TR-CS-96-03

autoson – a distributed batch system for UNIX workstation networks (version 1.3)

Brendan D. McKay

March 1996

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

Table of Contents.

Some of the development of `autoson` was performed within the Advanced Computation Systems Cooperative Research Centre, Numerical Algorithms project.

The latest version of `autoson`, including this manual, will always be available via my home page: `http://cs.anu.edu.au/~bdm/index.html`

1. Introduction.

`autoson` is a tool for scheduling processes across a network of UNIX workstations. It provides a type of distributed batch queue that enables execution of a stream of processes in a flexible and convenient manner with minimum impact on interactive users.

`autoson` can be compiled in two "modes". In *single-user mode*, support is given for a single user who wishes to execute processes on one or more workstations. In *multiuser mode*, several users can cooperatively use the same queue.

`autoson` can execute independent processes on heterogeneous and/or unreliable processors. It does not provide a parallel environment that allows processes to communicate, it just executes them and waits for them to finish. If you want closely cooperating processes you need to use some other product.

The flavours of UNIX that support version 1.3 of `autoson` are:

(i) Solaris One and Solaris Two on Sun workstations;
(ii) IRIX on Silicon Graphics mips workstations;
(iii) OSF1 on Dec Alpha workstations;
(iv) ULTRIX on Decstations;
(v) Hewlett-Packard HP-UX computers;
(vi) The Linux operating system.

`autoson` can exist happily on a network containing any mixture of these.

`autoson` is the son of a program called `automan`, which ran on a cluster of VAXes. It was named after a short-lived American television series which featured an electronic super-hero.

2. Entries, states, sequences and streams.

`autoson` works by maintaining a file called the "queuefile". In single-user mode, the queuefile is usually called `autoson.queue` and resides in your home directory. In multiuser mode, the name of the queuefile is fixed at compile time. Both of these conventions can be changed.

The queuefile must be visible from all the machines on which you intend to run `autoson`. The queuefile contains a queue of tasks called "entries". Each entry has a unique entry number and is in one of six states:

PENDING : The entry is ready to run, unless it is waiting for other entries to finish.

HOLDING : The entry is on hold indefinitely.

WAITING : The entry is waiting for a particular time to arrive, at which moment it will be moved to state PENDING.

CURRENT : The entry is being executed.

LOST : The entry was discovered in state CURRENT but the processes that were running it appear to have vanished. Usually this is due to a machine crash.

SICK : The entry could not be run due to an error. Currently the errors detected are: bad directory, bad logfile, and exit(101) executed by child process. These are explained more below.

Usually we abbreviate the state names to four characters.

The basic mode of operation is as follows. On each machine in the network, the program `aurun` runs continuously. It repeatedly selects one entry from the queue and executes it. When it finishes, it selects another, and so on. Entries can be inserted into the queue from any machine.

In general, an entry represents one or more jobs to run, as determined by the parameters `cycle`, `step` and `limit`. It defines a "sequence" of jobs which are the same except that they have a "sequence number" that runs through the list `cycle`, `cycle`+`step`, ..., `limit`. You can control the number of members of a sequence that can execute at once. The program that is actually run can be made to depend on the value of the sequence number.

Entries have a large number of "attributes". Below is a complete list, even though the meanings of some will not become clear until later. In each case we give the attribute name, the type of its value, the default (if it has one) and a brief description. The type *unsigned* means a non-negative integer. Some of the defaults can be changed during compilation.

`entrynumber` (*unsigned*)
This is just a number that identifies the entry. Entry numbers are never reused, unless you edit the queuefile manually.

`state` (*state*)
The state, one of the six possibilities described above.

`cycle` (*unsigned, default*=1)
`step` (*integer, default*=1)
`limit` (*unsigned* or ∞*, default*=`cycle`)
`oldlimit` (*unsigned* or ∞)
These four attributes describe parameters of the sequence to which the entry belongs, and the position of this entry in the sequence. When a new entry is created by the user, `cycle` is set to the first cycle number of the sequence, and `limit` and `oldlimit` are both set to the last cycle number (or to ∞ if there is no last). When `aurun` selects a PEND entry to run, the value of `cycle` becomes the value of the macro character `#` (see Section 10). If $\texttt{step} \geq 0$ and $\texttt{cycle} + \texttt{step} \leq \texttt{limit}$, or $\texttt{step} < 0$ and $\texttt{cycle} + \texttt{step} \geq \texttt{limit}$, a new entry like this one is then created, with state PEND (or WAIT—see `resubdelay`) and `cycle` incremented by `step`. It belongs to the same sequence. Then the original entry is put into CURR state with `limit` set equal to `cycle`. `oldlimit` is not changed, so normally it will remain equal to the initial value of `limit`. The purpose of `oldlimit` is merely to remember the initial value of `limit`, for use with the `=` macro character (Section 10).

`stream` (*string, default*=none)
If this attribute has a non-empty value, it is the name of a stream to which the entry belongs. A "stream" is a set of entries with no particular structure except having the same stream name. Streams have two significant uses. One is that the number of simultaneously executing members of a stream can be limited (see `maximum`), and the other is that `aurun` can be asked to restrict itself to one stream. If the value of `stream` is the empty string, the entry is not regarded as belonging to any stream.

`username` (*string*)
: This is the username of the user who created the entry. In single-user mode, it is not used for anything; entries are executed using the uid of the `aurun` process. In multiuser mode, it defines who has permission to modify the entry and the uid used for running it. More details are given in Section MU.

`priority` (*unsigned, default*=10)
: This is the priority for the `autoson` queue. All else being equal, the entry with the greatest priority will be executed first. If there is more than one, the lowest entry number is preferred.

`schedtime` (*unsigned*)

`serialize` ($\{no, maybe, yes\}$, *default*=*maybe*)
: These are attributes used for scheduling in multiuser mode. The value of the `schedtime` attribute is taken from a global counter G, which increases monotonically over the life of the queue. When an entry is created, its `schedtime` attribute is initialized to the current value of G. When an entry is selected for execution, and it either has `serialize` = 1, or it has `serialize` = 0 and the `aurun` process has `-serialize`, then all entries in states PEND, WAIT, HOLD or LOST owned by the same user have their `schedtime` attributes set to the current value of G. The usage of `schedtime` is as follows. When an entry is to be chosen for execution and there is more than one that are eligible but have the same `priority` attribute, one with those with the lowest `schedtime` is chosen. If there is still more than one choice, the one with the lowest `entrynumber` is chosen.

`nice` (*unsigned, default*=20)
: This is the UNIX niceness that will be used when the entry is run. It is an increment relative to the niceness of the `aurun` process, in the manner of the shell command `nice`. Larger values mean less urgent.

`maximum` (*unsigned, default*=0)
: In order for an entry in PEND to be selected for running, the number of CURR or LOST entries in the same sequence or same stream must be less than `maximum`. However a value of 0 indicates that there is no restriction.

`resubdelay` (*unsigned, default*=0)
: When a new member of a sequence is automatically created as described under `cycle`, it is put into state PEND only if `resubdelay`=0. Otherwise, it is put into state WAIT until `resubdelay` seconds have passed.

`logfile` (*string, default*=~/autoson.@@.log)

`actlogfile` (*string*)
: These attributes refer to the file, called the "logfile", which receives the output (`stdout` and `stderr` combined) of the entry when it is executed. Every entry has the `logfile` attribute, but only entries in state CURR, LOST or SICK have the `actlogfile` attribute. The value of `logfile` is a string which becomes the name of the logfile when the macro characters are replaced by their values. (For example `@@` is the name of the machine executing the entry, followed by the last character from the name of the `aurun` process.) It is relative to the directory given by the `directory` attribute unless it begins with `~` or `/`. The value of `actlogfile` is the fully-expanded name of the logfile in actual use. The relationship between `logfile` and `actlogfile` is explained in detail in Sections 10 and 11.

`append` (*boolean, default=false*)
: This specifies whether an existing logfile should be appended to or overwritten.

`umask` (*integer, default=*-1)
: This is the value of *umask* used for creation of the entry logfile. It is also the initial *umask* value for the processes that will run the entry. A value of -1 means to inherit *umask* from the `aurun` process. Only the low-order 9 bits are used. *umask* is a UNIX parameter affecting the permissions on files. See your friendly UNIX manual.

`directory` (*string*)
`actdirectory` (*string*)
: These attributes refer to the initial directory to be in when the entry is executed, and the directory to which `logfile` is relative unless it is an absolute path name. Every entry has the `directory` attribute, but only entries in state CURR, LOST or SICK have the `actdirectory` attribute. The value of `directory` is a string which becomes the directory name when macro characters are replaced by their values. It is relative to your home directory unless it begins with `~` or `/`. The default is the directory which you are in when you create a new entry using the `auadd` command. The value of `actdirectory` is the fully-expanded actual directory name. `aurun` will "cd" to there when it runs the entry. The relationship between `directory` and `actdirectory` is explained in detail in Sections 10 and 11.

`submittime` (*time*)
`timestamp` (*time*)
: These are two times associated with the entry. `submittime` is the time when the entry was created, whether manually by `auadd` or `aumod -copy`, or automatically as described under `cycle`. `timestamp` is initially equal to `submittime`, but if the entry is in state WAIT it gives the time which it is waiting for. It is also reset to the current time when an entry is selected for running.

`prerequisites` (*entry-list, default=*empty)
: This attribute specifies the entry numbers of entries which must be gone from the queuefile before this entry can be executed. If the entry number of this entry itself is present, it has no effect.

`identification` (*string, default=*empty)
: This is just a string that has no effect except to label the entry if you display the queue.

`hosts` (*hostgroup, default=*empty)
: This specifies a set of machines that are allowed to run this entry. The default is that any machine can run it. The value is a structured set called a "host group" which is defined in detail in Sections 10 and 11.

`nexthosts` (*string, default=*none)
: If this entry is moved to PEND or LOST as a result of being found abandoned, the value of the `hosts` attribute will be modified if this attribute is non-null. If the value is "`=same`", `hosts` is set equal to the name of the current host. This will force further executions to continue on the same machine. If the value of `nexthosts` is "`=different`", `hosts` has the group "`~`*host*", where *host* is the

name of the current host, prepended. This will force further executions to take place on anywhere that was previously allowed, except for the current host. Any other value of `nexthosts` must be a host name, a host group name starting with `@` or `=`, or one of those preceded by `~`. In that case, its value replaces the value of `hosts`.

`yawn` ({*no,maybe,yes*}, *default=maybe*)
`grace` (*unsigned, default*=10)
`cautious` (*boolean, default=true*)
`loadfactor` (*real, default*=1.0)

“yawn” is a protocol for controlling the execution of processes according to the interactive activity on a a particular machine. Section 12 contains a detailed description. In the case `yawn`=*maybe*, the use of yawn is determined by the presence of the flags `-yawn` or `-noyawn` on the `aurun` command. If neither is present, yawn is used unless the executing host belongs to the hostgroup `@noyawn`. If the yawn protocol is used, `grace` seconds are waited first to give you time to log out before it sees you. An alternative is to tell yawn to ignore you always, as described in Section 12.

If the `cautious` attribute is *true*, `aurun` will not start an entry if it determines that yawn would immediately suspend it. However, the test is imperfect in that it ignores `grace`. More discussion appears in Section 5.

The attribute `loadfactor` is used by the yawn protocol in time class LOAD. It represents the additional load average that a machine can expect when it executes this entry. A more precise description is given in Section 12.

`stopped` ($\{0, 1, 2\}$)

When an entry is in state CURR, this indicates whether it is running or stopped, as far determined from which signals have been sent by the yawn protocol or by `auzap`. Value 0 means running under `aurun` control. Value 1 means stopped under `aurun` control. Value 2 means that apparently this entry still exists but no `aurun` process is controlling it.

For SICK and LOST entries, the value is preserved from the last time the entry was in state CURR. Entries in other states do not have a `stopped` attribute.

`last` (*boolean, default=false*)

When aurun finishes executing this entry, it will exit if `last`=*true*.

`once` (*boolean, default=false*)

If `once`=*true* each `aurun` process will execute at most one member of the sequence to which this entry belongs.

`verbose` (*boolean, default=true*)

If this attribute is *false*, nothing except the output from the entry will appear in the logfile. Otherwise, `aurun` will add some information in brief header and trailer, such as the machine name and cpu time. A more complete description appears in Section 16.

`retries` (-1*, unsigned or* ∞*, default*=0)

Due to machine crashes or other factors, an entry might be left in the queuefile in state CURR even though the processes running it have vanished. If `aurun` notices such an entry, it will process it according to the value of `retries`. If `retries`=−1, the entry will be deleted. If `retries`=0, the entry will be moved into state LOST. If `retries`>0, the entry will be moved into state PEND with the value of `retries` decremented. In other words, non-negative values of `retries` specify one less than the total number of attempts which can be made to execute an entry before giving up. The attribute `limit` will be set equal to `cycle` in all cases.

`freeze` (*special-string, default*=none)
: The value is a string, either `none`, `all` or a string over the alphabet {`c`, `d`, `l`}. These characters refer to the command, directory and logfile parameters. The act of "freezing" involves setting the value of a "generic" attribute (such as `logfile`) equal to the value of the "actual" attribute (such as `actlogfile`), and the value of `freeze` determines which attributes are frozen in particular circumstances. The most important time this decision is made is when an entry is selected for running. A fuller description appears in Section 10.

`runninghost` (*string*)
`runningpid` (*unsigned*)
`aurunpid` (*unsigned*)
: These attributes are only defined for entries in state CURR, LOST or SICK. They give the name of the machine executing the entry, the group-id of the processes executing it, and the pid of the `aurun` process.

`command` (*string-list*)
`actcommand` (*string-list*)
`commandverb` (*string*)
`arguments` (*string-list*)
: The attributes `command` and `actcommand` refer to the command and its arguments, i.e. the UNIX command to be executed on behalf of this entry. The attribute `command`, held by all entries, contains a list of strings that will become the command and its arguments once macro characters are replaced by their values. The actual command and arguments appear in `actcommand`, but only for entries in state CURR, LOST or SICK. `commandverb` and `arguments` are alternative names for the first word, and the remaining words of `command` A more complete description appears in Section 10.

`sickness` (*unsigned*)
`immunity` (*unsigned, default*=0)
: The `sickness` attribute is only defined for entries in state SICK, and indicates the reason for that state. Values defined at the moment are these:
1 = Could not "cd" to the entry directory (see `directory`);
2 = Could not open entry logfile for writing (see `logfile`);
3 = The exit status 101 was returned by the subprocess (see Section 5).
The value of `immunity` is the number of SICK entries allowed in the same stream or sequence. If this limit has been reached or exceeded, this entry will not be executed.

`requeuetime` (*time, default*=0)
: This is a time-valued token used to pass a message from `auzap` to `aurun` in the event that `-requeue` is used. It contains the time that the entry processes were signalled. If it is less than 60 seconds before the present when the entry finishes

executing, `aurun` will put the entry into PEND state instead of deleting it. Most other operations on an entry will zero this attribute.

3. Overview of `autoson` commands.

There are six commands implemented as links to the executable file `autoson`. These are:

`auadd`	add new entries to the queue
`aumod`	modify existing entries
`auzap`	kill the processes running entries
`aurun`	execute entries
`aulook`	examine the queue
`aulock`	lock the queuefile and maybe edit it

There is actually no command `aurun`, but there can be any number of commands whose names consist of "`aurun`" plus one more character. The usual convention is to use names `aurun0`, `aurun1`, etc.. This is done to distinguish between multiple servers on the same machine. Throughout this manual, when we refer to "`aurun`", we mean any of those processes.

Commands take switches that begin with `-`.

The names of switches can be abbreviated as far as they are still unique. For example, `-current` can be given as `-curr` or even `-cu`. If a switch needs an argument, it is not appended to the switch name but given as the next command parameter. If it allows several arguments (a list) they are separated by commas without spaces, for example: `-host sun1,sun2`.

Many switches have positive and negative forms, like `-holding` and `-nohold`. It is permitted to use the same switch several times, or both the positive and negative forms, in which case the rightmost version wins. This is useful if you want to define aliases.

There is one switch allowed on all `autoson` commands, provided that run-time selecting of the queuefile has been enabled. Usually this is enabled for single-user mode and disabled for multiuser mode.

`-queuefile` *string*

The name of the queuefile is determined as follows. Start with the value of `-queuefile`, if present, or the environment variable AUTOSON_QUEUE, if it exists. If neither exists, take the string "`autoson.queue`". Then, if the string does not begin with '`~`' or '`/`', prepend the name of the login directory of the user running this process.

If run-time selection of the queuefile has been disabled, `-queuefile` is illegal.

4. The `auadd` and `aumod` commands.

The `auadd` command adds one entry to the queuefile. You can put it into state PEND (default), WAIT (`-after`) or HOLD (`-holding`).

`auadd` ⟨*switches*⟩ `command [argument]...`

The `aumod` command changes an existing entry, or a copy of an existing entry (`-copy`). The `-entry` switch is compulsory.

`aumod` ⟨*switches*⟩ `[command [argument]...]`

The switches available for `auadd` and `aumod` are listed below. To understand our description of them, it may sometimes be necessary to consult the list of entry attributes in Section 2.

`-append`, `-noappend`
`-cautious`, `-nocautious`
`-freeze` *string*, `-nofreeze`
`-grace` *unsigned*, `-nograce`
`-identification` *string*, `-noidentification`
`-immunity` *unsigned*, `-noimmunity`
`-last`, `-nolast`
`-loadfactor` *real*
`-logfile` *string*, `-nologfile`
`-maximum` *unsigned*, `-nomaximum`
`-nexthosts` *string*, `-nonexthosts`
`-nice` *unsigned*
`-once`, `-noonce`
`-priority` *unsigned*
`-resubdelay` *unsigned*, `-noresubdelay`
`-serialize`, `-noserialize`
`-stream` *string*, `-nostream`
`-umask` *octal*, `-noumask`
`-verbose`, `-noverbose`
`-yawn`, `-noyawn`

These switches specify the values of the entry attributes of the same names. `-nologfile` is the same as `-logfile /dev/null`; otherwise the meanings of the negative switches should be clear. The argument of `-stream` can be null, but can not contain spaces.

`-cycle` *unsigned*
`-limit` *unsigned*, `-nolimit`
`-step` *integer*
`-keeplimit`,
`-nokeeplimit`

If `-nokeeplimit` is present (which it is by default), `-cycle` sets the values of `cycle`, `limit` and `oldlimit`. If `-keeplimit` is present, `-cycle` only sets `cycle`. `-limit` (done after `-cycle` irrespective of which one you give first) changes `limit` and `oldlimit`. `-nolimit` is the same as `-limit` ∞.

`-directory` *string*

Set the value of `directory`. If the value is "`.`" or starts with "./", the period is replaced by the full path name of the current directory. If it does not then start with `/` or `~`, it is assumed relative to your home directory.

`-prerequisites` *entry-list*, `-noprerequisites`

Set the value of `prerequisites`, which is a list of entry numbers. The format of the argument is the same as for `-entry` (see below). Recall that an entry is never a prerequisite for itself; this means you can use `-pre :` to make it wait until it is alone in the queue. ("`:`" specifies all entries, see the description of `-entry` below). Example: `-pre 34,45:62,99:`

`-hosts` *hostgroup*, `-nohosts`

Define the value of the `hosts` attribute. The format of the argument is a comma-separated list with no embedded spaces. Formally it is a host group, with the understanding that an empty list means that every machine is ok. `-nohosts` means the same. See Section 9 for a complete description of host groups. Example: `-hosts sun1,=alpha,~alpha3`

`-after` *time*

Put the entry into state WAIT until the specified time. The syntax of the argument is very flexible. Examples: `17:00`, `5pm`, `"3am tomorrow"`, `"next week"` `"3:30:13 Aug 9"`, `"3/13/1997"`, `"2 hours 15 min"` (from now), `"2am aest"` (Australian Eastern Standard Time). Note that quotes are needed if there are embedded spaces.

`-holding`
`-noholding` (`aumod` only)

`-holding` will put the entry into state HOLD, and `-noholding` will put it into state PEND. The default for `aurun` is PEND. For `aumod`, the default if PEND if `-copy` is specified, and no change of state otherwise. To move an entry from LOST to PEND, it is better to use `-nolost`.

`-nowaiting` (`aumod` only)

If the entry is in state WAIT, move it to state PEND. Otherwise, do nothing.

`-nolost` (`aumod` only)

Move an entry from state LOST to state PEND. Only the entries in state LOST are considered, so you can process all the LOST entries at once using
`aumod -ent : -nolost` .

`-show`, `-peek`, `-some`

`-show` and `-peek` display the queue in verbose and brief format, respectively. Without `-some` the entire queue is displayed, otherwise only interesting entries such as the entries you just inserted or modified. `-some` is ignored unless you have `-show` or `-peek`. The brief format is user-definable, see Section 7.

`-silent`

This turns off informational messages like "`modified entry` xx".

`-volatile`, `-novolatile`
`-rerun`, `-norerun`
`-retries` *unsigned*, `-noretries`

These set the attribute `retries` to -1, 0, ∞, 0, the value of the argument, and

0, respectively.

`-entry` *entry-list* (`aumod` only)
: This is compulsory for the `aumod` command, and specifies the entries which are to be modified or copied. The argument is a comma-separated list with no imbedded spaces. Each item can be an entry number or a range of entry numbers. A range has one of the forms `10:20` (all entries numbered from 10 to 20), `:30` (all entries numbered up to 30), `40:` (all entries numbered 40 or more) or `:` (all entries). Each item in the list must refer to at least one existing entry. Example: `-entry 10,20:23,60:` .

`-user` *string*, `-mine` (`aumod` only)
: Limit the set of entries specified by `-entry` to those owned by the specified user.

`-copy` (`aumod` only)
: Instead of modifying an existing entry (the default behaviour), make a copy of the entry (replacing state CURR by PEND if necessary) then modify the copy.

`-newsequence`, `-nonewsequence` (`aumod` only)
: Sequences are identified by an internal attribute `isn` constructed from the entry number and time when the first entry in a sequence is created. This is copied to all entries formed from that one, including entries formed by `aumod -copy`. The switch `-newsequence` causes `isn` to be recomputed rather than copied.

`-renumber` (`aumod` only)
: After performing any other changes that are requested, renumber all the entries except those in state CURR as entries $1, 2, \ldots$. New entries will have numbers in the new low sequence, so it is important that existing CURR entries are gone before the entry numbers build up to overtake them. Prerequisites are only renumbered if they are simple numbers, not ranges. In other words, this facility is rather dangerous. It is still required to use `-entry`, for consistency, but its value is ignored.

`-delete` (`aumod` only)
`-really` (`aumod` only)
: Delete the specified entries. Normally you can't delete a CURR entry, but if you use `-delete -really` you can delete anything. Deleting CURR entries from the queue will not stop them running. `-delete -really` is intended for those bad moments when `autoson` won't delete something that it should.

Any other parameters are a command and arguments, setting the entry attribute `command`. The command name and arguments must be separate arguments to `auadd` and `aumod`, and cannot contain commas unless they are escaped (as "`\,`").

Specifying a command and arguments with `aumod` will cause the previous ones to be replaced completely. There is no way to change just some part of them.

Examples:

```
auadd -peek -some -log out cmd arg1 arg2 arg3
```

The command `cmd arg1 arg2 arg3` will be run, with output going the file `out` in the current directory. After inserting it in the queue, display it in brief format.

```
auadd -cyc 1 -lim 100 -max 3 -noapp -log xyz.#.out doxyz #
```

For `#` taking the values $1, 2, \ldots, 100$, the command `doxyz #` will be executed with the output going to the file `xyz.#.out` in the current directory, overwriting any existing file of that name. However, at most three of these 100 jobs will be executed at once.

```
    auadd -log pipe.out -nice 0 sh -c 'cmd1 arg; cmd2 | cmd3 >file'
```

This example shows how to run a sequence or pipe. Of course, you can also do it by writing a shell script and running that. In this example the sequence will be executed at interactive priority. The quotes are essential. (In multiuser mode, you can omit the "`sh -c`" from this command.)

```
    aumod -ent 43,44 -pri 11
```

Change the `priority` attribute of entries 43 and 44 to 11.

```
    aumod -ent 399 -copy -hold -cyc 100 -lim 200
```

Take a copy of entry 399, change its `cycle`, `limit` and `oldlimit` attributes, and put it into state HOLD.

```
    aumod -ent 76 newcomm newarg
```

Change the `command` attribute of entry 76.

```
    aumod -ent 400:500 -delete
```

Delete all entries numbered from 400 to 500.

5. The `aurun` command.

`aurun` is our generic name for the program which controls entry execution. In reality, the name will be `aurun0`, `aurun1`, or in general any 6-character name starting with "`aurun`". The normal mode of operation is to have it running continuously. You can run any number of them as background processes on machines that can see the queuefile. It is valid to run more than one on the same machine, provided they have different names.

The operation of `aurun` is to repeatedly select the most eligible entry from the queue and execute it. The criterion for an entry to to be eligible for running is that all these conditions are satisfied:

(1) `state`=PEND;
(2) the values of `stream` and `identification` match the values of `-stream` and `-identification` on the `aurun` command (if any);
(3) `aurun` does not have the `-only` switch, or the argument of `-only` equals the value of `entrynumber`;
(4) `hosts`=*empty*, or the current machine is in the host group `hosts`;
(5) `prerequisites`=*empty*, or there is no entry in the queue (whatever state, but ignoring this entry) matching any term;
(6) `maximum`=0, or there are fewer than `maximum` entries in states CURR or LOST that belong to the same sequence or same stream as this one;
(7) there are not more than `immunity` SICK entries in the same stream or sequence;
(8) `once`=*false*, or this `aurun` process has not previously run an entry in the same sequence which had `once`=*true*;
(9) either `yawn`≠*yes*, or `cautious`=*false*, or this entry would not be immediately stopped (ignoring `grace`) by the yawn protocol.

Amongst those entries eligible for running, the most preferred is the one that has the least `entrynumber` amongst those with the least `schedtime` amongst those with the greatest `priority`.

`aurun` will continue executing entries or waiting for an entry to execute until one of these occurs:

(1) it runs an entry with `last`=*true*;

(2) it runs the single entry specified with `-only`;

(3) it runs as many entries as specified with `-maximum`;

(4) it has the `-clear` switch and there are no entries that could be run even if they had `cautious`=*false*.

While `aurun` has nothing to run it sleeps for periods whose duration can be up to 20 minutes depending on how long it has been since there was anything to do. You can make it wake up straight away by sending it a SIGHUP signal.

The process of running an entry is like this:

(1) If $\texttt{step} \geq 0$ and $\texttt{cycle} + \texttt{step} \leq \texttt{limit}$, or $\texttt{step} < 0$ and $\texttt{cycle} + \texttt{step} \geq \texttt{limit}$, create a new entry in this sequence in state PEND or WAIT, as described in Section 2 (attribute `cycle`).

(2) Compute the values of `actdirectory`, `actlogfile` and `actcommand`, as explained in Section 11.

(3) Open a file `actlogfile` for writing, creating it if it doesn't exists already. If that cannot be done, put the entry into state SICK with `sickness`=2 and give up trying to run this entry.

(4) Go to the directory `actdirectory`. If that cannot be done, put the entry into state SICK with `sickness`=2 and give up running this entry.

(5) Start the command `actcommand` in a subprocess. (More details at the end of this section.)

(6) Wait for the command to finish, using the yawn protocol if appropriate. If it finishes with exit status 101, put the entry into state SICK with `sickness`=3; otherwise, if it has a `requeuetime` value less than 60 seconds old, put the entry into state PEND with `last`=*false*; otherwise, delete the entry.

The main use of exit status 101 in step (7) is by `aurun` itself, in the event that you try to execute a nonexistent command, or one without execute permission. Due to timing details, it is possible that several members of a sequence are unsuccessfully tried before the whole sequence is halted. (Recall that entries are not run if they belong to the same sequence or stream as too many entries in state SICK.)

It is best to execute `aurun` at or near interactive priority, as it uses few resources but must deal with the queuefile quickly. The UNIX priority of entries themselves is determined by their `nice` attribute.

The available switches for `aurun` are as follows.

`-logfile` *string*, `-nologfile`

This specifies a file to receive the master log for this `aurun`. The default is `~/autoson.@.run`; i.e., `autoson.`⟨*hostname*⟩`.run` in your home directory. The contents of this file are explained in Section 16. It is best to use a full path name for the argument. `-nologfile` is the same as `-logfile /dev/null`.

`-only` *unsigned*
: The argument is an entry number. Just run that entry and exit. If there is no such entry, exit as soon as abandoned entries are checked for. (See the description of `retries` in Section 2.)

`-silent`
: Turn off informational messages, such as messages about entries being found abandoned. These are written to `stderr` when `aurun` starts up.

`-maximum` *unsigned*, `-nomaximum`
: Specify how many entries can be run before `aurun` exits. The default is to have no limit, which is the effect of either `-nomaximum` or `-maximum 0`. Note that there is no connection between this switch and the `maximum` attribute of entries.

`-stream` *string*, `-nostream`
`-identification` *string*, `-noidentification`
: Restrict this `aurun` to entries with the given identification and/or stream. In both cases, the argument can be an empty string. An empty stream name means no stream. The default behaviour is that there is no restriction.

`-append`, `-noappend`
: Specify whether an existing master log is overwritten or appended to. The default is to append.

`-yawn`, `-noyawn`
: Specify yawn default for entries with `yawn`=*maybe*. See the description of the `yawn` attribute, and Section 12, for more details.

`-clear`
: If ever `aurun` finds that there is nothing in the queue which it is allowed to run, it will exit. However, it will not exit if it could run something but for it having `cautious`=*true*.

`-umask` *octal*, `-noumask`
: This defines the default *umask* value for entries which do not specify one. The argument is an octal number between 0 and 777. `-noumask` (the default) requests `aurun` to continue using the *umask* value in effect when it is started.

`-serialize`, `-noserialize`
: Set serialization on or off for those entries which have `serialize`=*maybe*. The default is `-serialize` in multiuser mode and `-noserialize` in single-user mode.

The `startaurun` script is a convenient way to start `aurun`. See Section 19 for details.

The actual process of running an entry differs in single-user and multiuser modes, though it is intended that the results be close.

In single-user mode, all of the environment (such as $PATH) and process privileges are inherited from the `aurun` process. The command arguments are presented to the subprocess as they appear in `actcommand`, except that the escape sequences `\~`, `\#`, `\=`, `\@`, `\^` and `\,` are replaced by their unescaped single characters.

In multiuser mode, a basic context is constructed for the user who owns the entry. The `uid` and primary `gid` of the user are adopted, as well as any supplementary groups

to which the user might belong. A simple environment is constructed:
USER = the username of the user
HOME = the home directory of the user
SHELL = the name of the login shell of the user
PATH = `/bin:/usr/bin:/usr/sbin`.

The process run in multiuser mode is the login shell of the user owning the entry, in the following manner:

$\langle$*login-shell*$\rangle$ `-c` $\langle$*command-string*$\rangle$

Here, $\langle$*command-string*$\rangle$ is a string made up thus:

$\langle$*initialization-string*$\rangle$ $\langle$*commandverb*$\rangle$ `"`$\langle arg_1 \rangle$`"` `"`$\langle arg_2 \rangle$`"` ...

The value of $\langle$*initialization-string*$\rangle$ is `exec` by default, but it can be changed by means of initialization records in the queuefile, as described below. In forming the command verb and arguments in $\langle$*command-string*$\rangle$, the escape sequences `\~`, `\#`, `\=`, `\^`, `\@`, and `\,` are replaced by their unescaped single characters.

The value of $\langle$*initialization-string*$\rangle$ can be changed on the basis of which machine is running the entry, and what $\langle$*login-shell*$\rangle$ is. The format of initialization records is thus: `I` *hostgroup shellname arg0 initstring*

hostgroup is the name of a host group or a hostname, optionally preceded by `~`. *shellname* is a login shell name, exactly as it appears in password files (usually a full path like "`/bin/csh`"). *shellname* can also be "`*`", which matches all shell names.

arg0 is a string to be used as the 0-th argument when the shell is invoked. Some shells take note of this; for example `/bin/csh` will do a full login for a user if invoked with `-csh` as the 0-th argument. If *arg0* is given as "`*`", the 0-th argument is the final component of the login shell. (For example, if $\langle$*login-shell*$\rangle$=`/bin/csh`, then the 0-th argument is "`csh`".) This is also the default behaviour if there is no matching initialization record.

Apart from one leading space, everything on the line from *arg0* to the end is *initstring*. It is recommended that the initialization string ends in "`exec`".

Sample initialization records:

```
I =sun /bin/csh csh source $HOME/.login ; exec
I =sun * * exec
```

The order of the initialization records matters: the first one which matches the current machine and the login shell name of the user applies, if any.

6. The `auzap` command.

The **auzap** command sends a UNIX signal to the processes running an entry, or to an **aurun** process.

If the job is running on another machine, an attempt will be made to contact that machine using a remote shell command (usually `rsh`). This will fail if `rsh` has been disabled or you don't have necessary permissions (`.rhosts` entry or similar).

aurun establishes a process group for each CURR entry. Unless your jobs do things with process group numbers, **auzap** will send the same signal to every process in that group.

The special jobs described in Section 15 can't be signalled, because they are not separate processes.

The switches available for **auzap** are as follows.

`-entry` *entry-list*
: This switch is compulsory. The syntax is like for **auadd**, except that each item in the list (entry number or range) must specify at least one entry in CURR state, and all other entries are ignored.

`-last`, `-nolast`
: Before sending any signals, modify the `last` attribute of the entries. For a remote host, this is only done if the attempt to reach that host is successful. See Section 5 for more information. Ignored if `-aurun` is present.

`-requeue`, `-norequeue`
: Before sending any signals, set the `requeuetime` attribute of the entries. For a remote host, this is only done if the attempt to reach that host is successful. See Section 5 for more information. Ignored if `-aurun` is present.

`-signal` *signal*
: Specify which UNIX signal to send. This can be one of the strings `KILL`, `STOP`, `CONT`, `QUIT`, `TERM`, `INT`, `HUP`, or one of those strings with the prefix `SIG` (e.g. `SIGSTOP`). It can also be a number. Be aware that different versions of UNIX use different numbers for the same signal, so is it safer to use the name if you want to send a signal to a process on another machine. The default is `-signal KILL`.

`-aurun`
: Send the signal to the **aurun** process instead of the processes running the entry. This is not much use. The preferred way to close down one **aurun** process and the entry it is running is `auzap -last`. **aurun** is designed to exit gracefully on receipt of `SIGINT` signals, so `-signal INT` is recommended if you need to kill **aurun** with a direct signal.

`-overlap` *unsigned*, `-nooverlap`
: In order to speed throughput, **auzap** will run multiple `rsh` processes at the same time. The normal limit is 5, but you can change it with this switch. A value of 0 or 1 (or `-nooverlap`) will force it to do one at a time. Too much overlap risks running into system limits on the number of processes or open files.

`-silent`
: Turn off informational messages.

`-user` *string*, `-mine`   (`aumod` only)
  Limit the set of entries specified by `-entry` to those owned by the specified user.

When `autoson` is compiled, there is an option of having two different remote shell commands available. For example, the "normal" one might be `ssh` and the "alternate" one might be `rsh`. In this case, the alternate command is used when the remote host is in the host group `@altrsh`. Note that host groups whose names start with "`=`" are not interpretted correctly in `@altrsh`.

Examples.

```
    auzap -ent 34
```
Send a SIGKILL signal to entry 34.

```
    auzap -ent 57 -last -requeue
```
Kill entry 57, and instruct the controlling `aurun` process to put it back into state PEND before dying.

7. The `aulook` command.

The `aulook` command displays the contents of the queuefile or part of it.

The normal use is to display some or all entries. There are two formats, called `peek` (brief default format) and `show` (complete details). The format for output from the `peek` version is user-definable, as described below.

The switches `-groups`, `-formats`, `-rewrites`, `-speed` and `-initializations` make `aulook` behave completely differently (with no entries displayed at all).

`-entry` *entry-list*
  Specify which entries to display. The default is to display all entries. The format of the argument is the same as for `auadd`.

`-waiting`, `-nowaiting`
`-pending`, `-nopending`
`-current`, `-nocurrent`
`-holding`, `-noholding`
`-lost`, `-nolost`
`-sick`, `-nosick`
  Specify states that entries must have, or must not have, before being displayed. By default there is no restriction on state.

`-peek`, `-show`
  Specify brief or complete display. The default is `-peek`.

`-user` *string*, `-mine`   (`aumod` only)
  Limit the set of entries specified by `-entry` to those owned by the specified user.

`-groups`
  Don't display any entries, but display the host group definitions. See Section 9 for a description of host groups. An asterisk indicates those host groups the current machine belongs to. Apart from the special host groups (those starting with `=`), host groups are defined by lines in the queuefile, as described in Section 14.

`-formats`
: Don't display any entries, but display the formats for `peek`. "PCWHLS" refer to entry states. For each state, the default format is displayed, then on the next line the user-defined format is displayed (if any). See below for the meaning.

`-rewrites`
: Don't display any entries, but display the rewriting rules given in the queuefile. See Section 11 for the meaning.

`-speed`
: Don't display any entries, but display the speed factor of the current machine, according to the `T` lines in the queuefile (Section 16).

`-initializations`
: Don't display any entries, but display the initialization records in the queuefile. See Section 5 for the meaning.

`-nolock`
: Display the queue without gaining exclusive access to the queuefile first. This can be useful, for example, if the lock file exists due to malfunction, or it you don't have write permission on the directory containing the queuefile. Failure to seek exclusive access might mean that you catch the queuefile in some intermediate state, so try again if it is claimed to be empty or invalid. This is not dangerous, as the queue is only read, not written.

The output layout for the `aulook` command is intended to be reasonably friendly already, but you can change it if you like, individually for each state. The details are given in Appendix A. In the default formats, the "state" of CURR entries is shown as STOP if `stopped`=1 (the entry has been stopped under `aurun` control) and ORPH if `stopped`=2 (the controlling `aurun` has vanished).

We will give some sample displays using the default layouts. Lines starting with `%` are commands typed by the user. This sequence was produced by user `bdm` on a computer `tyl` in the directory `/home/bdm/autotest`. The queuefile is initially empty.

```
% auadd -log xx#.out -cyc 2 -step 2 -lim 8 sleep 100
added new entry 33

% aulook
33/bdm PEND cyc=2[2]/8 nomax  sleep 100

% aulook -sh
33 PEND pri=10 nice=19 cyc=2[2]8(8) retr=0 app cau grace=10 max=0
submitted Tue Mar 12 17:42:18 1996; timestamp Tue Mar 12 17:42:18 1996
isn=e650021 dir=/home/bdm/autotest logfile=xx#.out id=
user=bdm stime=105 umask=77 nexthosts= imm=0 loadf=1.00
command = sleep 100

% aulook          (rm some time later)
33/bdm CURR[tyl      ,Mar 12 17:44:37] cyc=2[2]  cmd=sleep
34/bdm PEND cyc=4[2]/8 nomax  sleep 100
```

```
% aulook -ent 33 -sh
33 CURR[tyl:242,245] pri=10 nice=19 cyc=2[2]2(8) retr=0 app cau
                                   (first line continued) grace=10 max=0
submitted Tue Mar 12 17:42:18 1996; timestamp Tue Mar 12 17:44:37 1996
isn=e650021 dir=/home/bdm/autotest logfile=xx#.out id=
user=bdm stime=105 umask=77 nexthosts= imm=0 loadf=1.00
actdir=/home/bdm/autotest actlog=/home/bdm/autotest/xx2.out stop=0
command = sleep 100
actcommand = sleep 100
```

8. The `aulock` command.

This is a rarely used command for managing the queuefile. There are four basic functions, in order of priority:

`aulock -unlock`	unlock the queue
`aulock -nocommand`	lock the queue and wait for EOF or RETURN
`aulock -edit`	lock the queue and edit it
`aulock`	lock the queue and run a subprocess

The fourth function is performed if none of `-unlock`, `-edit` or `-nocommand` is present.

The locking is performed by creating a lock file whose name is obtained by appending "`.lock`" to the name of the queuefile. It is created using `link()`, which fails if the file already exists. All `autoson` utilities use this mechanism to serialize access to the queuefile, except for `aulook -nolock`. The lockfile contains the name of the machine and command that created it, and a reason code. This information is not used except for debugging purposes.

`-unlock`
`-older` *unsigned*

If the queue is not locked, exit silently. If it is locked and `-older` is absent, report how long it has been locked for and ask permission to unlock it. If `-older` is present and the queue has been locked for at least that number of seconds, unlock the queue with an informative message (without asking).
`-unlock` is intended only for those abnormal times when the queue is locked but shouldn't be. This can happen due to system glitches, or if you kill an autoson utility at the wrong instant. `-unlock` takes precedent over all other switches.

`-command` *string*

If `-edit` is present, this should be the name of a program which can take the name of the queuefile as an argument. Otherwise, it should be a program that can be executed as is. The argument might be compound, for example `-command "vi -r"` is fine but the quotes are needed. If a program name is needed (i.e. neither `-unlock` nor `-nocommand` is present), but `-command` is absent, the name of the program will be:
With `-edit`: environment variable `EDITOR` if defined, else `vi`.
Without `-edit`: environment variable `SHELL` if defined, else your login shell.

`-nocommand`

After locking the queue, wait until EOF or RETURN is received from the standard input, then unlock the queue and exit. `-edit` is ignored.

`-edit`

`aulock` will lock the queue and run a subprocess "⟨*command*⟩ ⟨*queuefile*⟩", where ⟨*command*⟩ is determined as described under `-command`. If this switch is absent, the subprocess to run is just "⟨*command*⟩", which is typically a shell. In all cases, the queue is unlocked when the subprocess exits.

Examples:

```
    aulock -edit
```

Lock the queue and open the queuefile for editing with `vi` (or another editor named by the environment variable `EDITOR`). When finished, unlock the queue.

```
    aulock -edit -command less
```

Lock the queue and display the queuefile using `less`. When finished, unlock the queue.

9. Host groups.

A *host group* is a set of machines. The most important use is with the `hosts` attribute of entries, to specify which machines are permitted to run them.

There are three types of host group:

(a) A single machine, identified by its name, is a host group.

(b) Some sets of machines, defined by hardware and operating system, are predefined "special" host groups. They are identified by a name beginning with `=`.

`=sun`	Sun computer
`=sun3`	Sun3 computer
`=sun4`	Sun4 computer
`=sunos`	Solaris 1 operating system (such as 4.1.3)
`=solaris`	Solaris 2 operating system (such as 5.2)
`=sgi`	Silicon Graphics mips computer running IRIX
`=alpha`	Alpha computer
`=dec`	Digital ULTRIX workstation
`=hpux`	Hewlett Packard HP-UX computer
`=linux`	Linux operating system
`=all`	Every computer

(c) The user can define a host group in terms of other host groups, by inserting definitions in the queuefile. These groups are identified by a name beginning with `@`. For example:

`H @mine sun1 sun2 sun3 sun4`	(one of sun1..sun4)
`H @group1 ~=alpha sun2 sun3`	(sun2 or sun3, but not an Alpha)
`H @digital =alpha =dec ~@mine`	(an Alpha or a Dec, but not sun1..sun4)
`H @notsgi ~=sgi`	(anything not an SGI)

The character `~` indicates complementation. The order of the list is insignificant. Suppose the definition of `@G` is like this:

`H @G` $P_1\ P_2\ \ldots\ P_p\ \tilde{}N_1\ \tilde{}N_2\ \ldots\ \tilde{}N_n$

Suppose h is an actual machine, and define the propositions $\mathcal{P} = (h \in P_1 \cup \cdots \cup P_p)$ and $\mathcal{N} = (h \notin N_1 \cup \cdots \cup N_n)$. Then $h \in$ `@G` if and only if $\mathcal{N} \wedge (\mathcal{P} \vee (p = 0 \wedge n > 0))$.

Undefined host groups are treated as empty. The value of the switch `-hosts` is interpretted in the same way as a host group definition, except that an empty definition (or the absence of the switch) indicates that every machine can run the entry.

There is no command to add host groups. You need to use an editor on the queuefile (see `aulock -edit`).

Three user-defined host groups have special meaning. The group `@noyawn` is used to disable use of yawn on specified machines. This host group can also contain items of the form `user:`⟨*username*⟩. See Section 12 for an explanation.

The group `@disabled` also has special meaning. If `aurun` sees the name of its own machine in there, it will immediately exit. However, it doesn't look while it has an entry running.

Finally, `@altrsh` is used to select an alternate remote shell for `auzap`; see Section 6.

10. Macro characters and freezing.

Some characters have special meanings when present in the `directory`, `logfile` or `command` attributes of entries. These "macro characters" and their meanings are as follows:

- `^` : one character indicating the system type (sun3=`3`, sun4=`4`, SGI=`s`, Alpha=`a`, Dec=`d`, HP=`h`, Linux=`l`)
- `^^` : a string indicating the system, the first of these that applies: `alpha dec sgi solaris sun4 sun3 hpux linux sun`
- `%` : The value of `entrynumber`.
- `#` : The value of `cycle`.
- `=` : The value of `oldlimit` (a number or `nolimit`).
- `@` : The value of `runninghost`.
- `@@` : The value of `runninghost` followed by the last character of the name of the `aurun` command. For example, if `aurun0` is running on host `tyl`, then `@@` is `tyl0`.
- `~` : The name of your home directory, provided it is at the start of a string and is followed by the end of the string or `/`. (`autoson` does not know about the `~user` construct, but see the examples in Section 11 for a way to handle it.)

In the case of numbers (`%`, `#`, non-infinite `=`), a minimum field width can be specified by repeating the macro character. Leading zeros are used for padding. For example, with `cycle`=37, the string "`# ## ### ####`" represents "`37 37 037 0037`".

As described in Section 2, entries in state CURR, LOST or SICK have "actual" attributes `actdirectory`, `actlogfile` and `actcommand`, in addition to the "generic" attributes `directory`, `logfile` and `command` that all entries have. We will now describe the relationship between these.

When an entry is initially created, the specified values are stored in the generic attributes (there are no others, in fact, as entries cannot be created in CURR, LOST or SICK state).

When an entry is chosen for running by `aurun`, each actual attribute is set equal to the result of applying path rewriting (see Section 11) and macro substitution to the corresponding generic attribute. Then, those attributes specified for freezing by the `freeze` attribute are frozen: this means that the generic value is set equal to the actual value, permanently losing the old generic value. The actual values are used to execute the entry, and if execution completes satisfactorily the generic value plays no further role. However, if the entry is discovered abandoned and moved to state LOST, both the generic and actual values are preserved in the LOST entry.

In making any modification to an entry, as by using `aumod`, the presence of any `-freeze`/`-nofreeze` switch overrides the entry `freeze` attribute. If the initial state of the entry is CURR, LOST or SICK, the attributes specified by the (possibly overridden) `freeze` attribute are frozen first, before other modifications are done. Any use of `-logfile`, `-directory`, or a command on the `aumod` command changes only the generic attribute. If the state is changed from CURR, LOST or SICK to PEND, WAIT or HOLD, the generic parameter is the one preserved.

If a running entry is discovered abandoned but put straight into PEND due to having `retries`>0, the result should be the same as if it was put into LOST then manually moved to PEND using `aumod -nolost`.

For example, suppose you have SGI and Dec machines and create an entry using `auadd -log ^^/test.log -freeze l ^^/cmd arg` .
If this is selected for running by an `aurun` process on an SGI machine, it is moved to CURR with attributes thus:

`logfile`=`sgi/test.log`, `actlogfile`=`sgi/test.log`

`command`=`^^/cmd arg`, `actcommand`=`sgi/cmd arg`.

Because the `freeze` attribute of the entry specified "`l`", `logfile` was set equal to `actlogfile`. If this entry later is found abandoned and put into state LOST, the attributes will still be like that. If it is then moved to state PEND, the generic attributes only are preserved: `logfile`=`sgi/test.log`, `command`=`^^/cmd arg`.
If a Dec machine runs it now, the CURR entry will appear thus:

`logfile`=`sgi/test.log`, `actlogfile`=`sgi/test.log`

`command`=`^^/cmd arg`, `actcommand`=`dec/cmd arg`.

Thus, the same logfile will be used even though another executable is chosen. If no `-freeze` parameter had been given on the initial `auadd` command, the `logfile` attribute would have remained `^^/test.log`, so the second execution would have started a new logfile called `dec/test.log`.

The expansion of macro characters can be disabled by escaping them with `\`. For example, "`\#`" is a normal, non-macro, `#`. Note that `\` is special to the shell, so when you type it in you need to quote it.

11. Path rewriting.

This facility is provided to ease the difficulty of working in a network where file names vary from one machine to another, or where multiple links to directories can cause paths to be legal on one machine but not on another. By means of lines in the queuefile (see Section 14) the `directory`, and `logfile` and `commandverb` attributes can be systematically editted (but not the command arguments).

The general form of a rewriting rule is like this:

`R` *when attr hostgroup inpattern outpattern*

when is an integer, 0 to indicate rewriting at entry creation, 1 to indicate rewriting at entry running, or 2 to indicate both. *attr* is either `all` or `none` or a nonempty string over {`c`, `d`, `l`}. `all` is equivalent to `cld`. These characters indicate a subset of {`commandverb`, `directory`, `logfile`}. *hostgroup* is the name of a host group or its complement. *inpattern* and *outpattern* are arbitrary non-empty strings, or the word `NULL` to indicate the empty string.

inpattern is a pattern consisting of ordinary characters plus possible wild cards. The meanings of the wild cards are:
`?` : match any single character except '`/`'
`**` : match zero or more characters (as little as possible)
`*` : (except `**`) match zero or more characters other than '`/`' (as much as possible).

In addition, *inpattern* may end with `$`, which matches zero characters at the end of the input string. (Any `$` not at the right end of *inpattern* is just an ordinary character.)

outpattern is a string consisting of ordinary characters plus possibly the escape sequences `\0`, `\1`, ... These escape sequences represent the substrings matched by the wild cards in *inpattern.*

In addition to the string represented by *outpattern*, the part of the input string not matched by *inpattern* (necessarily a suffix) is copied unchanged to the output. Note that matching always starts at the first character of the input string.

We continue with some examples:

inpattern: `/tmp_mnt/`
outpattern: `/`
input: `/tmp_mnt/home/bdm`
output: `/home/bdm`
input: `/etc/tmp_mnt/xyz`
output: match fails (must start at first character)

inpattern: `**/MACH`
outpattern: `\0/SUN`
input: `/home/bdm/MACH`
output: `/home/bdm/SUN`
input: `/home/bdm/MACH/etc`
output: `/home/bdm/SUN/etc`

inpattern: `**/MACH$`
outpattern: `\0/SUN`

input: `/home/bdm/MACH`
output: `/home/bdm/SUN`
input: `/home/bdm/MACH/etc`
output: match fails (the `$` only matches the end of the input)

inpattern: `/home/*/SUN?/*$`
outpattern: `/fac/\0/SUN\1`
input: `/home/bdm/SUN3/tmp`
output: `/fac/bdm/SUN3`
input: `/home/SUN2/SUN3/SUN4`
output: `/fac/SUN2/SUN3`
input: `/home/bdm/etc/SUN3`
output: match mails (`*` can't match a string with '`/`')

inpattern: `~?*`
outpattern: `/home/\0\1`
input: `~/etc`
output: match fails (`?` won't match '`/`')
input: `~freddy/dir/file`
output: `/home/freddy/dir/file`

Consider for example a hypothetical network of Suns and SGIs, with the feature that directories which appear as `/home/...` on Suns appear as `/fac/home/...` on SGIs. Moreover the string `/tmp_mnt` sometimes appears on the name of the current directory even though not all machines in the network understand it. We can handle this situation with these rewriting rules:

```
R 0 =all  all  /tmp_mnt/  /
R 1 =sun  all  /fac/home/ /home/
R 1 =sgi  all  /home/     /fac/home/
```

The flag *when*=0 indicates a rewriting rule to be applied immediately when an entry is created or when the `logfile`, `directory` or `command` attributes are changed using `aumod`. So, the first rule says that the prefix "`/tmp_mnt/`" is to be replaced by "`/`" at that time. The rule could also be written

```
R 0 =all  all  /tmp_mnt   NULL
```

but then it would incorrectly edit "`/tmp_mnt1/xx`" into "`1/xx`". Rules with *when*=0 affect the generic attributes only.

The flag *when*=1 indicates a rewriting rule to be applied when an entry is selected for running. This is done at the same time as the `actdirectory`, `actlogfile` and `actcommand` attributes are created, but before expansion of macro characters (see Section 10). The actual order is like this:

(1) Apply legal rewriting rules to a copy of the value of `directory`.
(2) If the resulting string does not now start with `/` or `~`, prepend the name of the home directory.
(3) Expand macro characters and assign the result to `actdirectory`.
(4) Apply legal rewriting rules to a copy of the value of `logfile`.

(5) If the resulting string does not now start with `/` or `~`, prepend the value of `actdirectory`.
(6) Expand macro characters and assign the result to `actlogfile`.
(7) Apply legal rewriting rules to a copy of the value of `commandverb`.
(8) Expand macro characters in the resulting string, and in the value of `arguments`. Assign the results to `actcommand`.
(9) Perform any required freezing operation (see Section 10).

Note that the values of `directory`, `logfile` and `command` are unchanged except perhaps in step (9). In our example, if the machine that runs the entry is a Sun, paths `/fac/home/...` are rewritten as `/home/...`, and if it is an SGI, `/home/...` is rewritten as `/fac/home/...` .

The item *hostgroup* is any host group, user-defined or special, or the complement of one (for example, `~=alpha`). As always, undefined host groups are implicitly empty. In all cases, the rewriting rules are executed repeatedly in circular order until none apply, with the limit that no rule is executed more than once.

The rewriting feature provides a nice way to change directories automatically between different types of machines. Suppose you run a mixture of SUN and SGI machines, as before. Suppose your executables reside in subdirectories SUN and SGI. Define these rules:

```
R 1 =sun  all  WORK/     SUN/
R 1 =sgi  all  WORK/     SGI/
```

Then `auadd WORK/command arg1 arg2` will find the right executable. However, `auadd etc/WORK/command arg1 arg2` will not work because the pattern matching always starts at the beginning of the string. That more general case is handled by rules like these:

```
R 1 =sun  all  */WORK/   \0/SUN/
R 1 =sgi  all  */WORK/   \0/SGI/
```

and then `auadd etc/WORK/command arg1 arg2` will work. Many variations on this are possible.

12. The yawn protocol.

If the yawn protocol is selected as described above (see the attribute `yawn` in Section 2), the machine is monitored for activity to reduce the impact of your processes on other users.

According to rules you provide in a “yawn times file”, each moment in a week is classified into five “time classes”. These classes may overlap. Their names and meanings are as follows.

ALWAYS: jobs are allowed to run

NEVER: jobs are not allowed to run

MAYBE: jobs are not allowed to run if there has been less than 10 minutes since the last use of the the keyboard or mouse, or of a terminal device other than `/dev/console` belonging to a logged-in user. The `utmp` or `utmpx` file is used to find these terminal devices, so sessions not recorded there will be ignored.

CONSOLE: jobs are not allowed to run if it has been less than 10 minutes since the last use of the the keyboard or mouse, or if someone is logged into the console (regardless of activity).

LOAD: jobs are stopped if the system load average is above a given upper limit, and allowed to run again if the load average drops below a given lower limit.

Class NEVER overrides any of the other classes. Similarly, any class overrides class ALWAYS. Otherwise, any particular time of day can be in any combination of classes at once, and jobs are only allowed to run if all the requirements of each class are met.

The periods belonging to each time class are specified in a “yawn times file”, which will be the first of these four that exists and is readable:

`$HOME/yawn.`⟨*hostname*⟩
`YAWNDIR/yawn.`⟨*hostname*⟩
`$HOME/yawn.default`
`YAWNDIR/yawn.default`

In these names, “YAWNDIR” is the name of a directory that can be specified when `autoson` is compiled. The two possible names involving $HOME are omitted in multiuser mode.

The yawn times file contains lines, of which these are examples:

```
weekday 0900 1700 never
weekend 0900 1800 load 0.6 1.1
all 0000 0900 always
```

The first field is one of `sun`, `mon`, ..., `sat`, `weekend`, `weekday`, and `all`. The next two fields are starting and finishing times as *hhmm* (from 0000 to 2359). The fourth field is one of `never`, `maybe`, `always`, `load` and `console`. In the case of `load`, two real numbers x and y also need to be given. These are used in conjunction with the loadfactor f of the current entry. If the system load is at most x when the entry is not running, it is allowed to run. If the load is more than $x + f * y$ when the entry is running, it is stopped.

The time class is monitored periodically, and SIGSTOP/SIGCONT signals are sent appropriately. While the job is running in class MAYBE, the devices are monitored every 30 seconds so on average a new interactive user will be seen within 15 seconds.

The yawn times file may be editted while a job is running, and will take effect before very long. Actually, what happens is that a change to the modification time of the current yawn times file will cause `aurun` to reread the yawn times files as described above. For example, if it is using `yawn.default` but you want it to use a new file `yawn.`⟨*hostname*⟩, just "touch" `yawn.default` when `yawn.`⟨*hostname*⟩ is ready.

Finally, there is a mechanism to tell yawn to ignore terminal lines in use by particular users. Include in the `@noyawn` host group an item like `user:`⟨*username*⟩. This will also cause console logins by ⟨*username*⟩ to be ignored, but will not stop yawn from looking at the keyboard and mouse.

13. Environment variables.

The environment variables `HOME` and `PWD` are no longer used.

A non-standard name for the queuefile can be specified in the environment variable `AUTOSON_QUEUE`. This is described in Section 3.

The only other environment variables which `autoson` might use are `EDITOR` and `SHELL`, as described in Section 8.

14. The format of the queuefile.

The queuefile is an ascii text file, divided into "records" whose type is indicated by the first character.

`N`	Next entry number record (1 line)
`S`	Global clock record (1 line)
`H`	Host group definition record (1 line)
`F`	Format definition record (1 line)
`R`	Rewriting rule record (1 line)
`I`	Initialization record (1 line)
`E`	Entry record (6 or 8 lines)
`T`	Machine speed record (1 line)

These are described one at a time below.

Next entry number record:
`N` *unsigned*

When the next new entry is created, this is the `entrynumber` it will have.

Global clock record:
`S` *unsigned*

This is the counter used by by the `schedtime` attributes of entries. See Section 2 for an explanation.

Format definition record:
`F` *char string*

The character is one of `C`, `P`, `W`, `H`, `L` and `S`. The string is a layout for the state indicated by the character. The details are given in Appendix A.

Host group definition record:
`H` *string hostgroup*

The string is the name of a host group being defined, and must begin with the character `@`. The host group is a space-separated list of host groups. Any names beginning with `@` or `=` must have been previously defined. The semantics are given in detail in Section 9.

Rewriting rule record:
`R` *unsigned string string string string*

This gives a rule for rewriting file names. A full description appears in Section 11.

Initialization record:
`I` *string string string string*

This defines an initialization string used when an entry is run in multiuser mode. A full description appears in Section 5.

Machine speed record:
`T` *string realnumber*

This defines a relative speed factor for a class of machines. The *string* is a host group or the complement of a host group. The *realnumber* is any floating point number. Each machine has relative speed factor equal to the number given on the first `T` line which matches it, or 1 if there are none that match. It doesn't need to correspond to reality. Currently the only use of the relative speed factor is to compute the scaled cpu time displayed in logfiles (see Section 16).

Entry record:

This type of record has 7 lines (PEND, WAIT, HOLD) or 9 lines (CURR, LOST, SICK). It contains the values of each attribute of one entry. We will give the contents line by line. Boolean values are given as 0 or 1. Times are given as integers, as for the UNIX system call `time()`. The value ∞ appears as $2^{31} - 1$. Some strings are written with the character `X` in front, so that null strings are visible.

(1) `E entrynumber isn state priority nice submittime timestamp cycle limit oldlimit grace tries maximum freeze append cautious`
States are encoded thus: 1=CURR, 2=PEND, 3=WAIT, 4=HOLD, 5=LOST, 6=SICK. `isn` is an internal attribute used to uniquely identify sequences. `freeze` is a non-null string.

(2) `Xstream directory logfile last resubdelay yawn verbose once sickness`
The three values of `yawn` are encoded as $-1, 0, 1$.

(3) `Xusername Xnexthosts X schedtime serialize immunity umask loadfactor step−1 requeutime 0`
The three values of `yawn` are encoded as $-1, 0, 1$. The value of `umask` is written in decimal, with -1 meaning "none". The isolated `X` and `0` are placeholders for future use.

(4) `runninghost runningpid aurunpid stopped actdirectory actlogfile`
This line only appears if the state is CURR, LOST or SICK.

(5) `hosts`
This is a blank separated list of strings.

(6) `prerequisites`
This is a comma-separated list of items, each *integer* or *integer:integer*.

(7) `identification`
All of this line is the value, empty or not.

(8) `command`
This is a list of one or more strings, comma separated.

(9) `actcommand`
This is like line (7), but only appears for state CURR, LOST or SICK.

15. Special jobs.

There are a small number of "special jobs" known to `aurun`, identified by their `commandverb` attribute. The selection of special jobs to run is carried out exactly the same as for ordinary jobs, but instead of running a program provided by the user, `aurun` performs some action itself.

`_reset`
This job causes `aurun` to restart. This involves closing and reopening the `aurun` logfile, and looking for devices used by yawn (Section 12). Arguments `noappend` and `append` override any `aurun` switches, so `_reset noappend` is a way to truncate a `aurun` logfile that has become too long.

`_getlost`
This job causes `aurun` to look for abandoned entries on the current machine, and for entries for which the controlling `aurun` appears to have vanished. Normally this is only done when `aurun` starts up. Messages about any that are found will appear in the `aurun` logfile.

Example:

```
    auadd -once -nolim -nolog _reset noappend
```
Truncate all `aurun` logfiles on all machines. Don't forget to delete it when it has run on every machine.

16. Output formats.

We now give an example of logfiles. Suppose you start `aurun0` with a fresh logfile and execute the command `who`. After it finishes, the logfile of the entry (as given by the `-logfile` switch in the `auadd` command) might look like this:

```
Autoson entry 57, started at Mon Mar 18 19:17:53 1996 on tyl (solaris)
-----------
bdm        console      Feb 26 09:37
bdm        pts/2        Feb 26 09:37
bdm        pts/3        Feb 26 09:37
bdm        pts/5        Feb 26 09:37
bdm        pts/4        Feb 26 09:37
-----------
Entry 57 terminated on tyl at Mon Mar 18 19:17:54 1996 (status 0)
cpu=0.02u,0.09s,0.18x  pf=118
--------------------------------------------------------
```

The actual command output is visible in the center. Most of the rest should be self-evident if you know that `tyl` is the name of the machine that executed the entry. On the first line, `solaris` is the value of the macro `^^`. On the last line, the cpu times are in seconds. There are user-time, system-time and the sum of these two multiplied by the relative speed factor for this machine (1.6), `pf=118` means there were 118 page-faults, and `(status 0)` means that the process terminated using `exit(0)` or equivalent; it would say `(signal 9)` if it was killed using signal number 9.

The contents of the master logfile for the `aurun` process which executed this entry might look something like this:

```
Autoson Version 1.3; Mar 17, 1996 (single-user)
mouse=/dev/mouse keyboard=/dev/kbd console=/dev/console
@@=tyl0 ^=4 ^^=solaris
-------
YAWN: using times file /home/bdm/yawn.default
Starting entry 57 on tyl at Mon Mar 18 19:17:53 1996
who
user=bdm dir=/home/bdm/autoson log=/home/bdm/autoson/who.out
-----------
Entry 57 terminated on tyl at Mon Mar 18 19:17:54 1996 (status 0)
cpu=0.02u,0.09s,0.18x  pf=118
--------------------------------------------------------
```

At the beginning there is some information about yawn devices and macro values. Then there is some information about the entry, much of it a copy of things that appear in the entry logfile but also the values of `actcommand`, `username`, `actdirectory` and `actlogfile`. The only other thing in this example is a message from yawn stating which times file it is using (not repeated unless it changes). In a longer running case, there might be messages from yawn telling us what it is doing, for example like this:

```
YAWN: stopped at Tue Feb 20 18:08:45 1996, reason=/dev/kbd
YAWN: continued at Tue Feb 20 18:50:46 1996, class=maybe
YAWN: stopped at Wed Feb 21 09:00:25 1996, reason=load
YAWN: continued at Wed Feb 21 17:27:25 1996, class=maybe&load
```

At 18:08:45, yawn stopped the job because the keyboard device was not idle. At 18:50:46 all relevant devices were idle, so it started the job again and it ran until 9:00:25 the following morning, when the prescribed system load was exceeded. Finally, at 17:27:25, the job was restarted again.

17. Special considerations for multiuser mode.

This section will summarize the special features of `autoson` when run in multiuser mode.

The set of all possible users is divided into four classes.

(a) *Supervisors* are specified in a list compiled into the code. These users have access to all the features of `autoson`.

(b) *Entry owners* are the users whose username matches the `username` attribute of some entry. This class is only defined in the context of an entry.

(c) *Authorized users* are users whose username appears in the authorization file. The name of the file is the same as the queuefile, plus a further extension `.users`. The syntax of that file is free: usernames separated by blanks, tabs or newlines.

(d) *Other users* are everyone else.

These classes are not necessarily disjoint.

For users in the last class, the only available `autoson` facility is the use of `aulook`.

Only authorized users and supervisors can create entries. Only entry owners and supervisors can modify existing entries or use `auzap` in regard to an entry.

Only supervisors can perform the following operations:

(a) use `auzap -aurun` to signal an `aurun` process;

(b) set the `last` attribute of entries;

(c) use the `aulock` command or the `aurun` command;

(d) use `-cautious`, `-really`, `-renumber`, `-serialize`, `-yawn`, or their negations.

(e) run special jobs (Section 15).

In addition, users other than supervisors are restricted to bounds on the values of the `priority`, `nice` and `loadfactor` attributes, normally the same as the default values.

18. System notes.

Sun:

You need to compile `autoson` separately on Solaris 1 (for example SunOS 4.1.3) and Solaris 2 (for example 5.3) because the format of the `utmp` file is different and signal numbers are different.

On Solaris 1, the cpu time written in the trailer is sometimes wrong for unknown reasons. On Solaris 2, the page fault count always appears as 0.

HP-UX:

Both the cpu times and the page fault count appear as 0, because there seems to be no way to get them.

Alpha, Dec:

I don't know how to test for keyboard activity. I will add it if someone tells me how and it isn't too complicated.

SGI:

The mouse and keyboard devices are insufficient for monitoring their activity. There is some code to do this using `/dev/qcntl0`, but it only works on fewer versions of the operating system and then only with root privilege. It is recommended that single-user mode `autoson` NOT be installed with root privilege, for security reasons. Use multi-user mode if it is essential that the mouse and keyboard be monitored.

19. The `startaurun` script.

There are two shell scripts, `startaurun_singleuser` and `startaurun_multiuser`, which differ only in the parameters they use on the `ps` command. We will describe them together as "`startaurun`".

The normal operation of `startaurun` is to start `aurun0` on the current machine unless it is already running. It assumes the default name for the logfile, and copies the last 200 lines of any previous logfile to a file of the same name except it has suffix "`oldrun`" instead of "`run`".

If you wish to run more than one `aurun` on some machines, create a file called `startaurun.cmds` in your home directory. Then, for each such machine, insert a line containing the name of the machine and one single character for each `aurun` you wish to run. For example, to run `aurun0` and `aurun1` on machine `fred`, `startaurun.cmds` should contain a line "`fred 0 1`". There is no need to include machines for which you want `aurun0` only.

`startaurun` can be run manually or via `crontab`.

20. The `splat` program.

The program **`splat`** is completely independent of **`autoson`**, but is included here due to its usefulness in managing jobs in a network environment. It can be used to execute a command on each machine in a network, or some subset of those machines, using multiple overlapping **`rsh`** processes for faster throughput.

Usage:

```
splat [-s] [-n] [-#] [-t timeout] [-f file] [-d] command...
```

This will execute the command `command...` on each machine listed in the file `file`, or some subset of those indicated by the `-#` switch (# is a number).

Explanations of the switches:

`-s` : suppresses the "`hostname:`" header for each host output

`-n` : omit the current host

`-#` : specify a host mask (see below). # is an unsigned integer.

`-f file` : specify the host file (containing machine names, see below). The default is `$HOME/.splathosts`. "`-`" means `stdin`.

`-t timeout` : specify a maximum number of seconds to wait The default is 34, but you need to increase that if the network is very slow or if you are running a command that takes a long time. There is a granularity of about 5 seconds.

`-d` : write some debugging information

`command..` is a command to execute. It can consist of any number of arguments, with escaping or quoting to avoid meddling by the shell. For example

```
splat 'ps | grep xyz'
splat ps \| grep xyz
```

have the same effect.

The argument to `-f` or `-t` can be either appended to the switch name or be a separate parameter.

The output from each host is not necessarily in the same order as the hosts appear in the hosts file, but the outputs are not intermingled.

The hosts file should contain a list of host names with white space separating, any number per line. An optional mask can be associated with each host, for example `hawaii:2` gives the mask value 2, an unsigned decimal integer. The default mask value is 1. Any name starting with the character `#`, and the remainder of that line, is ignored.

The switch `-#` (where # is an unsigned decimal integer) can be used to select hosts according to their masks.

(a) If #=0, all hosts are selected.

(b) If #>0, those hosts whose mask has bitwise AND with # are selected.

(c) The default value of # is `~0`, which selects all hosts with nonzero masks.

21. Getting started.

In this section we describe the initial installation and testing of `autoson`. Version 1.3 of `autoson` is distributed as a compressed tar file `autoson13.tar.Z`. It an be unpacked using one of these commands:

```
zcat autoson13.tar.Z | tar xvf -
uncompress autoson13.tar.Z ; tar xvf autoson13.tar.
```

The installation procedure is significantly different for single-user and multiuser mode, so we will describe it separately for each.

Single-user mode installation

(a) Edit the file `audefaults.h` to select single-user mode and change any defaults you wish. It is a good idea to read the entire file carefully, except the multiuser mode section.

(b) Edit the first few lines of `makefile` if necessary, according to the instructions that appear there.

(c) Type `make all`.
Unless some error occurs, this will make executable files `autoson0` and `autoson1`, and links to these called `aulook`, `auadd`, `aumod`, `auzap`, `aulock`, `aurun0` and `aurun1`. It will also make `splat`.

(d) Move the just-mentioned seven links to your `bin` directory (or add the current directory to your path). Also move `splat` and `startaurun_singleuser` there.

(e) (Omit this step for Linux and Alpha.) If you wish to use the LOAD class of the yawn protocol, it is necessary that `autoson0` can read the special file `/dev/kmem`. Get a super-user to change the group owner of `autoson0` to the same as the group owner of `/dev/kmem`, which is usually `sys` or `kmem`. Then change the mode of `autoson0` to `2750`. If you omit this step, the only thing that won't work is the LOAD class of yawn.

(f) Create an empty file `autoson.queue` in your home directory. Copy the file `yawn.default` to your home directory, and edit it if necessary.

That completes the essential installation. If you type `aulook` now, it should respond "`The queue is empty.`". Now try adding a short sequence:

```
auadd -noyawn -lim 9 -log echo.#.out echo This is case #.
```

You will see it there if you use `aulook` again, but it won't run yet because you haven't started `aurun`. This one entry represents a sequence of nine jobs, with `cycle` taking the values $1, \ldots, 9$. So, start `aurun0` using the command `startaurun_singleuser`.

The command `echo` is of course very quick, so before you know it, the queue will be empty again. Then you should find (in the same directory as where you were when you executed the `auadd` command) output files `echo.1.out`, ..., `echo.9.out` containing the output of the nine jobs.

If that was successful, all that remains is to start `aurun0` on your other machines as well. Of course, you need to compile separate executables for each system type (including separate executables for Solaris 1 and Solaris 2 if you have both). You need to make sure your login files (`.login`, `.cshrc`, etc.) set PATH to select the right ones.

In some environments, you will need to add some rewriting rules before everything works properly on a mixture of machine types. See Section 11 for instructions.

Multiuser mode installation

In multiuser mode, `aurun` and `auzap` require root privilege, and care must be taken to avoid security holes. No known security problems exist with `autoson` if correctly installed, but please notify the author immediately if you find one.

You need to create a new user account. I will call it `autouser` and suppose for purpose of explanation that the home directory of `autouser` is called `/home/autouser`.

(a) Log in as `autouser`.

(b) Set the mode of `/home/autouser` to `755` and make a subdirectory `private` of mode `700`. Go to `private` and unpack the file `autoson13.tar.Z` there.

(c) Edit the file `audefaults.h` to select multiuser mode and set all defaults that you wish. The defaults in there should be ok already, except that you should change "`autouser`" and "`/home/autouser`" to what the real names are.

(d) Edit the first few lines of `makefile` if necessary, according to the instructions that appear there.

(e) Type `make all`.
Unless some error occurs, this will make executable files `autoson0` and `autoson1`, and links to these called `aulook`, `auadd`, `aumod`, `auzap`, `aulock`, `aurun0` and `aurun1`. Set the ownership and modes of these files as follows:
  `autoson0`: owner=`root`, mode=`4755`.
  `autoson1`: owner=`autouser`, mode=`4755`.
Move all nine executable files to some place your users can see them.

(f) Repeat steps (d) and (e) for other system types, if you are running more than one. Note that separate executables are needed for Solaris 1 and Solaris 2.

(g) Create an empty file `/home/autouser/autoson.queue`, owner=`autouser`, mode=`644`.

(h) Copy `yawn.default` to `/home/autouser`, mode=`644`. Edit it as you like. You can also make separate yawn times files for other machines; see Section 12.

(i) If necessary on your system, create a `.rhosts` file that enables `autouser` to use `rsh` from any machine to any other. (If this is not possible, for example if you have disabled `rsh`, `auzap` will only work on local entries. Everything else will work.)

(j) Create a file `/home/autouser/autoson.queue.users`, mode `644`, containing a list of authorised users. The format is free: usernames separated by white space.

(k) Try running a test job, as described for single-user mode. Preferably try it for several authorized users. Use `startaurun_multiuser` to start `aurun0`. Note that users other than supervisors are not allowed to use `-noyawn`.

(l) Start `aurun0` on all desired machines.

Appendix A. Formats for `aulook`.

The brief output format used by `aulook`, or in response to the `-peek` switch on some commands, can be configured freely. The default values are determined at compile time according to the file `audefaults.h`, and these can be overridden by means of lines in the queuefile. There is a different format used for each state. As the program is distributed, the defaults are:

(1) For single-user mode:

```
CURR: %e %Y[%@|%-9s|,%T]%#%+%=%v%y%z%S%m|| max=%d| max=%d|   %I
PEND: %e %s%#%+%=%p%v%y%z%h%P%f%S%m%1   %c %a
WAIT: %e %s[%T]%#%+%=%v%y%z%h%P%f%S%m%1   %i
HOLD: %e %s%#%+%=%p%v%y%z%h%P%f%S%m%1   %i
LOST: %e %Y[%@|%-9s|,%T]%#%+%=%v%y%z%S%m|| max=%d| max=%d|   %I
SICK: %e %s[%@,%T]%#%+%=%v%y%z%S%m%1   %I\n%k
```

(2) For multiuser mode:

```
CURR: %e/%u %Y[%@|%-9s|,%T]%#%+%=%v%y%z%S%m|| max=%d| max=%d|   %I
PEND: %e/%u %s%#%+%=%p%v%y%z%h%P%f%S%m%1   %c %a
WAIT: %e/%u %s[%T]%#%+%=%v%y%z%h%P%f%S%m%1   %i
HOLD: %e/%u %s%#%+%=%p%v%y%z%h%P%f%S%m%1   %i
LOST: %e/%u %Y[%@|%-9s|,%T]%#%+%=%v%y%z%S%m|| max=%d| max=%d|   %I
SICK: %e/%u %s[%@,%T]%#%+%=%v%y%z%S%m%1   %I\n%k
```

The items permitted in a format are listed below.

`\x`

For any character `x` this just writes `x`, except that `\t` and `\n` write tab and newline respectively.

`x`

Any character `x` except `%` or `\` represents itself.

`%x`

For some characters `x` this writes an entry attribute. A complete list of these characters is given below. In each case the output is defined by one or more formats, here called $F_1, F_2, \ldots$. Which format to use is determined by the rules given here. The formats are used directly in `fprintf` statements (see `fprintf(3)`). For each attribute there is a default set of formats, or you can specify your own in the layout by appending them immediately.

For example, the default for `%a` (command arguments) is F_1="" (if no arguments), F_2="`%s`" (for the first argument) and F_3="` %s`" (for later arguments). If the arguments are `arg1 arg2 arg3`, this will produce "`arg1 arg2 arg3`". If you want the output "`args = arg1,arg2,arg3`" instead, you can use
`%a||args = %s|,%s|`. You must give the same number of formats as in the default, separated by vertical lines.

item	attribute	default	meaning and values provided
`%a`	`arguments`	`||%s| %s|`	F_1 for no arguments *[none]*
			F_2 for first argument *[string]*
			F_3 for later arguments *[string]*
`%A`	same as `%a`, but using `actcommand` for state CURR, LOST or SICK		
`%c`	`commandverb`	`|%s|`	*[string]*
`%C`	same as `%c`, but using `actcommand` for state CURR, LOST or SICK		
`%d`	`directory`	`|dir=%s|`	*[string]*
`%D`	same as `%d`, but using `actdirectory` for state CURR, LOST or SICK		
`%e`	`entrynumber`	`|%d|`	*[int]*
`%f`	`freeze`	`|| fr=%s| fr=all|`	F_1 if no freezing *[string]*
			F_2 if intermediate *[string]*
			F_3 if *all* *[string]*
			The values are substrings of `cdl`.
`%F`	`loadfactor`	`|| lf=%3.2f|`	F_1 if default (1.0) *[double]*
			F_2 if not default *[double]*
`%g`	`grace`	`|| gr=%d|`	F_1 if default (10) *[int]*
			F_2 if not default *[int]*
`%h`	`hosts`	`|| host=%s|,%s|`	F_1 if null *[none]*
			F_2 for first value *[string]*
			F_3 for later values *[string]*
`%H`	`nexthosts`	`|| nh=%s|`	F_1 if null *[none]*
			F_2 if not null *[string]*
`%i`	`identification`	`|id=%s|cmd=%s|`	F_1 if non-null id *[string]*
	or `commandverb`		F_2 if null id *[string]*
`%I`	same as `%i`, but using `actcommand` for state CURR, LOST or SICK		
`%k`	`sickness`	`|sickness=%s|`	*[string]* (only SICK)
	The string is a description of the reason.		
`%K`	`immunity`	`|| imm=%d|`	F_1 if default (0) *[int]*
			F_2 if not default *[int]*
`%l`	`logfile`	`|log=%s|`	*[string]*
`%L`	same as `%l`, but using `actlogfile` for state CURR, LOST or SICK		
`%m`	`maximum`	`| nomax|| max=%d|`	F_1 if `maximum`=0 *[int]*
			F_2 if `maximum`=1 *[int]*
			F_3 if `maximum`>1 *[int]*
`%n`	`nice`	`|| nice=%d|`	F_1 if default (1) *[int]*
			F_2 otherwise *[int]*
`%o`	`oldlimit`	`||(%d)|(-)|`	F_1 if `oldlimit`$\leq$`cycle` *[int]*
			F_2 if `cycle`$<$`oldlimit`$<\infty$ *[int]*
			F_3 if `oldlimit`$=\infty$ *[int]*
`%p`	`priority`	`|| pri=%d|`	F_1 if default (10) *[int]*
			F_2 otherwise *[int]*
`%P`	`prerequisites`	`||:%d| pre=%d|,%d|`	F_1 if none *[none]*

			F_2 for hi end of ranges *[int]*
			F_3 for first prerequisite *[int]*
			F_4 for later prerequisites *[int]*
`%q`	`cautious`	`| nocau||`	F_1 if *false [none]*
			F_2 if *true [none]*
`%r`	`resubdelay`	`|| res=%d|`	F_1 if zero *[int]*
			F_2 if non-zero *[int]*
`%R`	`serialize`	`| noser|| ser|`	F_1 if *no [none]*
			F_2 if *maybe [none]*
			F_3 if *yes [none]*
`%s`	`state`	`|%4.4s|`	*[string]*
`%S`	`stream`	`|| str=%s|`	F_1 if null *[string]*
			F_2 if not null *[string]*
`%t`	`submittime`	`|%15.15s|`	*[string]*
			eg. `"Fri Oct 21 23:22:29 1994"`
`%T`	`timestamp`	`|%15.15s|`	*[string]*, as above
`%u`	`username`	`|?|%s|`	F_1 if unknown *[none]*
			F_2 if known *[string]*
`%U`	`umask`	`|| umsk=%03o|`	F_1 if none *[-1]*
			F_2 if given *[int]*
`%v`	`retries`	`| vol|| re=%d| rerun|`	F_1 if -1 *[int]*
			F_2 if 0 *[int]*
			F_3 if > 0 *[int]*
			F_4 if ∞ *[int]*
`%V`	`verbose`	`|| verb|`	F_1 if *false [none]*
			F_2 if *true [none]*
`%w`	`append`	`| noapp||`	F_1 if *false [none]*
			F_2 if *true [none]*
`%x`	`runningpid`	`|%d|`	*[int]* (only CURR, LOST or SICK)
`%X`	`aurunpid`	`|%d|`	*[int]* (only CURR, LOST or SICK)
`%y`	`yawn`	`| noyawn|| yawn|`	F_1 if *no [none]*
			F_2 if *maybe [none]*
			F_3 if *yes [none]*
`%Y`	`state+stopped`	`|STOP|%4.4s|ORPH|`	F_1 if `state`=CURR and `stopped`=1 *[string]*
			F_3 if `state`=CURR and `stopped`=2 *[string]*
			F_2 otherwise *[string]*
	The string is the state name.		
`%z`	`last`	`|| last|`	F_1 if *false [none]*
			F_2 if *true [none]*
`%1`	`once`	`|| once|`	F_1 if *false [none]*
			F_2 if *true [none]*
`%#`	`cycle`	`|| cyc=%d|`	F_1 if `cycle`=`limit`=1 *[int]*
			F_2 otherwise *[int]*

`%=`	`limit`	`||/%d|/-|`	F_1 if `limit`=`cycle` *[int]* F_2 if $\infty >$`limit`$\neq$`cycle` *[int]* F_3 if `limit`$=\infty$ *[none]*
`%+`	`step`	`||[%d]|`	F_1 if `step`=1 *[int]* F_2 otherwise *[int]*
`%@`	`runninghost`	`|%s|`	*[string]* (only CURR, LOST or SICK)

Acknowledgements.

Gunnar Brinkmann, Staszek Radziszowski and Stuart Ramsden were especially helpful in testing `autoson` and providing many valuable ideas for its development.
I also wish to acknowledge help from Clement Lam, Gordon Royle and Christoph von Stuckrad.

Restrictions.

The following copyright notice appears in `autoson.h`.