wr_ad_rcntno 37
wr_ad_status 37
wr_q2dr 40

MTAconsole 36

O

or/ 5

P

parameter structure 11
parse/ 6
pp/ 6
PP_DBG 4, 31
PP_DEBUG_ALL 4
PP_DEBUG_SOME 5
pp/Lib/tai/tai_sys.c 2
PP_LOG 3
PP_NOTICE 4, 31
PP_OPER 3
PP_PDU 5
PP_PDUS 31
pps_1adr 30
pps_adr 29
pps_aend 29
pps_cc 29
pps_end 31
pps_file 30
pps_hdr 30
pps_init 29
PP_SLOG 3
PP_STAT 4
pps_tinit 30
pps_to 29
pps_txt 30
PP_TRACE 4, 31
pptsapd/ 6
pptsapd 34
prm 11
prm_free 11
prm.h 11
prm_init 11

Q

q_free 13
q.h 11
qid2dir 38, 45
q_init 13
qmgr/ 6
qmgr 36
Q_struct 13, 25
queue structure 13

R

rd_end 37
rd_msg 37
Reply structure 23
retcode.h 23
RP_Buf 23
rp_isbad 23
rp_isgood 23

S

set_1dr 39
SLOG 5
Src/ 6
status_failureSharedDR 35
status_messageFailure 35
status_mtaAndMessageFailure 35
status_mtaFailure 35
status_negativeDR 35
status_positiveDR 35
status_success 35
status_successSharedDR 35
struct prm_vars 11
submit/ 6
submit 8, 11, 23, 25, 26, 27, 28, 29
submit protocol 8
sys_init 1

T

table/ 6
tai/ 6
Tools/ 6
type_Qmgr_Channel 32
type_Qmgr_DeliveryStatus 32
type_Qmgr_ProcMsg 32
type_Qmgr_QID 32
type_Qmgr_RecipientId 32

U

Uip/ 6
util/ 6

W

Index

A
addr/ 5
ADDR 20
address structure 20
addressing 16
adr_add 17
adr_free 17
adr.h 16
adr_init 17
adr_new 17
adr_tfree 17
ASN.1 5

B
/bin/mail 28

C
chan_acheck 41
channel_control 34
Chans/ 7

D
debugging 31
delivery_init 36
delivery_set 36
delivery_setall 36
doc/ 6

F
fcontrol 45
format/ 5

H
h/ 6

I
io/ 5
io_adend 26
io_end 27
io_init 25
io_tdata 26
io_tdend 26
io_tend 27
io_tinit 26
io_tpart 26
io_wadr 25
io_wprm 25
io_wrq 25
isode/logger.h 2

L
LaTeX 46
lchan_acheck 41
Lib/ 5
link 26
list_bpt_add 10
list_bpt_dup 10
list_bpt_find 10
list_bpt_free 10
list_bpt_new 10
LIST_RCHAN 9
list_rchan_add 9
list_rchan_free 9
list_rchan_new 9
list_rchan.h 9
LLog 2
ll_log.h 3, 11
log_norm 2
log_oper 2
log_stat 2

M
man/ 6
msg_rend 38
msg_rfile 38
msg_rinit 38
MTAconsole/ 6

```
                DRD_UNRECOGNISED_OR, buffer);
    }
...
```

Figure 3.3: Example Delivery Report Generation

3.4 Formatters and Converters

Formatters and converters can run in two ways. Generally the simple filters work on single files, the complex filters work on several files at once. The majority of filters should be of the simple kind.

3.4.1 Simple converters

The simple converters are programs that reads from the standard input, do some formatting and write the result to the standard output. This type of formatter is invoked by the filtercontrol program, `fcontrol`.

These simple filters should return a suitable exit status to confirm that the conversion completed successfully. They follow the standard UNIX[1] tradition of returning `0` on success and anything else on failure.

These filters may be anything from simple shell/awk/sed scripts to more complex *C* programs.

3.4.2 Complex Filters

More complex formatters are necessary if the formatting needs to convert more than one file. This is the case for formatters that change the directory structure, or combine several files into one. These formatters require more work, but are similar to the channel programs in their operation. The `qmgr` interface is in fact identical. The only real difference is in the `qid2dir` routine used to locate the directory. The starting directory is found using the normal `qid2dir` routine. The destination directory is found using the same routine, but first incrementing the formatter count in the address structure, and then calling the routine with the flag set to `FALSE`. The formatter should however place it's output in a temporary directory which will then be renamed to the destination directory if the transformation is successful. This allows for failure at any point in the reformatting.

3.4.3 Example

to be written.

[1] UNIX is a trademark of AT&T Bell Laboratories.

```
        }                                                              90

        if (ad_list == NULLADDR) {
                PP_LOG (LLOG_EXCEPTIONS, ("No recipients in user list"));
                rd_end ();
                return deliverystate;
        }

        deliver (ad_list, qp); /* do what is required */

        rd_end();                                                     100

        return deliverystate;
}


static void dirinit()       /* Change into pp queue space */
{
        if (chdir (quedfldir) < 0)
                err_abrt (RP_LIO, "Unable to change directory to '%s'",
                                                        quedfldir);   110
}
```

Figure 3.2: Example outbound channel

3.3.2 Local Pickup

This needs to be written - both in code and here!

3.3.3 TWA channel

Not yet written

3.3.4 Delivery Reports

A typical interface to the delivery report functions might be the following.

```
...
   {
        char buffer[BUFSIZ];

        (void) sprintf (buffer, "Address %s failed for reason %s",
                        ap -> ad_r822adr, rstr);

        (void) delivery_set (ap -> ad_no,
                             int_Qmgr_status_permanentFailure);
        set_dr (qp, ap -> ad_no, msg_name,                            10
                DRR_UNABLE_TO_TRANSFER,
```

```
struct prm_vars prm;
struct type_Qmgr_UserList *up;
Q_struct        Qstruct, *qp = &Qstruct;                                40
int     retval;
ADDR    *ap,
            *ad_sendr = NULLADDR,
            *ad_recip = NULLADDR,
            *alp,
            *ad_list = NULLADDR;
int     ad_count;

if (this_msg) free (this_msg);
                                                                        50
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);  60

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

sender = ad_sendr -> ad_r822adr;


for (ap = ad_recip; ap; ap = ap -> ad_next) {                           70
            for (up = arg ->users; up; up = up -> next) {
                        if (up -> RecipientId -> parm != ap -> ad_no)
                                    continue;

                        if (chan_acheck (ap, mychan,
                                                ad_list == NULL, &cur_mta) == NOTOK)
                                    continue;
            }
            if (up == NULL)
                        continue;                                       80

            if (ad_list == NULLADDR)
                        ad_list = alp = (ADDR *) calloc (1, sizeof *alp);
            else {
                        alp -> ad_next = (ADDR *) calloc (1, sizeof *alp);
                        alp = alp -> ad_next;
            }
            *alp = *ap;
            alp -> ad_next = NULLADDR;
```

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

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;
{
```

- 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 two functions, `chan_acheck` and `lchan_acheck`.

These two calls are similar in function, one checks the address is ok to process for filters and outbound channels, the other for local channels. 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.

Diagnostic	Meaning
DRD_UNRECOGNISED_OR	The sender or recipient name is unknown
DRD_AMBIGUOUS_OR	The sender or recipient name is ambiguous
DRD_MTA_CONGESTION	Problems with the MTA (catch all!)
DRD_LOOP_DETECTED	A loop has been detected in routing
DRD_UA_UNAVAILABLE	The user agent is not available
DRD_MAX_TIME_EXPIRED	The message was not delivered before the specified delivery time
DRD_ENCINFOTYPES_NOTSUPPORTED	The encoded information types are not supported at the destination
DRD_CONTENT_TOO_LONG	The message is too big to be delivered
DRD_CONVERSION_IMPRACTICAL	The conversion required by this message is not sensible
DRD_CONVERSION_PROHIBITED	The conversion required for delivery is prohibitied
DRD_IMPLICITCONV_NOTREGISTERED	The implicit conversion required can't be performed
DRD_INVALID_PARAMETERS	There is a problem with the message parameters

Table 3.2: Diagnostic codes

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 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:

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

Table 3.1: Reason codes

`reason:` A reson 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 Table 3.2 on page 40.

`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:

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

```
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, reason, diag, str)
Q_struct *qp;
int      ad_no;
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.

```
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 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 qid, 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_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 are 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 can not 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 then set the status 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 occured. 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.

```
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 can't 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.

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

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, 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 the 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.

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 it's 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 it's 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;
```

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.

```
#define type_Qmgr_Channel       qbuf
#define type_Qmgr_QID  type_UNIV_PrintableString

struct type_Qmgr_UserList {
       struct type_Qmgr_RecipientId *RecipientId;
       struct type_Qmgr_UserList *next;
};

struct type_Qmgr_RecipientId {
   int     parm;                                                    10
};

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;20
       struct type_Qmgr_DeliveryStatus *next;
};

struct type_Qmgr_IndividualDeliveryStatus {
   struct type_Qmgr_RecipientId *recipient;
   int     status;
#define int_Qmgr_status_success    0
#define int_Qmgr_status_permanentFailure            1
#define int_Qmgr_status_messageFailure     2
#define int_Qmgr_status_mtaFailure          3                       30
#define int_Qmgr_status_mtaAndMessageFailure       4
};
```

Figure 3.1: Channel / Queue Manager Interface Structures

These structures are derived from the queue manager ROS specification given in Volume 3.

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

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 17.

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.

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.

```
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 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;
```

```
        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 ();                                              20

        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 ();                                              30

        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 ();                                              40

        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 ();                                              50

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

         io_end(OK);
}
```

Figure 2.7: Example interaction with submit

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)))                                    10
                error ();
```

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 ofter 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;
```

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.

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;
```

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

2.2.2 Reply value structure

The reply value buffer structure is shown in Figure 2.6. This is used in all handshakes with *submit* to return a status and an explanatory string.

The structure here is:

`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.

```
typedef struct rp_bufstruct     /* for reading reply strings */
{
   unsigned char      rp_val;
   char               rp_line[256];
} RP_Buf;
```

Figure 2.6: Reply Structure

ad_recip_number_for_advice: The number for advice to call for delivery.

ad_phys_rendition_attribute: The physical rendition attributes for the address when transfered to the postal service.

ad_pd_report_request: An option to supply a physical delivery report.

ad_redirection_history: The redirection history for this address.

ad_message_token: A security parameter for message tokens.

ad_content_integrity: A content integrity check parameter.

ad_proof_delvery: Proof of delivery is requested.

ad_resp : The responsibility bit. This should be set to either TRUE or FALSE.

ad_mtareq: This maps to the X.400 mta report request bits. it should be set to one of the following:

Value	Meaning
AD_MTA_NONE	Undefined value
AD_MTA_BASIC	Generate Delivery failures
AD_MTA_CONFIRM	Generate Delivery confirmations
AD_MTA_AUDIT_CONFIRM	Generate full trace confirmations

ad_subtype: The sub type of the address if appropriate. This is only used for rfc-822 addresses at present. The values are:

Value	Meaning
AD_NOSUBTYPE	No particular sub type
AD_JNT	JANET style address
AD_REAL733	RFC-733 style address
AD_REAL822	RFC-822 style address

ad_pre_recip_ext_list: A list of extension attributes.

ad_value: This is the address submitted.

ad_next: The link to the next address if they are in a linked list.

Value
AD_EXP_NONE
AD_EXP_IA5_TEXT_TO_TELETEX
AD_EXP_TELETEX_TO_TELEX
AD_EXP_TELEX_TO_IA5_TEXT
AD_EXP_TELEX_TO_TELETEX
AD_EXP_TELEX_TO_G4_CLASS_1
AD_EXP_TELEX_TO_VIDEOTEX
AD_EXP_IA5_TEXT_TO_TELEX
AD_EXP_TELEX_TO_G3_FACSIMILE
AD_EXP_IA5_TEXT_TO_G3_FACSIMILE
AD_EXP_IA5_TEXT_TO_G4_CLASS_1
AD_EXP_IA5_TEXT_TO_VIDEOTEX
AD_EXP_TELETEX_TO_IA5_TEXT
AD_EXP_TELETEX_TO_G3_FACSIMILE
AD_EXP_TELETEX_TO_G4_CLASS_1
AD_EXP_TELETEX_TO_VIDEOTEX
AD_EXP_VIDEOTEX_TO_TELEX
AD_EXP_VIDEOTEX_TO_IA5_TEXT
AD_EXP_VIDEOTEX_TO_TELETEX

Table 2.2: Table of Explicit Conversion values

`ad_type`: The address type. This is one of the following values:

Value	Meaning
AD_X400_TYPE	An X.400 format address
AD_822_TYPE	An RFC-822 format address
AD_ANY_TYPE	An address in any of the above formats

`ad_orig_req_alt`: The originator requested alternate recipient, if supplied.

`ad_req_del`: The requested delivery method. This is an array of delivery method alternative.

`ad_phys_forward`: If physical forwarding is allowed.

`ad_phys_fw_ad_req`: A request to supply the physical forwarding address of the recipient.

`ad_phys_modes`: Specifies the type of delivery required.

`ad_reg_mail_type`: The type of registered mail in use.

```
        /* -- from here onwards extra PP specific info -- */
        /* -- Diagnostics for any address parsing failures -- */

        int             ad_parse_stat;      /* address parsing err status */
        int             ad_reason;          /* DR reason failure */
        int             ad_diagnostic;      /* DR diagnostic code */
        char            *ad_add_info;       /* DR supplementary info */
                                                                        160

        /* -- Address Control File offsets for fixed length variables -- */

        off_t           ad_no_offset;       /* offset param for recip number */
        off_t           ad_stat_offset; /* offset param for recip status */
        off_t           ad_rcnt_offset; /* offset param for reformat next */


        struct adr_struct   *ad_next;
} ADDR;                                                                 170
```

Figure 2.5: Address Structure

Only the non-internal fields are described here:

`ad_extension:` The extension identifier for this address.

`ad_status:` The status of this address, one of:-

Value	Meaning
AD_STAT_PEND	for recipients awaiting delivery
AD_STAT_DONE	for recipients not to be delivered to

`ad_dn:` The distinguished name component of the address.

`ad_usrreq:` This maps to the X.400 user request bits. It has the one of the values:

Value	Meaning
AD_USR_NOREPORT	No delivery reports requested
AD_USR_BASIC	Delivery failutes should be reported
AD_USR_CONFIRM	Deliveries should be confirmed
AD_USR_NONE	Undefined value

`ad_explicitconversion:` Is explicit conversion required. It should be one of the values defined in table 2.2.

```
#define AD_NTF_MHS              2
#define AD_NTF_BOTH             3                                       100
        Redirection     *ad_redirection_history;
        char            ad_redirection_history_crit;

        struct qbuf     *ad_message_token;
        char            ad_message_token_crit;

        struct qbuf     *ad_content_integrity;
        char            ad_content_integrity_crit;

        int             ad_proof_delivery;                              110
        char            ad_proof_delivery_crit;
                                /* boolean.  Is proof of delivery */
                                /* requested */

        /* -- MTA Service Params (calculated for MTS AS) */

        int             ad_extension;       /* extension-id */
        int             ad_resp;            /* responsibility bit */
        int             ad_mtarreq;         /* mta-report-request bit */
#define AD_MTA_NONE             0                                       120
#define AD_MTA_BASIC            1
#define AD_MTA_CONFIRM          2
#define AD_MTA_AUDIT_CONFIRM    3

        /* -- This may be supplied as a hint  but might get changed */

        int             ad_subtype;         /* address-subtype */
#define AD_NOSUBTYPE            0
#define AD_JNT                  1
#define AD_REAL733              2                                       130
#define AD_REAL822              3
#define AD_X400_84                          4  /* Use this if compatible */
#define AD_X400_88                          5


        Extension       *ad_per_recip_ext_list;
                                            /* bucket for new and private */
                                            /* odds and sods */

        /* -- parameters calculated by PP */                            140
        /* -- Note: the outbound mta is held in ad_outchan -- */

        char            *ad_r400adr;        /* x400-addr */
        char            *ad_r822adr;        /* rfc-addr */

        char            *ad_content;        /* outgoing content type */

        LIST_RCHAN      *ad_fmtchan;        /* reformatting channels */
        LIST_RCHAN      *ad_outchan;        /* outbound channel structure */
        LIST_BPT        *ad_eit;            /* outgoing eit's */        150
```

```
        int             ad_req_del[AD_RDM_MAX];     /* requested delivery method */
#define AD_RDM_NOTUSED  -1
#define AD_RDM_ANY      0
#define AD_RDM_MHS      1                                                   50
#define AD_RDM_PD       2
#define AD_RDM_TLX      3
#define AD_RDM_TTX      4
#define AD_RDM_G3       5
#define AD_RDM_G4       6
#define AD_RDM_TTY      7
#define AD_RDM_VTX      8
        char            ad_req_del_crit;

        char            ad_phys_forward; /* boolean - is physical          60          */
                                         /* forwarding allowed */
        char            ad_phys_forward_crit;

        char            ad_phys_fw_ad_req;
        char            ad_phys_fw_ad_crit;
                                         /* boolean - request for phys */
                                         /* foward address */

        int             ad_phys_modes;
        char            ad_phys_modes_crit;                                 70
#define AD_PM_ORD       0x1
#define AD_PM_SPEC      0x2
#define AD_PM_EXPR      0x4
#define AD_PM_CNT       0x8
#define AD_PM_CNT_PHONE 0x10
#define AD_PM_CNT_TLX   0x20
#define AD_PM_CNT_TTX   0x40
#define AD_PM_CNT_BUREAU        0x80
#define AD_PM_MAX       0x80
                                                                            80
        int             ad_reg_mail_type;
        char            ad_reg_mail_type_crit;
#define AD_RMT_UNSPECIFIED      -1
#define AD_RMT_NON_REG          0
#define AD_RMT_REG              1
#define AD_RMT_PERSON           2

        char            *ad_recip_number_for_advice;
        char            ad_recip_number_for_advice_crit;
                                                                            90
        OID             ad_phys_rendition_attribs;
        char            ad_phys_rendition_attribs_crit;

        int             ad_pd_report_request;
        char            ad_pd_report_request_crit;
#define AD_PRR_UNSPECIFIED -1
#define AD_PRR_UNDELIV_PDS      0
#define AD_NTF_PDS              1
```

The routine `adr_free` frees off all the allocated elements within a single address structure, but does not free the structure itself.

```
void adr_free (ap)
ADDR *ap;
```

```
typedef struct ad_redirection {
        struct ad_redirection *rd_next;
        char    *rd_addr;           /* O/R Name */
        char    *rd_dn;             /* Distinguished Name */
        int     rd_reason;
#define RDR_RECIP_ASSIGNED      0
#define RDR_ORIG_ASSIGNED       1
#define RDR_MD_ASSIGNED         2
} Redirection;

#define AD_ORIGINATOR           1
#define AD_RECIPIENT            2

typedef struct      adr_struct {
        int             ad_no;          /* recipient number */
        int             ad_status;      /* recipient status */
#define AD_STAT_UNKNOWN         0
#define AD_STAT_PEND            1
#define AD_STAT_DRREQUIRED      2       /* DR required */
#define AD_STAT_DRWRITTEN       3       /* DR written */
#define AD_STAT_DONE            4
        int             ad_rcnt;        /* reformatters done count */

        /* -- MTS Service Parms */

        char            *ad_value;      /* address-original */
        char            *ad_dn;         /* Directory Distinguished Name */
                                        /* QUIPU string encoding */

        int             ad_usrreq;      /* user-report-request bit */
#define AD_USR_NOREPORT         0
#define AD_USR_BASIC            1
#define AD_USR_CONFIRM          2
#define AD_USR_NONE             3
        int             ad_explicitconversion;      /* explicit conversion */

        int             ad_type;        /* address-type */
#define AD_X400_TYPE            1
#define AD_822_TYPE             2
#define  AD_ANY_TYPE            3

        char            *ad_orig_req_alt; /* originator requested alternate */
                                        /* recipient - std encoded */
        char            ad_orig_req_alt_crit;

#define AD_RDM_MAX      4
```

The Address structure

The address structure contains all the information about each address. The structure is defined in the file *adr.h*. The same structure is used for both Recipient and Originator address. The structure is shown in Figure 2.5. There are a number of support routines that help construct addresses.

The routine `adr_new` creates a new address structure with some of the values filled in. This structure is the minimum necessary set of values to submit a recipient. To create an Originator, the responsibility bit `ad_resp` should be set to false, and the status (`ad_stat`) should be set to `AD_STAT_DONE`. The routine is defined as:-

```
ADDR *adr_new (str, type, rno)
char *str;
int type;
int rno;
```

The arguments to this call are

`str`: The address in string format, either RFC-822 or the string form of the O/R name.

`type`: This is the type of address, it should be one of the values suitable for `ad\_type`.

`rno`: The recipient number. Set this to 0, if you don't care about this value.

The routine `ad_init` will initialise an `ADDR` structure to its default values. It is defined as:-

```
void adr_init (ap)
ADDR *ap;
```

The routine `adr_add`, adds an address onto the end of a linked list of addresses, or allocates the first node if the list is empty.

```
void adr_add (base, new)
ADDR **base;
ADDR *new;
```

The routine `adr_tfree` frees a list of addresses freeing of all allocated elements and the structures themselves.

```
void adr_tfree (ap)
ADDR *ap;
```

`ua_id:` The user agent ID. This is a string that identifies the message to a user agent.

`pp_content_correlator:` The internal version of the content correlator.

`general_content_correlator:` The content correlator of the message.

`originator_return_address:` The postal return address of the originator.

`forwarding_request:` An indication of whether the physical forwarding address should be returned to the originator.

`originator_certificate:` This element conveys the certificate of the originator when secuure messaging is being used.

`algorithm_identifier:` An indication of the encryption algorithm used for message encryption.

`message_origin_auth_check:` This element allows MTA's and recipients to authenticate the message origin.

`security_label:` A label indicating the security policy.

`proof_of_submission:` An indication of whether proof of submission is required.

`per_message_extensions:` A holder for any additional elements of the message.

`Oaddress:` Should be empty for submission.

`Raddress:` Should be empty for submission

`inbound:` The inbound channel/mta pair This is important as this controls authentication and authorisation.

`msgid:` The MTS identifier of this message (P1 level).

`trace:` The trace fields.

`dl_expansion_history:` A trace of the distribution lists this message has been expanded by.

The following variables can all be safely defaulted to those initialised by `q_init`, but can be varied as required.

`defertime:` The time the message should wait for before being delivered (for deferred delivery) – optional.

`latest_time:` The latest delivery time this message should be delivered by.

`nwarns:` The number of warning messages to be sent if the message is delayed.

`warninterval:` The interval of time between successive warnings.

`retinterval:` The time to wait before returning the message as undeliverable.

`priority:` The priority associated with the message. This takes one of the following values.

Value	Meaning
`PRIO_NORMAL`	The default
`PRIO_NONURGENT`	A low priority message
`PRIO_URGENT`	A high priority message

`disclose_recips:` A true/false flag indicating whether recipients should be disclosed.

`implicit_conversion:` A true/false flag indicating if implicit conversion can be performed.

`alternate_recip_allowed:` A true/false flag indicating if the message can be sent to an alternate recipient.

`content_return_request:` A true/false flag indicating if the message content should be returned on delivery failure.

`recip_reassign_prohibited:` A flag indicating if the recipient can reassign the message to a new recipient.

`dl_expansion_prohibited:` A flag indicating if the message can be expanded into a distribution list.

`conversion_with_loss_prohibited:` A flag indicating if conversion is allowed even if it looses information.

of the following

Value	Meaning
MT_UMPDU	Normal message
MT_DMPDU	A delivery report
MT_PMPDU	A Probe

The q_init routine defaults this value to MT_UMPDU.

cont_type: The content type of the message. This should be one of the defined types. Normally one of

Value	Meaning
"rfc-822"	An RFC-822 content
"p2"	A P2 1984 content
"p22"	A P2 1988 content

As these values are common, they are available via the following string variables.

```
extern char *cont_822;
extern char *cont_p2;
extern char *cont_p22;
```

An empty content implies that the message will be submitted in structured form (usually the default for RFC-822 messages).

encodedinfo: A list of encoded information types. This should include all the encoded types that are in the message, and should be a memeber of the list defined in the bodypart tailor variable. Although any encoded information type present in the bodypart list is allowed, PP knows about some special cases. These are:

Value	Meaning
"ia5"	An Ia5 (ascii) body part
"hdr.822"	An RFC-822 header
"hdr.p2"	A P2 1984 header
"hdr.p22"	A P2 1988 header
"hdr.ipn"	An IPN header

These definitions are available in the external variables as follows:

```
extern char *hdr_822_bp;
extern char *hdr_p2_bp;
extern char *hdr_p22_bp;
extern char *ia5_bp;
```

Value	Meaning
CRITICAL_NONE	Not critical
CRITICAL_SUBMISSION	Critical for submission
CRITICAL_TRANSFER	Critical for Transfer
CRITICAL_DELIVERY	Critical for Delivery

Table 2.1: Criticality codes

```
/* -- MTA AS Parameters */
        LIST_RCHAN      *inbound;           /* Inbound MTA/Channel */
        MPDUid          msgid;              /* Message ID */
        Trace           *trace;             /* Trace info */
        DLHistory       *dl_expansion_history;                          80
        char            dl_expansion_history_crit;

/* -- PP calculated parameters */
        UTC             queuetime;          /* Time that message was enqueued */
        UTC             departime;          /* Time that message left the queue */
} Q_struct;
```

Figure 2.4: Queue Structure

The queue structure controls most of the per message parameters. There are a number of utility routines to help manipulate these structures.

The routine `q_init` can be used initialise an empty structure. The routine `q_free` can be used to free off allocated fields within the structure.

```
int q_init (qp)
Q_struct *qp;

int q_free (qp)
Q_struct *qp;
```

The elements of the queue structure are now described. Some elements have an associated flag with the name ending with `crit`. This refers to an element in the structure with a similar name. This element indicates what the criticallity constraints on the element are. Values for the settings of these are shown in table 2.1. These values may be logically or'd together to set several critical values.

`msgtype:` The type of message being submitted. This should be one

```
#define PRIO_URGENT         2

        char            disclose_recips;
        char            implicit_conversion;
        char            alternate_recip_allowed;
        char            content_return_request;
                                                                        30
        char            recip_reassign_prohibited;
        char            recip_reassign_crit;

        char            dl_expansion_prohibited;
        char            dl_expansion_crit;

        char            conversion_with_loss_prohibited;
        char            conversion_with_loss_crit;

        char            *ua_id;             /* UA content id */         40
        char            *pp_content_correlator;
                                            /* PP preferred string form  */
        struct qbuf     *general_content_correlator;
        char            content_correlator_crit;
                                            /* ASN.1 uninterpreted */

        FullName        *originator_return_address;
        char            originator_return_address_crit;

        int             forwarding_request;                             50
        char            forwarding_request_crit;

        struct qbuf     *originator_certificate;
        char            originator_certificate_crit;

        struct qbuf     *algorithm_identifier;
        char            algorithm_identifier_crit;
        /* content confidentiality */

        struct qbuf     *message_origin_auth_check;                     60
                        /* also for Probe */
        char            message_origin_auth_check_crit;

        struct qbuf     *security_label;
        char            security_label_crit;

        int             proof_of_submission_request;        /* boolean */
        char            proof_of_submission_crit;

        Extension       *per_message_extensions;                        70


        ADDR            *Oaddress;          /* Orig addr - linked list */
        ADDR            *Raddress;          /* Recip addrs - linked list */
```

`prm_logfile:` A logfile to be used for this specific message.

`prm_loglevel:` A logging level to be used for this message. It should be one of the levels of logging specified in *ll_log.h*.

`prm_opts:` A set of options to control *submit*. This should be one of the following:

Value	Meaning
`PRM_NONE`	No options
`PRM_ACCEPTALL`	*submit* should accept bad recipient addresses and generate error reports itself
`PRM_NOTRACE`	*submit* should not generate Received: or Via: lines.

`prm_passwd:` A password to authenticate the local user's submission (not used as yet).

The Per Message Information

The per message information is defined in the file *q.h* and the structure is shown in Figure 2.4. Only the fields which should or may be filled in by interface programs are described in detail. The rest are calculated by *submit* if necessary.

```
typedef struct Qstruct {
        int             msgtype;        /* Type of message */
#define MT_UMPDU        1
#define MT_DMPDU        2
#define MT_PMPDU        3

/* -- MTS Parameters */
        long            msgsize;        /* Size of message */
        UTC             defertime;      /* Deffered time */
        UTC             latest_time;                            10
        char            latest_time_crit;

        char            nwarns;         /* Number of warning msgs to send */
        int             warninterval;   /* Interval between warning msgs */
        int             retinterval;    /* Interval after which return mail */
        char            *cont_type;     /* Content type of message */
#define  CONT_822S      "822"     /* 822 default type */
#define CONT_P2S        "p2"      /* p2 default string */

        EncodedIT       encodedinfo;    /* Encoded information type20*/
        int             priority;       /* Priority of message */
#define PRIO_NORMAL     0
#define PRIO_NONURGENT      1
```

A new element may be added to the end of a list, or become the first member if the list is empty, by calling `list_bpt_add`. This is defined as

```
void list_bpt_add (base, new)
LIST_BPT **base;
LIST_BPT *new;
```

A list may be searched for a particular type of element by using `list_bpt_find`. This returns either the first macthing entry or `NULLIST_BPT` if no such element is found.

```
LIST_BPT *list_bpt_find (list, item)
LIST_BPT *list;
char *item;
```

Finally, a linked list may be deallocated up by calling the routine `list_bpt_free`.

```
void list_bpt_free (list)
LIST_BPT *list;
```

Management Parameters

The management parameters are defined in the file *prm.h* and the structure is shown in Figure 2.3. These values are set before each message submission to change the general policies of *submit*.

There are two support routines for manipulating this structure. The first, `prm_init`, initialises the structure to its default state. The second, `prm_free`, free's off all allocated variables within the structure.

```
void prm_init (pp)
struct prm_vars *pp;

void prm_free (pp)
struct prm_vars *pp;
```

```
struct  prm_vars {
        char    *prm_logfile;
        int     prm_loglevel;
        int     prm_opts;
        char        *prm_passwd;
};
```

Figure 2.3: Management Parameter Structure

The use of each of the members of this structure are:

`mta:` The name of the mta. This parameter is copied intop heap storage.

`chan:` The name of a channel. This is converted to a channel structure internally.

A new structure can be added onto the end of a list, or if the list is empty can be set as the first element by a call to `list_rchan_add`. This is defined as

```
void list_rchan_add (base, new)
LIST_RCHAN **base;
LIST_RCHAN *new;
```

The linked list structure can be freed by calling `list_rchan_free`, which is defined as

```
void list_rchan_free (list)
LIST_RCHAN *list;
```

Encoded Information Types

Encoded information types are used in several places in PP. They are kept internally as a linked list of strings which contain either the name of the information type, or a string representation of an object identifier. The structure is shown in Figure 2.2.

```
typedef struct list_bpt {
        char                *li_name;
        struct list_bpt      *li_next;
} LIST_BPT;
```

Figure 2.2: Encoded Information Type Structure

There are some support routines to help construct these lists. A new structure can be allocated with a value stored in help space by using the function `list_bpt_new`. This is defined as

```
LIST_BPT *list_bpt_new (value)
char *value;
```

A linked list structure can be duplicated by using `list_bpt_dup`. This returns a separate duplicate list. It is defined as:

```
LIST_BPT *list_bpt_dup (list)
LIST_BPT *list;
```

2.2 The io_ Interface

The `io_` interface is the full interface to PP. All routines have the prefix `io_` and all return some type of status indicating success or failure.

2.2.1 Major Structures

The following are the key structures used in submission. These structures should be filled in and passed over to *submit* by the procedures defined below.

Channel Lists

A channel list is the structure used to keep channels and hosts. It is defined in file *list_rchan.h* and the structure is shown in Figure 2.1

```
typedef struct list_rchan {
        char                *li_mta;
        CHAN                  *li_chan;
        AUTH                  *li_auth;
        struct list_rchan     *li_next;
} LIST_RCHAN;
#define NULLIST_RCHAN          ((LIST_RCHAN *)0)
```

Figure 2.1: Channel List structure

The use of the structure memebers are:

`li_mta:` The MTA name as a character string.

`li_chan:` The associated channel structure.

`li_auth:` Authorisation paramaters. This should not normally be filled in.

`li_next:` The next element in the list.

There are several routines to manipulate such structures.

A new list element can be created by using the routine `list_rchan_new`. This is defined as:-

```
LIST_RCHAN *list_rchan_new (mta, chan)
char *mta;
char *chan;
```

The parameters to this call are

Chapter 2

Message Submission

2.1 Overview

This chapter describes the procedural interface for message submission. This interface reflects the submission interface of both the Message Transfer Abstract Service and the MTA Abstract Service. It is therefore used for:

- User interfaces submitting messages (Message Transfer Abstract Service).
- Inbound channels submitting messages (MTA Abstract Service).

It is a generalised interface, with aspects beyond X.400, but it is broadly targeted for X.400 usage. The general function of the interface is as follows:

1. Initialise submission
2. Initialise Message (per-message parameters)
3. Handshake on addresses
4. Transfer one or more body parts
5. Final handshake to lock message into the queue or abort
6. Goto 2, until no more messages
7. Terminate submission

`util/`: This directory contains some generally useful common procedures.

1.5.2 Core Programs

The next major directory to be considered is the `Src/` directory. This contains the core programs required by PP. These are:

`MTAconsole/`: This is the source code for the X based MTA console program. It monitors the overall running of PP.

`submit/`: This is the source for the submission process. All messages entering the queue go through this process.

`qmgr/`: This is the source for the `qmgr` process.

`pptsapd/`: This is the source for the pp tsap daemon. This is the process that runs the channels invoked by the queue manager.

1.5.3 Other directories

The are the following other directories which have some significance.

`h/`: Contains all the public header files used in the PP system.

`doc/`: Contains the documentation on PP.

`man/`: Contains manual pages on the PP programs.

`Uip/`: Contains some user interface programs for submitting messages to PP.

1.5.4 Tools and Utilities

The `Tools/` directory contains several programs useful for building up parts of the PP system or verifying it.

1.5.5 Channels

The channel programs can all be found under the directory `Chans/`.

Non ASN.1 protocol exchanges can be logged with the `PP_LOG` macro at the `LLOG_PDUS` level.

If other logging calls not described here are needed, they can be constructed from the more general `SLOG` macro described in the ISODE manual. Use of this feature is discouraged.

1.5 Code location

The code to PP is arranged in a hierarchy of directories. This is a brief tour around the structure.

1.5.1 Library code

Much of the PP system is in the library code. This is found in the directory `Lib/` under the top directory of the PP distribution. The library itself is split into several subdirectories. These are as follows:

`addr/`: The RFC-822 address parser library. This contains ways to manipulate and process RFC-822 addresses.

`format/`: This directory contains support routines for formatting messages.

`io/`: This directory contains most of the routines concerned with writing out and reading in structures. It is generally used for reading in structures and writing them out.

`or/`: This directory has a set of routines for manipulating O/R-address.

`parse/`: This directory contains code to do the generic parsing of addresses. It contains the general interface to X.400 adn RFC-822 addresses.

`pp/`: This directory contains many of the basic PP specific routines such as manipulation of structures PP structures and the flattening and unflattening of these structures.

`qmgr/`: This directory contains the code required to interface with the `qmgr` process.

`table/`: This directory contains routines for accessing the databases used by PP for routing, aliases and similar.

`tai/`: This contains the routines responsible for reading and parsing the tailor file.

This macro is used to record interesting events in protocol exchanges. It should be used so that the basic flow of a program may be observed, without undue amounts of log file space or extra machine load. Common logging events should be connecting to a host; terminating a connection; the transfer of a message.

```
PP_TRACE ((format, arg1, arg2 ...))
char   *format;
```

This macro is used throughout the source code to trace the calling of functions and the general flow of the programs and libraries. It logs the information at the `LLOG_TRACE` level, and this macro is dependent on the value of the `PP_DEBUG` define. If this is not defined then the `PP_TRACE` macros are not compiled in. The arguments are identical to the `PP_STAT` macro.

```
PP_DBG ((format, arg1, arg2 ...))
char   *format;
```

This macro is similar to the `PP_TRACE` macro, except it logs at the level of `LLOG_DEBUG` and will not expand to anything unless `PP_DEBUG` is defined and has a value greater than 1 (usually `PP_DEBUG_ALL`). This macro should be used for very detailed tracing, or where large amounts of logging are likely to be generated.

```
PP_PDU ((fnx, pe, text, rw))
IFP    fnx;
PE     pe;
char   *text;
int    rw;
```

This macro is for logging of ASN.1 protocol data units. It is only compiled in if `PP_DEBUG` is `PP_DEBUG_SOME` or greater. The arguments are as follows:

`fnx`: A function to print out the pdu. This should be a function constructed by the printer option of pepy.

`pe`: The presentation element to print out.

`str`: An explanatory string, usually the name of the pdu.

`rw`: A flag indicating the direction the pdu is going. Set to `PDU_WRITE` to indicate the pdu is being written and `PDU_READ` to show it is being read.

```
char *what;
char *format;
```

This is similar to the PP_LOG macro, but the extra argument ensures that operating system call failures are logged. This macro should be used where it is suspected that a system call has failed.

level: The logging level.

what: A string which if not the null pointer logs the system error.

format: A printf like format string.

For more urgent cases of logging, the following macro is provided.

```
PP_OPER (what, (format, arg1, arg2 ...))
char   *what, *format;
```

This macro is used for logging of extreme problems. It should only be used for errors that require operator intervention. These errors typically end up in a separate log as well as the normal log and are always logged at the LLOG_FATAL level. It is assumed that the system administrator will look through this log frequently and will immediately take action on anything found there. The arguments are:

what: This is a string which is used to construct a system error message. This should only be present if a system call is suspected of failing. Otherwise the constant NULLCP should be used.

format: This string is used as a printf like syntax to produce the message together with optional arguments as in the case of PP_LOG. Again, this parameter and any arguments *must* be enclosed in brackets as shown.

Another logging call is provided for statistical and authorisational data.

```
PP_STAT ((format, arg1, arg2 ...))
char   *format;
```

This macro is used for logging statistical information. It should be used where useful to record statistics like number of messages and times etc. Its arguments are printf like, and *must* be enclosed in the double brackets as shown.

```
PP_NOTICE ((format, arg1, arg2 ...))
char   *format
```

LLOG_NONE	Not logged
LLOG_FATAL	Fatal errors
LLOG_EXCEPTIONS	Exceptional events
LLOG_NOTICE	General information
LLOG_PDUS	Printing of Protocol Data Units
LLOG_TRACE	Program Tracing
LLOG_DEBUG	Full debugging
LLOG_ALL	All Values

Table 1.1: Logging Levels

1.4.3 Logging macros

There are several macros defined to aid in the use of logging. The definitions of these macros can be found in the include file *ll_log.h*. These macros may be used in the following way.

```
PP_LOG (level, (format, arg1, arg2 ...))
int    level;
char   *str;
```

This macro is used for most of the non-debugging logging. The arguments to this call are as follows:

`level:` One of the logging levels specified above. However as there are macros provided for fatal errors and debugging, it is usually only the values of `LLOG_EXCEPTIONS` and `LLOG_PDUS` which are used as the argument here. The `LLOG_EXCEPTIONS` is used to log failures that are not critical. The `LLOG_PDUS` level is used to trace any protocol exchanges – using the `PP_PDUS` macro if suitable.

`format:` This is a printf like format string which together with any arguments *must* be enclosed in brackets as shown. The `format` and any number of optional `arg`'s produce the logging message.

The next macro is a more general macro which allows logging of system error messages to be attached to the basic log message.

```
PP_SLOG (level, what, (format, arg1, arg2 ...))
int  level;
```

1.3 Runtime Tailoring

The runtime tailoring for PP is complex, as there are many parameters that can affect the operation of the system. The main things tailored are pathnames to locate commonly used programs; file names and directories; or the more complex tailoring to define such things as channels and tables. See Volume 1 for the full detail.

Too add new items to be tailored, it is necessary to add some code to the parsing routines. For simple additions, it is necessary only to modify the tables and the switch statement in the file *pp/Lib/tai/tai_sys.c*. More complex tailoring is best done my creatinga separate procedure, possible in a new file. Such additional tailoring should not be undertaken lightly as it will make the new version of PP incompatable with existing and new versions.

1.4 Logging

All logging works through a standard package. This package includes features to limit the log files and to notify administrators of errors. The logging has some structure too it and the following are the general rules used when programming PP.

1.4.1 Logging Levels

The logging package takes a bit mask of values to log. This is compared with the current setting of the logging level and then certain values are logged. The levels available are shown in Table 1.1. The values `LLOG_NONE` and `LLOG_ALL` are special – the first is no logging at all, the latter sets all bits in the bitmask.

1.4.2 Logging structures

There are three log structures used in PP. The structures themselves are defined in the include file *isode/logger.h*. These structures are of type `LLog *` and are the following.

Structure	Usage
`log_norm`	Normal logging
`log_oper`	Critical Operator message
`log_stat`	Statistical and Authorisation

These structures are rarely used explicitly. The logging macros described below with few exceptions use the `log_norm` structure.

Chapter 1

General

1.1 Overview

There are certain approaches taken by PP processes. These are described here, to facilitate emulation by new processes. These are not strict guidelines, but following existing practises will usually be easier and make upgrading the software less painful.

1.2 Initialisation

All PP programs generally start by calling one or more initialisation routines. This may vary depending on what the channel needs to know. This initialisation does various things amongst which are

- Reading global parameters from the tailor file.
- Initialising the logging packages.
- Handling of signals.

The normal method of initialising a process is to call the following routine very early on (usually one of the first things).

```
sys_init (name)
char *name;
```

This routine does most of the initialisation required. The argument is usually the value of `argv[0]` which is used in the logging initialisation to record the name of this program.

Preface

This volume describes the PP programming interface. This covers:

- submission of messages such as might be used by a User agent.
- submission of message as by a protocol channel
- writing channel programs.

It also explains the major data structures used in PP and the conventions such as logging and initialisation.

List of Figures

2.1 Channel List structure 9
2.2 Encoded Information Type Structure 10
2.3 Management Parameter Structure 11
2.4 Queue Structure 13
2.5 Address Structure 20
2.6 Reply Structure 23
2.7 Example interaction with submit 28

3.1 Channel / Queue Manager Interface Structures 33
3.2 Example outbound channel 44
3.3 Example Delivery Report Generation 45

List of Tables

1.1 Logging Levels 2

2.1 Criticality codes 14
2.2 Table of Explicit Conversion values 21
2.3 Return codes 24

3.1 Reason codes 39
3.2 Diagnostic codes 40

3.2.1 Major Structures 32
3.2.2 Basic Access 34
3.2.3 Delivery Notifications 39
3.2.4 Verification and Trust 41
3.3 Example usage 42
3.3.1 Outbound Channels 42
3.3.2 Local Pickup 44
3.3.3 TWA channel 44
3.3.4 Delivery Reports 44
3.4 Formatters and Converters 45
3.4.1 Simple converters 45
3.4.2 Complex Filters 45
3.4.3 Example 45

Contents

1 General **1**

1.1 Overview 1
1.2 Initialisation 1
1.3 Runtime Tailoring 1
1.4 Logging 2
1.4.1 Logging Levels 2
1.4.2 Logging structures 2
1.4.3 Logging macros 3
1.5 Code location 5
1.5.1 Library code 5
1.5.2 Core Programs 6
1.5.3 Other directories 6
1.5.4 Tools and Utilities 6
1.5.5 Channels 7

2 Message Submission **8**

2.1 Overview 8
2.2 The io_ Interface 8
2.2.1 Major Structures 8
2.2.2 Reply value structure 23
2.2.3 Procedure Calls 23
2.2.4 Example 27
2.3 The RFC 822 interface 28
2.3.1 Initialisation 28
2.3.2 Addresses and Headers 29
2.3.3 Text Submission 30
2.3.4 Termination 31
2.4 Debugging 31

3 Channels **32**

3.1 Overview 32
3.2 The Queue Interface 32

PP

Volume 2: Programmer's Guide

Julian Onions
Nottingham University

March 18, 1991