3.2.2 Basic Access

The basic access provided to the queue is through a set of library calls. There are several classes of routines which can be divided up into queue manager interaction, address file processing routines and message body routines. Most of the operations described here are only useful for outbound channels.

Queue Manager Interaction

The queue manager interaction is normally handled by a single procedure, `channel_control`, which manages the session with the QMGR and calls the appropriate routines on receipt of ROS operations.

```
int channel_control (argc, argv, initfnx,
                     procfnx, endfnx)
int    argc;
char   **argv;
IFP    initfnx;
struct type_Qmgr_DeliveryStatus   *(*procfnx)();
IFP    endfnx;
```

The arguments are as follows:

`argc/argv`: The argument count and vector, normally copied from the `main` procedure. These are used to recapture the state of the association as outbound channels are normally run under the *pptsapd* (8) daemon.

`initfnx`: An initialisation function. This is called to initialise the channel and inform it of its name.

`procfnx`: A function called for each QMGR process message operation.

`endfnx`: A function called when the session with the QMGR is terminated. The constant `NULLIFP` may be used if this function is not required.

The `channel_control` function invokes its own error handlers and does not normally return.

The two functions `initfnx` and `procfnx` shown above are the key to the channels operation. The initialisation function should be defined in the following way:

```
int   initfnx (arg)
struct type_Qmgr_Channel *arg;
```

The function should then decide whether the given channel argument is sensible for this channel and return one of the constants `OK` or `NOTOK`. This function is normally only called once by the QMGR, but in cases where the same program can run as a number of channels it may be called subsequently to reinitialise the channel to a different mode of operation.

The process function is the one that does the real work of the channel. It should be defined like this:

```
struct type_Qmgr_DeliveryStatus *processmessage (arg)
struct type_Qmgr_ProcMsg *arg;
```

The function should then attempt to process the given message for the given recipients. This usually involves first reading the address control file given as part of the structure passed.

When the processing of the message has finished, the function should return a pointer to a list of recipient numbers together with a status field. This structure is the `type_Qmgr_DeliveryStatus` structure. This specifies for each recipient whether the operation was a success or failure. The status values are:

`int_Qmgr_status_success`: The operation was successful, the next processing stage can be considered.

`int_Qmgr_status_negativeDR`: The message has failed, and a delivery report *must* be written.

`int_Qmgr_status_positiveDR`: The message has succeeded, and a confirmation is required. This *must* be written out.

`int_Qmgr_status_successSharedDR`: The message has suceeded and a delivery report is to be sent. However, the delivery report will be sent as the result of processing another recipient.

`int_Qmgr_status_failureSharedDR`: The message has failed, and a delivery report should be sent for this recipient, but this will be sent as the result of another recipients actions.

`int_Qmgr_status_messageFailure`: The message has temporarily failed; other messages on the same channel may succeed and this message may succeed later at a second attempt.

`int_Qmgr_status_mtaFailure`: The remote MTA is not responding: the attempt should be retried at a later time. Further messages on this channel to this MTA should not be considered at this attempt.

`int_Qmgr_status_mtaAndMessageFailure`: This is an indication that there was a failure whilst transferring the message to the MTA. This channel should be attempted again in the near future.

To aid in setting these values, a set of support routines is provided in the library. These routines all update the global variable `deliverystate` which is suitable as a return value for the process function. Note, however, that the value of `int_Qmgr_status_permanentFailure` is “sticky”; this value may be set, but once set cannot be overwritten by these routines. This is because normally when an address is marked with this status, a delivery report is written out and the message is updated accordingly (see Section 3.2.3 for routines to write out delivery reports). To set the status then to something else would confuse the queue manager. The routines are as follows:

```
struct type_Qmgr_DeliveryStatus *delivery_init (users)
struct type_Qmgr_UserList *users;
```

This routine initialises the structure. It does by allocating an element for each of the users passed in the argument. The single parameter is the list of users passed over by the QMGR.

```
struct type_Qmgr_DeliveryStatus
       *delivery_setstate (rno, val, reason)
int  rno, val;
char *reason;
```

This routine sets the value of the given recipient number to the indicated value in the global structure (with the above proviso). It also includes a string indicating the reason for failure, if failure occurred. This string is passed back to the QMGR for display by the MTAconsole. It should be set to the constant `NULLCP` if the recipient was handled successfully or no diagnostic reason is available.

```
struct type_Qmgr_DeliveryStatus
       *delivery_setallstate (val, reason)
int  val;
char *reason;
```

This routine sets all users' status to the given value. This is useful when a connection cannot be established, for instance, and all recipients must be put in the same state (again, the above proviso about delivery reports holds). The reason field is the same as above.

Occasionally it may occur that a permanent failure is received but it is not possible to write a delivery report (e.g., a full disk). In this case the address file should not be updated and the DR status should be set to message failure (the next retry should also give a DR and then hopefully a report can be written). To aid in resetting these variables the following procedure is useful.

```
struct type_Qmgr_DeliveryStatus
       *delivery_resetDRs (val)
int     val;
```

The argument is the state to set the DR values to (nearly always `int_Qmgr_Status_messageFailure`).

The termination function, if used, should be defined as follows.

```
int endfuction ()
```

This routine is used to close down anything required before exiting the program. The calling routine will execute this function before sending the final disconnect to the QMGR and exiting.

Address File Routines

The address file reading is usually the first operation a channel does on receiving the instruction to process a message. This is usually achieved by first building up the name of the message file to be read in from the queue id, and then invoking the procedure `rd_msg`.

```
int rd_msg (msg, prm, qp, ad_sndr, ad_recip, ad_count)
char  *msg;
struct prm *prm;
Q_struct *qp;
ADDR  **ad_sndr,
      **ad_recip;
int   *ad_count;
```

The parameters are:

`msg`: The address control file.

`prm`: The message parameters.

`qp`: The message queue structure.

`ad_sndr`: the originator of the message.

`ad_recip`: The list of recipients of the message.

`ad_count`: A count of the number of recipient addresses.

The routine returns a status to indicate success or failure. As part of its operation it locks the address file to prevent simultaneous access by other channels.

When the message has been processed, the lock *must* be released by calling the function `rd_end`.

```
int rd_end ();
```

If the messages status changes in some manner, the address file must be updated. This is achieved by one of the following functions.

```
int wr_ad_status (ap, type}
ADDR *ap;
int type;

int wr_ad_rcntno (ap, rcnt)
ADDR *ap;
int  rcnt;
```

The `wr_ad_status` routine is used to update the status of the address. This can take one of the following values.

Value	Meaning
AD_STAT_PEND	The address is pending
AD_STAT_DONE	The address has been finished with

The `wr_ad_rcntno` is used to update the reformatter count variable.

Message files

There are a number of routines to allow a channel to iterate through the set of message files to deliver. First in this set is a function to allow location of the relevant directory. This function maps a given queue id as given by the QMGR into a directory basename with the message files in relating to a given address. It is used as follows:

```
int qid2dir (qid, ap, exists, cpp)
char   *qid;
ADDR   *ap;
int    exists;
char   **cpp;
```

The arguments to this are

`qid`: A character string containing the queue identifier.

`ap`: The address that is being processed.

`exists`: A flag, this should be `TRUE` if the directory should exist, and `FALSE` if it doesn't matter.

`cpp`: A character pointer that is assigned the resulting directory name.

The function will return `OK` on successful completion and `NOTOK` on failure.

The next operation is typically to visit each of the message files in turn. This is handled by the following routines.

```
int msg_rinit (dir)
char *dir;
```

The argument here is:

`dir`: The directory base of the message (usually obtained from a call to `qid2dir`).

This function returns the usual `OK/NOTOK` status.

Following this, the function `msg_rfile` is called repeatedly until it returns `RP_DONE` to obtain each of the files composing the message.

```
int msg_rfile (buf)
char *buf;
```

The argument is:

`buf`: A pointer to a character buffer (of size `FILNSIZE`) which is filled in with the file name if the call succeeds..

Finally, when all files have been processed, or if the operation is to be aborted, the function `msg_rend` is called:

```
int msg_rend ()
```

3.2.3 Delivery Notifications

Delivery reports should be written out by a channel when it is clear that the message cannot be delivered or if the message has been delivered and confirmation is requested. The routine `set_1dr` is then used to set the appropriate status in the address structure:

```
int  set_1dr (qp, ad_no, msg, reason, diag, str)
Q_struct *qp;
int      ad_no;
char   *msg;
int      reason;
int      diag;
char     *str;
```

The arguments to this function are as follows:

`qp`: A pointer to a queue structure representing the message. The queue structure should have the address components filled in (this is normally the case).

`ad_no`: The address number that this report is for.

`msg`: The queue name of the message being processed.

`reason`: A reason code for the message failure. This should be chosen from one of those in Table 3.1 on page 39.

`diag`: A diagnostic code for the message failure. This is supplementary information to the reason code. It should be one of those values given in the file *dr.h*. There are a large number of these available.

`str`: A character string which records the reason for the failure. This should be something that describes the failure. This field should always be filled in with something appropriate.

When calling this function the routine should return to the QMGR one of the following variables:

Qmgr code	When Applicable
`positiveDR`	For the first recipeint with a positive delivery report
`successSharedDR`	For subsequent recipients with positive delivery reports
`negativeDR`	For the first recipient with a negative delivery report
`failureSharedDR`	For subsequent recipients with negative delivery reports

Once the recipients have been marked with `set_1dr`, the last step necessary is to write out the actual delivery report by calling `wr_q2dr`. This is defined as follows:

Reason	Meaning
DRR_NO_REASON	No reason is given for the failure (should not be used normally)
DRR_TRANSFER_FAILURE	Unable to transfer the message to the destination
DRR_UNABLE_TO_TRANSFER	There is a problem with the message that causes the failure
DRR_CONVERSION_NOT_PERFORMED	A conversion is required and cannot be done
DRR_PHYS_RENDITION_NOT_PERFORMED	Can not print the message in the required format
DRR_PHYS_DELIVERY_NOT_PERFORMED	Physical delivery of the message didn't work
DRR_RESTRICTED_DELIVERY	The recipient subscribes to a restricted delivery service.
DRR_DIRECTORY_OP_UNSUCCESSFUL	A required directory operation failed

Table 3.1: Reason Codes

```
int wr_q2dr (qp, msgid)
Q_struct *qp;
char *msgid;
```

The arguments to this function are:

qp: A pointer to the queue structure representing the message with the address components filled in.

msgid: The queue id for this message.

This function will then write out the reports and update the queue entries with their new status.

3.2.4 Successful Delivery

A successful delivery is recorded in the stat log. This is done by the routine wr_q2dr if a positive delivery report is required. Alterntively, if no report is required, the delivery is noted with the function wr_stat which is called as follows.

```
int wr_stat (ap, qp, msg, size)
ADDR *ap;
Q_struct *qp;
char *msg;
int size;
```

The parameters to this are:

ap: The address that has been successfully delivered.

`qp`: The queue structure for this message.

`msg`: The queue name for this message.

`size`: The size in bytes of the message delivered. This may be different from the size in the queue file.

When a message has been delivered successfully, it is necessary to check several things. These are:

1. If a positive delivery report was requested, one should be composed with `set_1dr` and `wr_q2dr`.

2. Otherwise, the status should be updated to `AD_DONE` with a call to `wr_ad_status`.

3. The QMGR return status should be updated to either `int_Qmgr_Status_success` or `int_Qmgr_Status_positiveDR`.

4. If `wr_q2dr` is not necessary (there was no request for positive delivery reports). A call should be made to `wr_stat` to indicate in the statistics that successful delivery has been achieved.

3.2.5 Verification and Trust

Channels are trusted processes, and as such should respect this trust. As far as possible they should check that the data they are being given is reasonable. This includes the following:

- That the channel name and type is reasonable for this channel.
- That the address requested to be processed exists and is ready to be processed (e.g., it has passed through all reformatters).
- That the type of address is correct for this type of channel (e.g., RFC 822 for smtp or X.400 for X.400 channels).
- That the message is not locked by another channel.
- That the channel is the same as the channel given in the address structure.

For common outbound channels and filters these checks are encapsulated in the functions, `chan_acheck`, `lchan_acheck` and `chan_dcheck`.

These calls are similar in function; one checks the address is okay to process for filters and outbound channels; the second is for local channels; and the last is for channels that can handle delivery reports. They are called in the following way:

```
int chan_acheck (ap, mychan, first, mta)
ADDR *ap;
CHAN *mychan;
int first,
char **mta;
```

The arguments are as follows:

ap: The address to be checked.

mychan: The channel that is doing this checking.

first: If this is the first address to be checked. This is used to determine if the mta parameter can be changed.

mta: The MTA this should connect to. If the constant `NULLVP` is used, the mta parameter will not be filled in or checked. If the mta is the address of a pointer with `NULL` contents then it will be updated if required. If the mta is a pointer to a string, then it is checked that the mta does not change unless the `first` flag is set. This ensures for channels that batch messages to the same host, that all addresses in this message are destined for the same MTA.

It returns either the constant `OK` or `NOTOK`. If `NOTOK` then the `deliverystate` is updated in a suitable way. If `OK` then the address should be processed, and the `deliverystate` updated accordingly.

The function `lchan_acheck` is almost equivalent, except in place of the `mta` parameter is a user.

```
int lchan_acheck (ap, mychan, first, user)
ADDR *ap;
CHAN *mychan;
int first,
char **user;
```

The last function is similar but suitable for channels that can handle delivery reports (such as X.400).

```
int dchan_acheck (ap, asp, mychan, first, mta)
ADDR *ap, *asp;
CHAN *thechan;
int first;
char **mta;
```

The only difference for this call is that an address structure referencing the originator must be passed in as the second parameter.

3.3 Example Usage

3.3.1 Outbound Channels

Presented here is a skeleton of an outbound channel. This structure is not required, but is a common way of achieving the results.

```
main (argc, argv)
int     argc;
char    **argv;
{
        sys_init (argv[0]);   /* init the channel – and find out who we are */
```

```
        dirinit();              /* get to the right directory */
#ifdef PP_DEBUG
        if (argc > 1 && strcmp (argv[1], "debug") == 0)
                debug_channel_control (argc, argv, chaninit, process, endproc);
        else
#endif
                channel_control (argc, argv, chaninit, process, endproc);
        exit (0);
}

static int chaninit (arg)
struct type_Qmgr_Channel *arg;
{
        char    *p = qb2str (arg);

        if ((mychan = ch_nm2struct (p)) == (CHAN *)0)
                err_abrt (RP_PARM, "Channel '%s' not known", p);
        exam_init (mychan);
        free (p);
        return OK;
}

static int endproc ()
{
        if (cur_host)
                exam_close (OK);
}

static struct type_Qmgr_DeliveryStatus *process (arg)
struct type_Qmgr_ProcMsg *arg;
{
        struct prm_vars prm;
        struct type_Qmgr_UserList *up;
        Q_struct        Qstruct, *qp = &Qstruct;
        int     retval;
        ADDR    *ap,
                *ad_sendr = NULLADDR,
                *ad_recip = NULLADDR,
                *alp,
                *ad_list = NULLADDR;
        int     ad_count;

        if (this_msg) free (this_msg);

        this_msg = qb2str (arg -> qid);

        PP_TRACE (("process msg %s", this_msg));

        bzero ((char *)&prm, sizeof prm);
        bzero ((char *)qp, sizeof *qp);

        (void) delivery_init (arg -> users);

        retval = rd_msg (this_msg, &prm, qp, &ad_sendr, &ad_recip, &ad_count);

        if (rp_isbad (retval)) {
                PP_LOG (LLOG_EXCEPTIONS, ("rd_msg err:  %s", this_msg));
                return delivery_setall (int_Qmgr_status_messageFailure);
```