`dap_passwd:` A password to be used in conjunction with the `dap_user` variable in binding to the directory. It is not necessary to set this variable normally, unless the directory entries require an authenticated bind.

`queuestruct:` This is a variable used to control the structure of the queue. This is only useful to change if you are expecting very large queues ($>$ 2000 messages in the queue at one time.) It takes two numbers. The first number specifies the fan out of the queue. With a value of `100`, messages will be put in 100 sub-directories of the main queue file. This cuts down the searching of the main directory but is only useful for very large queues. The second optional number is the level of sub-directories to create. By default this is one, indicating an indirection of one directory. Again this parameter should not be changed unless extremely large queues are expected (say $>$ 50,000.)

`fsync:` This is a boolean variable which controls the use of the *fsync* (2) call. Fsync is normally used when critical files are written to before passing back a handshake. However, it can be switched off by a value of `no` if you consider that it is too expensive.

`diskuse:` This parameter is used to limit the disk space consumed by PP. There are two numbers, the first being the number of blocks that should be free, the second being the percentage of the disk space that should be free. Either of these can be effectively disabled by setting them to `0`. The disk that is checked is the partition holding the queue (`quedir`).

`adminstration_assigned_alternate_recipient:` This variable allows a undeliverable local address to be redirected to the user given here. This needs to be used with care, as messages that would normally be returned to the sender will be delivered to this address. Possible uses of this feature might be to deliver all failed messages to the postmaster for advice, or to deliver to a special program that returns fuzzy matches.

3.3 Logging Tailor File Entries.

This section describes how to tailor the logging produced by PP.

There are four main groups of logging; covering normal events, unusual operational events, authorisation, and ISODE. By default, each group will describe in a single file, the appropriate activities of all the programs that provide logs. However, the *tailor* file may be used to separate out logging on a per-program or per-channel basis (see Section 3.3.1 for more details). Thus, for example, all normal activities may be logged in one file except *submit* which may be logged in a different file, and at a different level.

All logging entries are comprised of a standard set of key/value pairs, described below.

`file=xxx`: Set the log file to `xxx`. This is either a fully qualified filename, or more usually just a relative pathname which is taken relative to the *logdir*. It defaults to one of `norm`, `stat` or `oper` depending on the log stream.

`level=value`: This adds to the logging levels the given value. Legal values are

`all`: Enable all logging (this fills disks very quickly!).

`fatal`: Enable reporting of fatal errors.

`exceptions`: Report exceptional happenings.

`notice`: Report interesting happenings.

`trace`: Trace the programs flow (very detailed).

`debug`: A full debugging trace.

`pdus`: Show protocol data units.

The last three levels are only available if PP was compiled with `PP_DEBUG` defined to a suitable level. The value `all` is special, in that it sets all values. The other levels only set the individual values (e.g., `debug` does not imply settings of any other values).

The default level is to enable `notice`, `exceptions` and `fatal`.

`dlevel=value`: This unsets a logging level. Legal values of this are the same as the above.

`sflags=value`: This sets the logging flags to a given value; the value being one of:

`close`: Close the log file after each entry is made.

`create`: Create the log file if it doesn't already exist.

`zero`: Zero the log file when it gets too big.

`tty`: Log to the standard error as well as to the file. This should be used only when debugging.

The default is to `create` new files.

`dflags`: This unsets one of the above levels for the log.

`size`: Set the maximum size in Kilobytes to which the log should be allowed to grow. The default is not to limit the log size. What happens when the logfile reaches the specified maximum is governed by the flags. If `zero` is not in force, logging will just stop. If `zero` is in force, then the log will be truncated and restarted.

There are currently four logging variables that can be specified although there are several logging structures.

The logging variables are:

`normlog`: This is the log where the bulk of the logging is done.

`operlog`: This is where urgent messages are logged. In the normal course of events no entries should be made in this log unless something serious has happened.

`authlog`: This indicates where authorisation tracing will be logged.

`isodelog`: This line allows the tailoring of specific ISODE tracing levels. The first argument is an ISODE logging level. The rest of the line is interpreted as a normal log tailoring line. Valid ISODE logging levels are:

`x25level`: X.25 level (if used).

`compatlevel`: Compatibility level.

`addrlevel`: Addressing level.

`tsaplevel`: Transport level.

`ssaplevel`: Session level.

`psaplevel`: Presentation structures.

`psap2level`: Presentation level.

`acsaplevel`: Association level.

`rtsaplevel`: Reliable transfer level.

`rosaplevel`: Remote operations level.

An example of logging tailoring is shown in Figure 3.1.

```
authlog          level=notice,      size=400
operlog          level=all,         size=400
normlog          level=notice,      size=400

submit           normlog file=submit     level=notice, size=400
qmgr             normlog file=qmgr       level=notice, size=400
x400in84         normlog file=x400in84   level=notice, size=400
x400out84        normlog file=x400out84  level=pdus,   size=400

isodelog compatlevel     level=pdus                              10
isodelog tsaplevel       level=pdus
isodelog ssaplevel       level=pdus
isodelog rtsaplevel      level=pdus
```

Figure 3.1: Example of Log Tailoring

3.3.1 Diverting Logging Streams

By default, all the ISODE logging is directed at the normal log.

Notice that for a given program or channel, the logging information can be diverted. If a logging entry starts with the name of the program (e.g., `argv[0]` normally) then that line is specific to that program; e.g.:

```
submit normlog file=submit
```

makes a *submit*-specific `normlog` tailoring entry. Hence all logging from *submit* directed at the `normlog` will go to the file *submit* instead of the file associated with `normlog`. The net result is that all the normal logging messages from *submit* end up in the file *submit* in the log directory.

3.4 Table Tailor File Entries

This section describes how the tables of aliasing and addressing information, referenced by *submit* and the channel programs, are defined in the *tailor* file.

Generally each table's entry is comprised of a line with several key/value pairs. The line starts with the keyword `tbl` which is followed by a string setting the default name for the table and the plain text file in *tbldir*. Then come the key/value pairs, described below.

`name=value:` Name this table with the given value (to override the default – see above). This is included only for completeness as it is set by first argument; it is not normally used. The name is used in all references to the table.

`file=value:` The tables contents are found in the given file (to override the default).

`show="value":` A descriptive string used when printing messages about this table, mainly for logging purposes. It defaults to the same as the table name.

`flags=value:` A set of flags that specifies how this table operates. It should be one of the following:

`dbm:` This table is stored in the database for fast access (the default).

`linear:` This table is searched with a linear pass through the file in *tbldir* (this is slow, but does not require rebuilding the database, and changes take immediate effect). Linearly searched files are still built into the database.

`override:` This allows values in the table to be overriden, or indeed if the table is small the whole table to be specified in the tailor file. Each occurance of this keyword adds a new key/value pair to a list. The format should be exactly the same as in a file with a key, a colon character (`:`) and a value.

An example of table tailoring is shown in Figure 3.2.

3.5 Channel Tailor File Entries

Channels are perhaps the most complex aspect of tailoring. There are several types of channels. Input channels carry messages into the system. Output channels carry messages out of the system. Reformatting channels change

```
tbl     aliases   show="Aliases:  mapping -> local id",
                  flags=dbm,file=local-aliases
tbl     domain    show="Mapping domain key ->full domain/MTA",
                  flags=dbm,override="xtel.co.uk:local"
tbl     or        show="Mapping O/R Address -> MTA",flags=dbm
tbl     or2rfc    show="RFC 987:  X.400 -> RFC 822",flags=dbm
tbl     rfc2or    show="RFC 987:  RFC 822 -> X.400",flags=dbm
tbl     channel   show="Binding MTA -> Channels",flags=dbm
tbl     auth.qmgr show="Qmgr authorisation",flags=linear
```

Figure 3.2: Example of Table Tailoring

the message structure in the queue. Finally there are a few miscellaneous channels that do other jobs such as message deletion.

The *channel tailor* file entries each consist of the keyword `chan` followed by a value which, by default, names the channel and the program associated with it; then comes a list of key/value pairs which provides more information on the channel.

A number of the values can be applied to a pair of channels differently depending on whether the channel is being used in outbound or inbound mode. Where the distinction is significant, there are separate tailor variables prefixed by `in` and `out`.

The key/value pairs consist of the following:

`name=value:` The name of the channel (overrides the default).

`prog=value:` The program associated with this channel, i.e., the actual binary to run (overrides the default).

`key=values:` A list of keys (comma separated) by which this channel is known. This can be used to map several logical channels onto one. See Section 3.7 on channel pairing on page 33. Note that a `key` is an equally valid way to refer to a channel. When referencing a channel, the first match on `key` or `name` is taken. If specifying several keys, the entier value should enclosed in quotes.

`show="value":` A descriptive string used in pretty printing. If this string starts with the keywords `with` or `via` then this value is included in `Received` lines generated for RFC 822 messages.

`type=value:` This parameter is *mandatory* and indicates the type of channel this is. This *must* be set to one of the values shown below:

Value	Type of Channel
`in`	An incoming channel.
`out`	An outgoing channel.
`both`	Both incoming and outgoing.
`shaper`	A reformatter that changes the shape of the message.
`warn`	A warning channel.
`delete`	A message deletion channel.
`qmgrload`	A queue manager loading channel.
`debris`	A debris collection channel.
`timeout`	A timeout channel.
`split`	A splitter channel.

`drchan=value:` This value must be set for outbound channels that cannot handle real delivery reports - that is channels of type `out` or type `both`.. At present, it is only the X.400 channels and the X.400-based delivery channels that can handle delivery reports. Other channels should set this to a suitable channel that can transform DRs into suitable messages. Currently this should always *dr2rfc*.

`content-in=value:` The content of this channel if it allows incoming messages. It is normally either `p2`, `p22` or `822`. It should be empty if the message is submitted in structured form (i.e. more than one file)

`content-out=value:` The content of this channel if it allows outgoing messages. Values are as for `content-in`.

`cost=value:` The cost of running this channel represented as a non-negative integer. This is used in the reformatting calculations to decide the best conversion path. The lowest cost route will be chosen, all other things being equal. Costs are relative; a cost of 0, the default, means that this channel essentially makes no changes to the message. Higher costs make the channel less favourable.

As a rule of thumb, any channel that makes a real change to a message should have a positive cost. Thus *rfc822norm* channels should have a small cost (e.g., 1) as they make small changes to the message. A channel such as *P2toRFC* should have a larger cost (e.g., 10) as this conversion should not be done unless needed. A channel that removes a content type by *removebp* should be given a large cost (e.g., 100) as this should only be run if there is no alternative. However, a channel such as *p2flatten* should have a zero cost as it makes no change to the message content, just combining it into one file.

`sort=value:` This value is a series of keys that are used for sorting purposes by the queue manager. The first key is the primary sort key and must be either `mta`, `user` or `none`. The value `none` should only be used as a primary sort key for delivery channels. The remaining keys are secondary sort keys and are taken in order. The values for this field are any of those shown below:

Value	Meaning
`mta`	Sort by destination MTA. This is the most common primary sort key.
`user`	Sort by username. This is useful for local deliveries.
`priority`	Sort by the priority of the message.
`time`	Sort by queued time.
`size`	Sort by the size of the message.
`none`	Don't attempt to sort

By default, all outbound channels are sorted by `mta` only. Local channels should be sorted by `user`. Reformatters and other channels are normally sorted by `none`. However, setting a reformatter to `mta` or `user` allows multiple instances of the same channel to run concurrently.

`outinfo=value/ininfo=value:` This contains channel-specific information that can be used for a variety of purposes. For example, the `outinfo` field is used by the simple formatters to give the real program to run. Typically a simple formatter is a shell script or similar, with *fcontrol* indicated as the program. In such cases the *fcontrol* program is then responsible for queue manager interaction and it runs the program in the `outinfo` field as a subprocess.

`inadr=value/outadr=value:` The type of addressing that is used by this channel. The value is one of those below:

Value	Meaning
`x400`	X.400 addressing
`822`	rfc822 addressing

`adr=type:` This assigns the given address format to both the `inadr` and `outadr`. It is used for backwards compatability and as a short cut.

`insubadr=value/outsubadr=value:` The type of subaddressing used by a channel, if appropriate. Currently this is only relevant in conjunction with a definition of `inadr=822` or `outadr=822`. The value is one of those below:

Value	Meaning
`real.rfc822`	normal rfc822 style addressing
`real.rfc733`	rfc733 style addressing
`jnt`	JNT style addressing

For inbound RFC 822 based channels, if `insubadr` is set to either `real.rfc733` or `jnt` then *submit* will recognise percentages as characters with routing properties. If `insubadr` is not set to either of these, *submit* views percentages as characters with no special properties.

`subadr=type:` This variable sets both the `insubadr` and `outsubadr` variables for a channel.

`adr-order=value:` The ordering of addresses on this inbound channel. This is one of the following:

Value	Meaning
`usa`	Internet style addressing is insisted upon.
`uk`	JNT style addressing is insisted upon.
`usapref`	Internet style addressing is preferred but both types will be tried.
`ukpref`	JNT style is preferred but both types will be tried.

The default value is `usa`. Normally only UK sites will need to modify this variable. Note that this flag only affects the behaviour of *submit*.

`bptin=value:` A list of body parts that the channel accepts as input. This must be one of the types defined in the `bodypart` tailoring item defined in Section 3.2.1 on page 18.

`bptout=value:` A list of body parts that the channel will output. The possible values are the same as for those for bptin.

`hdrin=value/hdrout=value:` These variables are much the same as the `bptin/bpout` variables but apply to the headers instead.

`outtable=value:` A table associated with the outbound part of this channel. This is used by the channel program, in a channel specific way. Not all channels require this.

`intable=value:` A table associated with the inbound part of this channel.

`mtatable=value:` This is a table that is used in binding to the appropriate channel. See Section 3.7 on page 33 on channel pairing for use of this feature.

`mta=value:` A destination MTA for this channel. If this is set, all messages will be delivered to this MTA regardless of the destination MTA given in the message. This is useful for relaying all messages for a given channel via another MTA.

`access=value:` The type of access this channel requires; only applicable to channels of type `in` and `both`. The value is either `mta`, in which case the channel must be run by a trusted user-id, or else `mts` in which case authentication of messages is enabled. The default is `mta`.

`domain-norm=value:` The amount of domain normalisation of MTS addresses on inbound channels. The value is one of those below (the default is `partial`):

Value	Meaning
`full`	all domains in an MTS address are normalised
`partial`	only the next hop of an MTS address is normalised

With `partial` local domains are recognised and skipped, so the first non-local domain will be normalised.

`maxproc=value:` This is the maximum number of instances of the channel the QMGR is allowed to run at any time. A value of `0` indicates that there is no maximum to the number of instances the QMGR can run. The default is `0`. See Section 7.1.6 for some discussion on limiting channels in this way.

`auth-table=value:` This is a table which contains the authentication data for the channel if it is an `mts` submission channel. The format is described in Section 4.11 on page 70.

`conv=value:` This variable specifies what effect the channel has on a message. This is used to check on conversion restrictions; in particular the X.400 and RFC 1148 conversion semantics. It may take one of the following values:

Value	Meaning
`none`	channel does no conversion
`1148`	channel does RFC 1148/987 header mappings
`conv`	channel does conversion without loss
`loss`	channel does conversion with loss

The default value is `none`. Channels that do mapping of P2 to RFC 822 *headers* and the reverse should use the value `1148` to indicate that conversion can be carried out even in the presence of the conversion-prohibited flag of X.400. This will allow messages with header mappings and no body part conversions to go through an RFC 1148/987 gateway. Channels that change formats without losing information (except RFC 1148/RFC 987 channels) should be marked as `conv`. Channels which lose data (such as, say, G3Fax to Ia5) should be marked as loss (they should also be given a high cost).

`probe=value:` This variable states whether this channel can support X.400 style probe messages. It has values `y` and `n`. The default

is `n`. This should be set for X.400 channels. If submit determines the channel cannot support probes it will return a delivery report.

`trace=value:` This variable tells *submit* what sort of trace element to generate. This is only applicable to RFC 822-based protocols, and can be set to either `received` to generate Received lines or `via` for Via lines. The default is `received`.

`solo-proc=value:` If this variable is set to `yes`, *submit* ensures that messages processed through this channel only have one recipient. This may result in a message with multiple recipients being resubmitted several times with one recipient each time. This feature is useful for when the processing must be specific to one recipient i.e. the *hdr2fax* filter (see Section 6.1.3). By default, `solo-proc` is set to `no`.

Note this feature uses the `splitter` channel. So such a channel must also be tailored.

`bad-sender-policy=value:` This can be set to one of the following keys to control the policy for unroutable or unreplyable sender addresses.

Value	Meaning
`strict`	Generate delivery report (default).
`sloppy`	Artificially route on failure.

It is STRONGLY recommended that the default policy is used. This will fail messages that arrive with an unroutable or unreplyable sender specification. If this is overridden, messages can easily be lost if any failure occurs.

If the `sloppy` mode is in effect, PP will attempt to route failure messages for unroutable senders either via the inbound MTA or if that fails to `postmaster`.

`check=value:` This variable sets the checking mode of the channel. Normally this should be left as the default, which is strict checking. However, for unusual cases it may be required to relax some of the constraints normally imposed. Setting `check` to `sloppy` will reduce the strictness of the checking to some extent. In particular it will allow RFC 822 messages with no, or multiple `Date` fields to be accepted.

An example of a channel tailoring is shown in Figure 3.3 and will help to make things a little clearer. The descriptions above can be used to determine exactly what each channel does.