uuencode` prior to packing. This increases the size of the archive. The recipient must have `uudecode` in order to unpack.

> Use of `uuencode` is not appreciated by many on the net, because people like to readily see, by mere inspection of a shell archive, what it is about.

`-M`
`--mixed-uuencode`

Mixed mode. Automatically determine if the files are text or binary and archive correctly. Files found to be binary are uuencoded prior to packing. This option is selected by default.

For a file is considered to be a text file, instead of a binary file, all the following should be true simultaneously:

1. The file does not contain any ASCII control character besides ⟨BS⟩ (backspace), ⟨HT⟩ (horizontal tab), ⟨LF⟩ (new line), ⟨FF⟩ (form feed) or ⟨CR⟩ (carriage return). The last check is needed on MSDOS systems.
2. The file does not contains a ⟨DEL⟩ (delete).
3. The file contains no character with its eighth-bit set.
4. The file, unless totally empty, terminates with a ⟨LF⟩ (newline).
5. No line in the file contains more than 200 characters. For counting purpose, lines are separated by a ⟨LF⟩ (newline).

`-z`
`--gzip`
: Use `gzip` and `uuencode` on all files prior to packing. The recipient must have `uudecode` and `gzip` (used with `-d`) in order to unpack.

  Usage of `-z` in net shars will cause you to be flamed off the earth.

`-g` *level*
`--level-for-gzip=`*level*
: When doing compression, use `-`*level* as a parameter to `gzip`. The `-g` option turns on the `-z` option by default. The default value is 9, that is, maximum compression.

`-Z`
`--compress`
: Use `compress` and `uuencode` on all files prior to packing. The recipient must have `uudecode` and `compress` (used with `-d`) in order to unpack. Option `-C` is a synonymous for `-Z`, but is deprecated.

  Usage of `-Z` in net shars will cause you to be flamed off the earth.

`-b` *bits*
`--bits-per-code=`*bits*
: When doing compression, use `-bx` as a parameter to `compress`. The `-B` option turns on the `-Z` option by default. The default value is 12, foreseeing the memory limitations of some `compress` programs on smallish systems, at `unshar` time.

2.1.5 Protecting against transmission errors

Transmission of shell archives is not always free of errors. So one should make consistency checks on the receiving site. A very simple (and unreliable) method is running the UNIX `wc` tool on the output file. This can report the number of characters in the file.

As one can guess this does not catch all errors. Especially changing of a character value does not change the computed check sum. To achieve this goal better method were invented and standardized. One very strong is MD5 (MD = message digests). This is standardized in RFC 1321. The produced shell scripts do not force the `md5sum` program to be installed on the system. This is necessary because it is not yet part of every UNIX. The program is however not necessary for producing the shell archive.

`-w`
`--no-character-count`
: Do *not* check with ‘`wc -c`’ after unpack. The default is to check.

`-D`
`--no-md5-digest`
Do *not* check with ‘`md5sum`’ after unpack. The default is to check.

`-F`
`--force-prefix`
Prepend the prefix character to every line even if not required. This option may slightly increase the size of the archive, especially if `-B` or `-Z` is used. Normally, the prefix character is ‘`X`’. If the parameter to the `-d` option starts with ‘`X`’, then the prefix character becomes ‘`Y`’.

`-d` *string*
`--here-delimiter=`*string*
Use *string* to delimit the files in the shar instead of ‘`SHAR_EOF`’. This is for those who want to personalize their shar files.

2.1.6 Producing different kinds of shars

`-V`
`--vanilla-operation`
This option produces *vanilla* shars which rely only upon the existence of `echo`, `test` and `sed` in the unpacking environment.

The `-V` disables options offensive to the *network cop* (or *brown shirt*). It also changes the default from mixed mode `-M` to text mode `-T`. Warnings are produced if option `-B`, `-z`, `-Z`, `-p` or `-M` is specified (any of which does or might require `uudecode`, `gzip` or `compress` in the unpacking environment).

`-P`
`--no-piping`
In the shar file, use a temporary file to hold the file to `uudecode`, instead of using pipes. This option is mandatory when you know the unpacking `uudecode` is unwilling to merely read its standard input. Richard Marks wrote what is certainly the most (in)famous of these, for MSDOS :-).

(Here is a side note from the maintainer. Why isn’t this option the default? In the past history of `shar`, it was decided that piping was better, surely because it is less demanding on disk space, and people seem to be happy with this. Besides, I think that the `uudecode` from Richard Marks, on MSDOS, is wrong in refusing to handle `stdin`. So far that I remember, he has the strong opinion that a program without any parameters should give its `--help` output. Besides that, should I say, his `uuencode` and `uudecode` programs are full-featured, one of the most complete set I ever saw. But Richard will not release his sources, he wants to stay in control.)

`-x`
`--no-check-existing`
Overwrite existing files without checking. If neither `-x` nor `-X` is specified, when unpacking itself, the shell archive will check for and not overwrite existing files (unless `-c` is passed as a parameter to the script when unpacking).

`-X`
`--query-user`
Interactively overwrite existing files.

Use of `-X` produces shars which *will* cause problems with some `unshar`-style procedures, particularily when used together with vanilla mode (`-V`). Use this feature mainly for archives to be passed among agreeable parties. Certainly, `-X` is *not* for shell archives which are to be submitted to Usenet or other public networks.

The problem is that `unshar` programs or procedures often feed '`/bin/sh`' from its standard input, thus putting '`/bin/sh`' and the shell archive script in competition for input lines. As an attempt to alleviate this problem, `shar` will try to detect if '`/dev/tty`' exists at the receiving site and will use it to read user replies. But this does not work in all cases, it may happen that the receiving user will have to avoid using `unshar` programs or procedures, and call `/bin/sh` directly. In vanilla mode, using '`/dev/tty`' is not even attempted.

`-m`
`--no-timestamp`
Avoid generating `touch` commands to restore the file modification dates when unpacking files from the archive.

When the timestamp relationship is not preserved, some files like '`configure`' or '`*.info`' may be uselessly remade after unpacking. This is why, when this option is not used, a special effort is made to restore timestamps,

`-Q`
`--quiet-unshar`
Verbose *off* at `unshar` time. Disables the inclusion of comments to be output when the archive is unpacked.

`-f`
`--basename`
Use only the last file name component of each input file name, ignoring any prefix directories. This is sometimes useful when building a shar from several directories, or another directory. If a directory name is passed to `shar`, the substructure of that directory will be restored whether `-f` is specified or not.

2.2 Invoking the unshar program

The format of the `unshar` command is:

`unshar [` *option* `]` ... `[` *file* ... `]`

Each *file* is processed in turn, as a shell archive or a collection of shell archives. If no files are given, then standard input is processed instead.

Options:

`--version`
Print the version number of the program on standard output, then immediately exits.

`--help` Print an help summary on standard output, then immediately exits.

`-d` *directory*
`--directory=`*directory*
Change directory to *directory* before unpacking any files.

`-c`
`--overwrite`
`-f`
`--force` Passed as an option to the shar file. Many shell archive scripts (including those produced by `shar` 3.40 and newer) accepts a `-c` argument to indicate that existing files should be overwritten.

The option `-f` is provided for a more unique interface. Many programs (such as `cp` and `mv`) use this option to trigger the very same action.

`-e`
`--exit-0` This option exists mainly for people who collect many shell archives into a single mail folder. With this option, `unshar` isolates each different shell archive from the others which have been put in the same file, unpacking each in turn, from the beginning of the file towards its end. Its proper operation relies on the fact that many shar files are terminated by a ‘`exit 0`’ at the beginning of a line.

Option `-e` is internally equivalent to `-E "exit 0"`.

`-E` *string*
`--split-at=`*string*
This option works like `-e`, but it allows you to specify the string that separates archives if ‘`exit 0`’ isn’t appropriate.

For example, noticing that most ‘`.signatures`’ have a ‘`--`’ on a line right before them, one can sometimes use ‘`--split-at=--`’ for splitting shell archives which lack the ‘`exit 0`’ line at end. The signature will then be skipped altogether with the headers of the following message.

2.3 Miscellaneous considerations

Here is a place-holder for many considerations which do not fit elsewhere, while not worth a section for themselves.

Be careful that the output file(s) are not included in the inputs or `shar` may loop until the disk fills up. Be particularly careful when a directory is passed to `shar` that the output files are not in that directory (or a subdirectory of that directory).

When a directory is passed to `shar`, it may be scanned more than once, to conserve memory. Therefore, one should be careful to not change the directory contents while `shar` is running.

No attempt is made to restore the protection and modification dates for directories, even if this is done by default for files. Thus, if a directory is given to `shar`, the protection and modification dates of corresponding unpacked directory may not match those of the original.

Use of the `-M` or `-B` options will slow down the archive process. Use of the `-z` or `-Z` options may slow the archive process considerably.

Let us conclude by a showing a few examples of `shar` usage:

```
shar *.c > cprog.shar
shar -Q *.[ch] > cprog.shar
shar -B -l28 -oarc.sh. *.arc
shar -f /lcl/src/u*.c > u.sh
```

The first shows how to make a shell archive out of all C program sources. The second produces a shell archive with all '`.c`' and '`.h`' files, which unpacks silently. The third gives a shell archive of all uuencoded '`.arc`' files, into files '`arc.sh.01`' through to '`arc.sh.`*nnn*'. The last example gives a shell archive which will use only the file names at unpack time.

3 Simple wrappers around `shar`

3.1 The `mailshar` command and arguments

3.2 The `mail-files` command and arguments

3.3 The `find-mailer` command and arguments

4 Remote synchronisation of directories

For using the `remsync` facility, besides `sharutils` of course, you also need `perl`, GNU `tar`, GNU `findutils` and `gzip`, all installed. You also need a `sum` program which is BSD-compatible, for example the one from GNU `textutils`.

The `remsync` program tries to maintain up-to-date copies of whole hierarchy of files over many loosely connected sites, provided there is at least some slow electronic mail between them. It prepares and sends out specially packaged files called *synchronization packages*, and is able to processes them after reception.

There is no *master* site, each site has an equal opportunity to modify files, and modified files are propagated. Among many other commands, the `broadcast` command prepares and sends a synchronization package from the current site to all others, while the `process` command is used to apply synchronization packages locally after reception from remote sites. `remsync` will never send a file to another site without being asked to with the `broadcast` command, and besides the project synchronization state files (always named '`.remsync`'), it will never modify a file locally without being asked to with the `process` command.

The unit of transmission is a file, whatever its size may be. Nothing less than whole files are being transmitted. People deciding to cooperate in keeping a synchronized set of files must have trust each other, as each participant has the power of modifying the contents of files at other sites. When `remsync` is used by a single individual travelling between many sites, as it is often the case, this confidence problem should be easier to resolve :-).

The `process` command will modify a file without asking confirmation, as long as there is no reason to believe that the file has been modified at more than one place. When some confusion arises from the fact many people independently modified a single file, the receiving user of conflicting files will have the duty of resolving them into a merged version. So, the merging has to be done at the site where the discrepancy is observed, from where it is propagated again to others participants. There is no locking mechanism, so people should use other means, like electronic mail, for telling each other what they do, and which part of a project they are working on.

4.1 Quick start at using `remsync`

If you are in a real hurry, you can follow the recipe given here, and postpone studying this manual further. However, we will consider only a simple case. In any case, it is good to read the full example, as it gives a good picture of the overall usage of `remsync`.

For any sizeable project, it might not be convenient to start with one site having it all and the other site having nothing, because this would cause the first synchronization to be huge. It is more practical to move over a copy of the project by other means, might it be diskettes, tapes, or `mailshar`. So let's presume both sites have a copy of the project, not necessarily identical, but close.

For the following example, we presume that under the same domain '`champignac.land`', there are two machines named '`spirou`' and '`fantasio`'. Further, the participating user on '`spirou@spirou.champignac.land`' has '`spirou`' for a login name, and similarily, the participating user on '`fantasio.champignac.land`' has '`fantasio`' for a login name. On the '`spirou`' machine, user '`spirou`' keeps the project under his home, in directory 'spirou-copy', while on the 'fantasio' machine, user 'fantasio' keeps the project under his home, in directory 'fantasio-copy'. Of course, user names might be the same, as well as the directories containing the project. We use different names here just to make the example clearer.

Here is a full transcript of the initialization session, normally executed only once, and slightly edited to make it more suitable for this manual. The example is broken down in little parts, allowing explanations and comments.

```
% cd ~/spirou-copy
% remsync
remsync (format *.*) - GNU sharutils *.*

>> mode init

init>> remote fantasio@fantasio.champignac.land ~/fantasio-copy
* Directory '~/spirou-copy is not ready for synchronization
Should I prepare it for its first time (y/n)? [y]
Please enter a short project description: Zorglub project
What is your full email address, here? [spirou@spirou.champignac.land]
```

These commands prepare the '~/spirou-copy' hierarchy for synchronization. You should be located at the top directory of the hierarchy at the time the command remsync is called.

The 'mode init' command instructs remsync that no files should be sent in the synchronization package, only their checksum. The goal here is to inform the other site of what we have, and what we don't, somewhat disregarding the fact the other site still looks like it has nothing yet.

The remote command is the key in establishing a synchronization link. It has two parameters, the first being the email address of the partner at the other site (as seen from here, if this matters), the second being the location of the directory where the package should reside on the remote site (as seen from there).

Because there is no '.remsync' file in the project's top-level directory, remsync concludes this is a first synchronization, and so, ask a few questions, often telling in square brackets what answer would be implied by a mere ⟨Return⟩ or ⟨Enter⟩. If the default reply seems inappropriate, just give the correct information.

```
init>> broadcast

  Broadcasting to address 'fantasio@fantasio.champignac.land'
  Studying local files for their signature
  Registering file 'file1'
  Registering file 'file2'
  Registering file 'file3'
* There were new registrations, please check them
Should I resume the current command (y/n)? [y]
Mailing shar to fantasio@fantasio.champignac.land
Message queued
  Command 'broadcast' done

init>> quit
```

```
%
```

The `broadcast` command produces an inventory of the project's files at this end, and mail it to the other partners. But before doing so, because some new files were registered into the synchronization, the user is given the opportunity of interrupting the command, if it is felt that some registered file should really not be there.

The `quit` command exits `remsync`, but only once it created the '`.remsync`' file on disk.

Then, on '`fantasio.champignac.land`', user '`fantasio`' will receive the synchronization package, easily recognizable by the fact the string '`.remsync.tar.gz`' appears in the `Subject` header of the message. Let's assume '`fantasio`' saves the whole message as file '`/tmp/synchro-message`'. Then, '`fantasio`' might use the following recipe:

```
% cd /tmp
% unshar synchro-message
uudecoding file .remsync.tar.gz
% remsync process
  Exploding archive '/tmp/.remsync.tar.gz'

  Package being received:
    from address 'spirou@spirou.champignac.land'
    for project 'Zorglub project'
  Visiting directory '~/fantasio-copy', remote was '~/spirou-copy'
  Initializing file '.remsync' from received information
  Studying local files for their signature
  Command 'process' done
```

In that '`remsync process`' call, the `process` command is being given non-interactively, so `remsync` avoids unneeded interactions and exits right away once the command is done. But equivalently, `remsync` might be called without arguments, the `process` command given interactively, and a `quit` command later required to get out of `remsync`.

When receiving a synchronization package, `remsync` should be executed in the directory where the file '`.remsync.tar.gz`' has been unpacked, which might be quite unrelated to the project itself. Here, '`fantasio`' executed `remsync` in '`/tmp/`', while the project resides in '`~/fantasio-project`'. The synchronization package itself contains enough information for `remsync` to automatically visit the proper directory.

After this operation, '`fantasio.champignac.land`' has a '`.remsync`' file in '`~/fantasio-copy`', and the remote synchronization initialization is completed. Either '`spirou`' or '`fantasio`' may then modify files on their respective machine. If '`spirou`' modifies '`file2`' in the project, '`spirou`' may execute:

```
% cd ~/spirou-copy
% remsync broadcast
  Reading configuration for project 'Zorglub project'

  Broadcasting to address 'fantasio@fantasio.champignac.land'
  Studying local files for their signature
  Packaging file 'file2'
shar: Saving file2 (gzipped)
Mailing shar to fantasio@fantasio.champignac.land
```

```
Message queued
  Command ‘broadcast’ done
```

In fact, any time a participant later feel like sending modified files to all partners, s/he just have to change the directory to the top of the project hierarchy, then call '`remsync broadcast`'. Any time a synchronization package is later received, at either end, the receiving user should apply `unshar` to related electronic messages for reconstructing the synchronization package '`.remsync.tar.gz`', then call '`remsync process`' in the directory containing this package.

4.2 The `remsync` command and arguments

At the shell prompt, calling the command `remsync` without any parameters initiates an interactive dialog, in which the user types commands and receives feedback from the program.

The command `remsync`, given at the shell prompt, may have arguments, in which case these arguments taken together form one `remsync` interactive command. However, '`--help`' and '`--version`' options are interpreted especially, with their usual effect in GNU. Once this command has been executed, no more commands are taken from the user and `remsync` terminates execution. This allows for using `remsync` in some kind of batch mode. It is unwise to redirect `remsync` standard input, because user interactions might often be needed in ways difficult to predict in advance.

The two most common usages of `remsync` are the commands:

```
remsync b
remsync p
```

The first example executes the `broadcast` command, which sends synchronization packages to all connected remote sites for the current local directory tree.

The second example executes the `process` command, which studies and complies with a synchronisation package saved in the current directory (not necessarily into the synchronized directory tree), under the usual file name '`remsync.tar.gz`'.

4.3 Automatic mechanisms in the `remsync` program

The following points apply to many of the `remsync` commands. We describe them here once and for all.

- The file '`.remsync`' describes the various properties for the current synchronization. It is kept right in the top directory of a synchronized directory tree. Some commands may be executed without any need for this file. The program waits as far as possible before reading it.
- If the '`.remsync`' file is not found when required, and only then, the user is interactively asked to fill a questionnaire about it.
- If the '`.remsync`' file has been logically modified after having been read, or if it just has been created, the program will save it back on disk. But it will do so only before reading another '`.remsync`' file, or just before exit. A preexisting '`.remsync`' will be renamed to '`.remsync.bak`' before it is rewritten, when this is done, any previous '`.remsync.bak`' file is discarded.

- Many commands refer to previously entered information by repeating this information. For example, one can refer to a particular `scan` statement by entering the wildcard to be scanned by this statement. An alternative method of specifying a statement consists in using the decimal number which appears between square brackets in the result of a `list` command.
- Whenever a site list must be given, it is a space separated list of remote sites. If the list is preceeded by a bang (!), the list is complemented, that is, the sites that will be operated upon are all those *not* appearing in the list. As a special case, if the site list is completely empty, then all sites are selected.

4.4 Commands for `remsync`

Program commands to `remsync` may be given interactively by the user sitten at a terminal. They can come from the arguments of the `remsync` call at the shell level. Internally, the `process` command might obey many sub-commands found in a received synchronization package.

Program commands are given one per line. Lines beginning with a sharp (#) and white lines are ignored, they are meant to increase clarity or to introduce user comments. With only a few exceptions, commands are introduced by a keyword and often contains other keywords. In all cases, the keywords specific to `remsync` may be abbreviated to their first letter. When there are many keywords in succession, the space separating them may be omitted. So the following commands are all equivalent:

```
list remote
l remote
list r
l r
listremote
lr
```

while the following are not legal:

```
l rem
lisremote
```

Below, for clarity, keywords are written in full and separated by spaces. Commands often accept parameters, which are then separated by spaces. All available commands are given in the table. The first few commands do not pre-require the file '`.remsync`'. The last three commands are almost never used interactively, but rather automatically triggered while `process`'ing received synchronization packages.

`?`

Display a quick help summary of available commands.

`!` [*shell-command*]

If *shell-command* has been given, execute it right now as a shell command. When not given, rather start an interactive shell. Exiting from the shell will return to this program. The started shell is taken from the `SHELL` environment variable if set, else `sh` is used.

`quit`
: Leave the program normally and return to the shell.

`abort`
: Leave the program with a nonzero exit status and return to the shell. No attempt is made to save a logically modified '`.remsync`' file.

`visit` *directory*
: Select another synchronized directory tree for any subsequent operation. *directory* is the top directory of the synchronized directory tree.

`process` [*file*]
`list` [*type*]
: List all known statements about some information *type*. Allowable keywords for *type* are `local`, `remote`, `scan`, `ignore` and `files`. The keyword `files` asks for all empty statements (see later). If *type* is omitted, then list all known statements for all types, except those given by `files`.

[`create`] *type value*
: Create a new statement introducing a *value* for a given *type*. Allowable keywords for *type* are `remote`, `scan` and `ignore`. The `create` keyword may be omitted.

  For `create ignore`, when the pattern is preceeded by a bang (`!`), the condition is reversed. That is, only those files which do match the pattern will be kept for synchronization.

`delete` *type value*
: Delete an existing statement supporting some *value* for a given *type*. Allowable keywords for *type* are `remote`, `scan` and `ignore`.

`email` *remote value*
: Modify the electronic mail address associated with some *remote* site, giving it a new *value*. The special `local` keyword for *remote* may be used to modify the local electronic mail address.

`home` *remote value*
: Modify the top directory of the synchronized directory tree associated with some *remote* site, giving it a new *value*. The special `local` keyword for *remote* may be used to modify the local top directory.

`broadcast` *site_list*
: Send by electronic mail an update package to all sites from *site_list*, containing for each site all and only those files which are known to be different between the remote site and here.

`version` *version*
: This command is not meant for interactive use. It establishes the `remsync` version needed to process the incoming commands.

`from` *site_list*
: This command is not really meant for interactive use. The first site from the *site_list* is the remote site which originated the synchronization package. All

the others are all the sites, including here, which were meant to be synchronized by the `broadcast` command that was issued at the originating remote site.

`sum` *file checksum*
: This command is not really meant for interactive use. It declares the *checksum* value of a particular *file* at the originating remote site. Also, if at least one `sum` command is received, then it is guaranteed that the originating remote site sent one `sum` command for each and every file to be synchronized, so any found local file which was not subject of any `sum` command does not exist remotely.

`if` *file checksum packaged*
: This command is not really meant for interactive use. It directs the `remsync` program to check if a local *file* has a given *checksum*. If the checksum agrees, then the local file will be replaced by the *packaged* file, as found in the received synchronization invoice.

4.5 How `remsync` works

How does `remsync` keep track of what is in sync, and what isn't? See Section 4.7 [Xremsync], page 18, for a the documentation on the '`.remsync`' file format. I understand that a mere description of the format does not replace an explanation, but in the meantime, you might guess from the format how the program works.

All files are summarized by a checksum, computed by the `sum` program. There are a few variants of `sum` computing checksums in incompatible ways, under the control of options. `remsync` attempts to retrieve on each site a compatible way to do it, and complains if it cannot.

`remsync` does not compare dates or sizes. Experience shown that the best version of a file is not necessarily the one with the latest timestamp. The best version for a site is the current version on this site, as decided by its maintainer there, and this is this version that will be propagated.

Each site has an idea of the checksum of a file for all other sites. These checksums are not necessarily identical, for sites do not necessarily propagate to all others, and the propagation network maybe incomplete or asymmetrical in various ways.

Propagation is never done unattended. The user on a site has to call `remsync broadcast` to issue synchronization packages for other sites. If this is never done, the local modifications will never leave the site. The user also has to call `remsync process` to apply received synchronization packages. Applying a package does not automatically broadcast it further (maybe this could change?).

If a site A propagates some files to sites B and D, but not C, site B is informed that site D also received these files, and site D is informed that site B also received these files, so they will not propagate again the same files to one another. However, both site B and D are susceptible to propagate further the same files to site C.

It may happen that a site refuses to update a file, or modifies a file after having been received, or merges versions, or whatever. So, sites may have a wrong opinion of the file contents on other sites. These differences level down after a few exchanges, and it is very unlikely that a file would not be propagated when it should have.

This scheme works only when the various people handling the various files have confidence in one each other. If site B modifies a file after having received it from site A, the file will eventually be propagated back to site A. If the original file stayed undisturbed on site A, that is, if `remsync` proves that site B correctly knew the checksum of the original file, then the file will be replaced on site A without any user confirmation. So, the user on site A has to trust the changes made by the user on site B.

If the original file on site A had been modified after having been sent in a synchronization package, than it is the responsibility of the user on site A to correctly merge the local modifications with the modifications observed in the file as received from site B. This responsibility is real, since the merged file will later be propagated to the other sites in an authoritative way.

4.6 Related file formats

4.7 Format of the '`.remsync`' file

The '`.remsync`' file saves all the information a site needs for properly synchronizing a directory tree with remote sites. Even if it is meant to be editable using any ASCII editor, it has a very precise format and one should be very careful while modifying it directly, if ever. The '`.remsync`' file is better handled through the `remsync` program and commands.

The '`.remsync`' file is made up of statements, one per line. Each line begins with a statement keyword followed by a single ⟨TAB⟩, then by one or more parameters. The keyword may be omitted, in this case, the keyword is said to be *empty*, and the line begins immediately with the ⟨TAB⟩. After the ⟨TAB⟩, if there are two parameters or more, they should all be separated with a single space. There should not be any space between the last parameter and the end of line (unless there are explicit empty parameters).

The following table gives the possible keywords. Their order of presentation in the table is also the order of appearance in the '`.remsync`' file.

`remsync`
: This statement identifies the '`.remsync`' format. The only parameter states the file format version.

`local`
: This statement should appear exactly once, and has exactly two parameters. The first parameter gives the electronic mail address the other sites should use for sending synchronization packages here. The second parameter gives the name of the local directory tree to synchronize, in absolute notation.

`remote`
: This statement may appear zero, one or more times. Each occurrence connects the synchronized directory tree to another tree on a remote site. The first parameter gives one electronic mail address where to send remote synchronization packages. The second parameter gives the name of the corresponding directory tree for this remote electronic mail address, in absolute notation.

`scan`
: This statement may appear zero, one or more times. When it does not appear at all, the whole local directory tree will always be scanned, searching for files to synchronize. When the statement appears at least once, the whole local directory tree will not be scanned, but only those files or directories appearing

in one of these statements. Each `scan` statement has exactly one parameter, giving one file or directory to be studied. These are usually given relative to top directory of the local synchronization directory tree. Shell wildcards are acceptable.

`ignore` This statement may appear zero, one or more times. Each occurrence has one parameter giving a regular expression, according to Perl syntax for regular expressions. These *regexps* are applied against each file resulting from the scan. If any of the `ignore` expression matches one of resulting file, the file is discarded and is not subject to remote synchronization.

After all the statements beginning by the previous keywords, the '`.remsync`' file usually contains many statements having the empty keyword. The empty keyword statement may appear zero, one or more times. Each occurrence list one file being remotely synchronized. The first parameter gives an explicit file name, usually given relative to the top directory of the local synchronized directory tree. Shell wildcards are *not* acceptable.

Besides the file name parameter, there are supplementary parameters to each empty keyword statement, each corresponding to one remote statement in the '`.remsync`' file. The second parameter corresponds to the first remote, the third parameter corresponds to the second remote, etc. If there are more remote statements than supplementary parameters, missing parameters are considered to be empty.

Each supplementary parameter usually gives the last known checksum value for this particular file, as computed on its corresponding *remote* site. The parameter contains a dash `-` while the remote checksum is unknown. The checksum value for the *local* copy of the file is never kept anywhere in the '`.remsync`' file. The special value '`666`' indicates a checksum from hell, used when the remote file is known to exist, but for which contradictory information has been received from various sources.

4.8 Format of synchronisation packages

Each synchronisation package is transmitted as a file named '`.remsync.tar.gz`', which has the format of a `tar` archive, further compressed with the `gzip` program. This archive always contains a file named '`.remsync-work/orders`', and zero or more files named '`.remsync-work/1`', '`.remsync-work/2`', etc. It contains no other files. Each numbered file is actually a full, non-modified file pertaining to the hierarchy of the project, as sent from the remote site.

The '`.remsync-work/orders`' file drives the processing of the received synchronization package. This ASCII file format quite closely resembles the '`.remsync`' format, which we do not explain again here. Only the keywords and their associated parameters are different, and there is no empty keyword. The following table gives the possible keywords, in the order where they normally appear.

`format`
`title`
`here`
`remote`
`ignore`
`scan`
: All those keywords are used exactly the same way as within the '`.remsync`' file, and their format is not explained again here. They state the file format, project title, local and possibly many remote identifications and directories, zero or more ignores, zero or more scans; all of these exactly as known to the remote site who created the synchronization package. In particular, the `here` line states the originating site of the package rather than the receiving one; the receiving site should still be described by one of the `remote` lines.

`visit`
: This statement appears exactly once, and has one numeric parameter. It specifies the zero-based index in the list of remote lines above. The index identifies the receiving site, that is, the site to which this package was sent.

`copy`
: This statement appears exactly once, and has one or more numeric parameters. Each specifies a zero-based index in the list of remote lines above. All indices specify the set of all sites who where broadcasted simultaneously, at the time this synchronization package was issued. The index specified by the `visit` line should also be one of the indices of the `copy` lines. The order in which the indices are given is important, as it also establishes the order in which file signatures are listed on the `check` lines below.

`check`
: This statement may appear zero, one or more times. Each occurrence describes one file known to the project at the originating site, and there is exactly one occurrence for each known file in the project. Each `check` line has exactly n+2 parameters, where n is the number of parameters of the `copy` command. The first parameter gives a file name, relative to the top directory. The second parameter gives the file signature for this file, as computed at the originating site. For each remote site presented in the `copy` command, and exactly in the same order, each supplementary parameter gives the originator's idea of the signature for the said file at this remote site. A dash (`-`) replaces the signature for a file known *not* to exist.

`update`
: This statement may appear zero, one or more times. Each occurrence describes what to do with one of the '`.remsync-work/`n' files, distributed within the synchronization package. In fact, there should be exactly as many `update` lines that there are numbered files in the synchronization package. Usually, each `update` line immediately follows the corresponding `check` line, and has exactly three parameters. The first parameter gives a file name in the project, relative to the top level directory of the hierarchy. The second parameter gives a file signature which the said file should have at the receiving site, for it to be replaced safely, with no questions asked (this is the originator's idea of what the file signature *was*, on the receiving site, prior to its replacement). A dash (`-`) replaces this signature for a file known *not* to exist. The third parameter is the number n, which indicates the file '`.remsync-work/`n' in the synchronization

package distribution which should replace the corresponding project file at the receiving site.

4.9 Other means to synchronization

One correspondent thinks that perhaps the news distribution mechanism could be pressed into service for this job. I could have started from C-news, say, instead of from scratch, and have progressively bent C-news to behave like I wanted.

My feeling is that the route was shorter as I did it, from scratch, that it would have been from C-news. Of course, I could have removed the heavy administrative details of C-news: the history and `expire`, the daemons, the `cron` entries, etc., then added the interactive features and specialized behaviors, but all this clean up would certainly have took energies. Right now, non counting the subsidiary scripts and shar/unshar sources, the heart of the result is a single (1200 lines) script written in Perl, which I find fairly more smaller and maintainable than a patched C-news distribution would have been.

4.10 Documentation for obsolete scripts

This is merely a place holder for previous documentation, waiting that I clean it up. You have no interest in reading further down.

4.10.1 mailsync

```
Usage: mailsync [ OPTION ] ... [ EMAIL_ADDRESS ] [ DIRECTORY ]
   or: mailsync [ OPTION ] ... SYNC_DIRECTORY
```

Option -i simply sends a `ihave` package, with no bulk files. Option -n inhibits any destructive operation and mailing.

In the first form of the call, find a synchronisation directory in DIRECTORY aimed towards some EMAIL_ADDRESS, then proceed with this synchronisation directory. EMAIL_ADDRESS may be the name of a file containing a distribution list. If EMAIL_ADDRESS is not specified, all the synchronisation directories at the top level in DIRECTORY are processed in turn. If DIRECTORY is not specified, the current directory is used.

In the second form of the call, proceed only with the given synchronisation directory SYNC_DIRECTORY.

For proceeding with a synchronisation directory, whatever the form of the call was, this script reads the `ident` files it contains to set the local user and directory and the remote user and directory. Then, selected files under the local directory which are modified in regard to the corresponding files in the remote directory are turned into a synchronisation package which is mailed to the remote user.

The list of selected files or directories to synchronize from the local directory are given in the `list` file in the synchronisation directory. If this `list` file is missing, all files under the local directory are synchronized.

What I usually do is to `cd` at the top of the directory tree to be synchronized, then to type `mailsync` without parameters. This will automatically prepare as many synchronisation packages as there are mirror systems, then email multipart shars to each of them. Note that the synchronisation package is not identical for each mirror system, because they do not usually have the same state of synchronisation.

`mailsync` will refuse to work if anything needs to be hand cleaned from a previous execution of `mailsync` or `resync`. Check for some remaining '`_syncbulk`' or '`_synctemp`' directory, or for a '`_syncrm`' script.

```
TODO:
- interrogate the user if 'ident' file missing.
- automatically construct the local user address.
- create the synchronisation directory on the fly.
- avoid duplicating work as far as possible for multiple sends.
- have a quicker mode, depending on stamps, not on checksums.
- never send core, executables, backups, '.nsf*', '*/_synctemp/*', etc.
```

4.10.2 resync

```
Usage: resync [ OPTION ]... TAR_FILE
   or: resync [ OPTION ]... UNTARED_DIRECTORY
```

Given a tar file produced by mailsync at some remote end and already reconstructed on this end using unshar, or a directory containing the already untared invoice, apply the synchronization package locally.

Option -n inhibits destroying or creating files, but does everything else. It will in particular create a synchronization directory if necessary, produce the '`_syncbulk`' directory and the '`_syncrm`' script.

The synchronization directory for the package is automatically retrieved or, if not found, created and initialized. `resync` keeps telling you what it is doing.

There are a few cases when a resync should not complete without manual intervention. The common case is that several sites update the very same files differently since they were last resync'ed, and then mailsync to each other. The prerequisite checksum will then fail, and the files are then kept into the '`_syncbulk`' tree, which has a shape similar to the directory tree in which the files where supposed to go. For GNU Emacs users, a very handy package, called emerge, written by Dale Worley <drw@kutta.mit.edu>, helps reconciling two files interactiveley. The '`_syncbulk`' tree should be explicitely deleted after the hand synchronisation.

Another case of human intervention is when files are deleted at the mailsync'ing site. By choice, all deletions on the receiving side are accumulated in a '`_syncrm`' script, which is not executed automatically. Explicitely executed, '`_syncrm`' will remove any file in the receiving tree which does not exist anymore on the sender system. I often edit '`_syncrm`' before executing it, to remove the unwanted deletions (beware the double negation :-). The script removes itself.

All the temporary files, while resynchronizing, are held in '`_synctemp`', which is deleted afterwards; if something goes wrong, this directory should also be cleaned out by hand. `resync` will refuse to work if anything remains to be hand cleaned.

TODO:
- interrogates the user if missing receiving directory in '`ident`'.
- allow '`remote.sum`' to be empty or non-existent.

Table of Contents