`rp_val`: The value returned; this is one of those defined in *retcode.h*.

`rp_line`: An optional explanatory string.

2.2.3 Procedure Calls

The submission interface is procedural and has a handshake at most of the interaction points. This interaction is reasonably complex but allows the submission to be flexible; this interface can be used to validate address as well as to submit message.

Most of these routines return a set of codes as defined in *retcode.h* to indicate success or failure of the operation. The codes indicate detailed status and are shown in Table 2.3. Two macros are useful for determining the general status of the reply. The behave as though they were defined as:

```
int rp_isgood (code)
int code;

int rp_isbad (code)
int code;
```

where "code" is the return code. The macros return boolean values of `TRUE` if the code is generally good or bad respectively.

Along with other parameters, these routines usually take an `RP_Buf` structure which is updated with the code and an explanatory string if the routine fails in some way.

Initialisation

To initialise the submission system it is first necessary to call the routine `io_init`:

```
int io_init (rp)
RP_Buf *rp;
```

This call initialises the submission by either invoking *submit* as a child process or by connecting to a *submit* daemon. It returns a good or bad status, and fills in the `rp` structure if the response is bad.

Message Management Parameters

The first set of parameters to negotiate with *submit* are the management parameters. These are defined in the structure in Figure 2.3 on page 11. This structure is passed over using the `io_wprm` routine:

```
int io_wprm (prm, rp)
struct prm_vars *prm;
RP_Buf *rp;
```

The parameter structure is often only sparsely filled (or even empty), as the default values assumed are usually correct for most common submissions.

Value	Meaning
RP_DONE	General purpose done value
RP_OK	General purpose OK value
RP_MOK	Message accepted
RP_MAST	Master side
RP_SLAV	Slave side
RP_AOK	Address OK
RP_HOK	Host OK
RP_AGN	Temporary failure
RP_TIME	Timeout
RP_NOOP	No op
RP_EOF	End of file
RP_NET	Network failure
RP_BHST	Bad host interaction
RP_DHST	destination connection broke
RP_NIO	Network I/O problem
RP_NS	Temporary nameserver failure
RP_FIO	File I/O problem
RP_FCRT	File creation problem
RP_FOPN	File opening problem
RP_LIO	Local I/O problem
RP_LOCK	Resource locked
RP_MECH	Mechanism failed
RP_NO	General purpose failure
RP_BAD	General purpose failure
RP_PROT	General protocol failure
RP_RPLY	Bad reply code
RP_NAUTH	Bad authorisation
RP_NDEL	Couldn't deliver
RP_HUH	Parse error
RP_NCMD	No such command
RP_PARM	Bad parameter
RP_UCMD	Unknown command
RP_PARSE	Address parse error
RP_USER	Unknown user

Table 2.3: Return Codes

Per-Message Parameters

The next stage is the per-message parameters. This is achieved by filling in a `Q_struct` structure and passing it to *submit*. Once the structure is complete it can be written to *submit* using the `io_wrq` routine:

```
io_wrq (qp, rp)
Q_struct *qp;
RP_Buf *rp;
```

Originator and Recipient Addresses

After the general parameters have been passed by the above routines, the send is passed over in the standardised address structure. The call to achieve this is the `wr_adr` routine:

```
int io_wadr (ap, type, rp)
ADDR *addr;
int  type;
RP_Buf *rp;
```

The type parameter is either `AD_ORIGINATOR` or `AD_RECIPIENT`. First the originator address is passed over and checked followed by the list of recipients. This is then followed by each of the recipient addresses. In this way, each address is checked and any failures can be pinpointed.

When all the addresses have been submitted, the address list is terminated with the call:

```
int io_adend (rp)
RP_Buf *rp;
```

This concludes the preliminaries, and the body of the message can be transferred.

Body Part Submission

The body parts presented are dependent on the type of message being transferred. To initialise for body part submission, the following function is called:

```
io_tinit (rp)
RP_Buf *rp;
```

This initialises for text submission.

Following this call it is necessary to initialise for the first body part. This is done by:

```
int io_tpart (name, link, rp)
char    *name;
int     link;
RP_Buf *rp;
```

The name is the name of this part, for an RFC 822 body part it is usually something like `hdr.822` and `2.ia5` for the two parts of the message. For X.400, it is more often just a single file `p2` or similar with the whole of the message in. The link parameter is a Boolean value to indicate whether the name is actually a secret link to another file in the queue to be linked with *link* (2); this is only for use by special channels and normally will be rejected by *submit*. If the link is false, then the data stream is read for the body part contents.

All Data is passed with the following routine:

```
io_tdata (buffer, length)
char *buffer;
int  length;
```

The buffer is a pointer to an array of characters of length `length`, which are written into the file created by the `io_tpart` call.

Once the body part has been written, the data stream is terminated by the call:

```
io_tdend (rp)
RP_Buf   *rp;
```

which closes the stream and returns the status in rp.

Termination

Once all the body parts have been passed over the message is finally submitted with the following call:

```
io_tend (rp)
RP_Buf   *rp;
```

This puts the message into the queue and allows it to be delivered.

Finally, the whole session with *submit* can be terminated using the `io_end` call. The termination can either be graceful, or a program can, upon detecting an error, call an abort to the submission process at any time with this function:

```
int io_end (type)
int type;
```

The `type` parameter is set to one of the following:

Value	Meaning
`OK`	The submission is successful
`NOTOK`	The submission should be aborted

2.2.4 Example

An example interaction with submit is shown in Figure 2.7.

---

submit_message (orig, recips, fp)

```
ADDR     *orig;
ADDR     *recips;
FILE     *fp;
{
         RP_Buf    rps, *rp = &rps;
         char      buf[BUFSIZ];
         ADDR      *ap;

         if (rp_isbad (io_init(rp)))
                   error ();

         if (rp_isbad (io_wprm (prm, rp)))
                   error ();

         if (rp_isbad (io_wrq (q, rp)))
                   error ();

         if (rp_isbad (io_wadr (orig, AD_ORIGINATOR, rp)))
                   error ();

         for (ap = ad_recipients; ap; ap = ap -> ad_next)
                   if (rp_isbad (io_wadr (ap, AD_RECIPIENT, rp)))
                             error ();

         if (rp_isbad (io_adend(rp)))
                   error ();

         if (rp_isbad (io_tinit(rp)))
                   error ();

         if (rp_isbad (io_tpart ("hdr.822", FALSE, rp)))
                   error ();

         while (fgets( buf, sizeof buf, fp))
                   if (rp_isbad (io_tdata (buf, strlen (buf))))
                             error ();

         if (rp_isbad (io_tdend(rp)))
                   error ();

         if (rp_isbad (io_tpart ("1.ia5", FALSE, rp)))
                   error ();

         while (fgets (buf, sizeof buf, stdin))
                   if (rp_isbad (io_tdata (buf, strlen (buf))))
                             error ();

         if (rp_isbad (io_tdend (rp)))
                   error ();

         if (rp_isbad (io_tend (rp)))
                   error ();

          io_end(OK);
}
```

Figure 2.7: Example Interaction with Submit

2.3 The RFC 822 Interface

PP provides a full service interface, most of which is not needed by RFC 822-based systems. A simplified interface is provided on top of the `io_` interface. This interface is characterised by procedure calls with the prefix `pps_` for PP simple interface.

It assumes a large number of default parameters and does not allow for much variation. It does provide an easy route for constructing */bin/mail*-like programs and can be used within programs to fire off system generated messages. This interface runs as a state machine, and the procedures must be called in the correct order to get the correct results.

All routines return the constant `NOTOK` on failure and `OK` on success. The `NOTOK` is unrecoverable, and the interface must be started again from the beginning. All routines in addition take an `RP_Buf` pointer which is updated if the call fails. This may be used to give better diagnostics.

2.3.1 Initialisation

To initialise this interface, a client program calls the `pps_init` routine. This attempts to initialise the association with *submit*.

```
pps_init (subject, rp)
char    *subject;
RP_Buf  *rp;
```

The parameters to this routine are:

`subject`: the subject line of the message, the constant `NULLCP` may be used if no subject line is required.

`rp`: An `RP_Buf` structure pointer that is updated if the call fails.

The errors produced by this call are usually the result of some deep system problem, such as failing to connect with the *submit* process.

2.3.2 Addresses and Headers

To supply recipient addresses for the message, the routines `pps_adr`, `pps_to` and `pps_cc` are called. The `pps_adr` supplies a new address to the system for this message. Use of the `pps_to` and `pps_cc` routines switches the status of the address between that of primary recipient and secondary recipient (i.e., the To and CC header lines).

```
pps_adr (address, rp)
char    *address;
RP_Buf *rp;

pps_to(rp)
RP_Buf *rp;

pps_cc(rp)
RP_Buf *rp;
```

The arguments are:

`address`: The recipient address as a legal RFC 822 format address.

`rp`: An `RP_Buf` structure pointer that is updated if the call fails.

The possible errors from these calls usually arise from the failure to call `pps_init`. The `pps_to` and `pps_cc` calls are entirely optional and may be interspersed with the `pps_adr` calls in any order.

Finally, when the addresses have all been submitted, the address phase is ended by calling `pps_aend`.

```
pps_aend (rp)
RP_Buf *rp;
```

At this stage, other headers may optionally be added by the `pps_hdr` routine. This is normally not necessary as the header fields `To`, `Cc`, `From`, and `Date` will automatically be generated.

```
pps_hdr (name, contents, rp)
char    *name,
        *contents;
RP_Buf *rp;
```

If the message is only being submitted to one recipient, there is a shorthand form which replaces the routines from `pps_init` up to and including the `pps_tinit` call. This routine, `pps_1adr`, is occasionally useful for submitting error reports to the system administrator or similar.

```
pps_1adr (subject, address, rp)
char   *subject,
       *address;
RP_Buf *rp;
```

The parameters to this routine are:

`subject`: The subject of the message. As above this can be the constant `NULLCP` for no subject.

`address`: The RFC 822 format recipient address

2.3.3 Text Submission

To add the body of the message, the routine `pps_tinit` is called first to initialise for text. This terminates the header part of the message and prepares for the text to be submitted.

```
pps_tinit (rp)
RP_Buf *rp;
```

The text is then submitted in two ways. The first method is as a sequence of null terminated character strings using the `pps_txt` routine. The second method is by passing an open file descriptor to the routine `pps_file`.

```
pps_txt (string, rp)
char *string;
RP_Buf *rp;

pps_file (fp, rp)
FILE *fp;
RP_Buf *rp;
```

The parameters to these calls are:

`string`: A null terminated string of characters.

`file`: A stdio file stream opened for reading. This is read up to the end of file.

These calls may be freely intermixed in any fashion.

2.3.4 Termination

To finally submit the message, it is necessary to call the routine `pps_end`. This terminates the interaction for this message and resets the state.

```
pps_end (status, rp)
int status;
RP_Buf *rp;
```

The status parameter is either `OK` if the submission is satisfactory and `NOTOK` if the submission is to be aborted. The `NOTOK` version may be called at any point to abort the submission and return to the `pps_init` state.

2.4 Debugging

Most of the debugging can be done through the logs. Typically turning the logging levels to include trace and debug will give large amounts of information and will slow the processes down considerably! The system should not normally be run with full logging for these reasons.

The use of the `PP_TRACE` and `PP_DBG` macros described in Section 1.4.3 on page 3 are very useful for inclusion in program text. the `PP_TRACE` should be used to indicate the flow of control. Usually a `PP_TRACE` call is included in each significant procedure call and typically displays the procedure name and any useful information about procedure arguments.

The `PP_DBG` macro is used anywhere that very detailed tracing might be needed such as printing out individual variables or similar. In particular, library routines usually use this macro to log tracing, as this sort of logging is not required as often as the tracing in programs.

The `PP_NOTICE` call should be used when interesting events happen to allow the progress of the program to be monitored.

The `PP_PDUS` macro and the `LLOG_PDUS` logging level should be used to log any protocol exchanges that might be of interest.

Chapter 3

Channels

3.1 Overview

This chapter describes the interface which provides the message delivery aspects of the message transfer abstract service and MTA abstract service. The major component of this is a set of routines for accessing the PP queue. As these routines are usually used in the context of a channel process, their usage is illustrated in various types of channel. There are also additional functions to facilitate reformatting/conversion.

The following types of channel are supported:

- Protocol outbound channel (MTA AS delivery).
- Local delivery channel (MT AS delivery).
- Local pickup channel (MT AS delivery).
- Remote TWA channel (MTA AS delivery).
- Reformatters.

3.2 The Queue Interface

3.2.1 Major Structures

The major structures in use for reading the queue are the following:-

- The parameter structure described in Figure 2.3 on page 11.
- The queue structure described in Figure 2.4 on page 12.
- The address structure described in Figure 2.5 on page 18.

There is also a separate set of routines and structures for interacting with the queue manager. This interface makes use of the structures defined in Figure 3.1.

These structures are derived from the queue manager ROS specification given in Appendix F of Volume 1.

```
struct type_Qmgr_ProcMsg {
    struct type_Qmgr_QID *qid;

    struct type_Qmgr_UserList *users;

    struct type_Qmgr_Channel *channel;
};

struct type_Qmgr_DeliveryStatus {
        struct type_Qmgr_IndividualDeliveryStatus *IndividualDeliveryStatus;

        struct type_Qmgr_DeliveryStatus *next;
};

struct type_Qmgr_IndividualDeliveryStatus {
    struct type_Qmgr_RecipientId *recipient;

    integer     status;
#define  int_Qmgr_status_success      0
#define  int_Qmgr_status_successSharedDR    1
#define  int_Qmgr_status_failureSharedDR    2
#define  int_Qmgr_status_negativeDR         3
#define  int_Qmgr_status_positiveDR         4
#define  int_Qmgr_status_messageFailure     5
#define  int_Qmgr_status_mtaFailure         6
#define  int_Qmgr_status_mtaAndMessageFailure       7

    struct          qbuf      *info;
};
```

Figure 3.1: Channel / Queue Manager Interface Structures