ISODE CONSORTIUM

ISODE Volume 15

Programmer's Guide

X.400 UA

This documentation and the software are based on contributions from multiple sources. Acknowledgements for contribution are given in *ISODE Volume 1: Overview*.

This manual is associated with Version 2.0 of March 1994, and was last updated in March 1994.

Contents

1 Use of Processes **1**

1.1 Overview 1

1.2 Initialisation 1

1.3 Runtime Tailoring 2

1.4 Logging 2

1.4.1 Logging Levels 2

1.4.2 Logging Structures 3

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 7

2 Message Submission **8**

2.1 Overview 8

2.2 The io_ Interface 8

2.2.1 Major Structures 9

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 28

2.3.3 Text Submission 29

2.3.4 Termination 30

2.4 Debugging 30

List of Tables

1.1 Logging Levels 3

2.1 Table of Explicit Conversion Values 21
2.2 Return Codes 24

List of Figures

2.1 Queue Structure 14
2.2 Address Structure 21
2.3 Example Interaction with Submit 27

Preface

This guide is published in support of the ISODE Consortium Release 1 (IC R2). It is intended for programmers concerned with X.400 User Agent functions in an OSI environment.

It is assumed that readers of this guide are already aware of the contents of *ISODE Volume 1: Overview* and *ISODE Volume 9: Programmer's Guide: General Services*. Other guides to other IC R2 products and their uses are listed under Related Publications later in this Preface.

What this Guide Covers

Chapter 1 explores the conventions used in the programming interface applicable to the PP message handling module of IC R2. These include recommended approaches to initialisation, logging and code location

Chapter 2 describes the procedural interface for incoming messages in the PP message handling module of IC R2. This interface (known as the *io_interface*) is common to messages submitted by User Agents (message transfer abstract service) and messages entering through inbound channels from peer MTAs (MTA abstract service). Details are given of all the structures and parameters used to define and handle messages of various types in accordance with X.400.

Software and Documentation Versions

IC R2 is the second major release of ISODE software made by the ISODE Consortium. It is derived, with revisions and additions, from the openly available ISODE 8.0 release.

The release is accompanied by a comprehensive documentation set comprising the following volumes:

Volume	Title
01	*ISODE Volume 1: Overview*
02	*ISODE Volume 2: Administrator's Guide: Installation*
03	*ISODE Volume 3: Administrator's Guide: General Services*
04	*ISODE Volume 4: Administrator's Guide: Message Handling Services*
05	*ISODE Volume 5: Administrator's Guide: Directory Services*
07	*ISODE Volume 7: User's Guide: Message Handling Services*
08	*ISODE Volume 8: User's Guide: Directory Services*
09	*ISODE Volume 9: Programmer's Guide: General Services*
10	*ISODE Volume 10: Programmer's Guide: ASEs*
12	*ISODE Volume 12: Programmer's Guide: X.500 DUA*
13	*ISODE Volume 13: Programmer's Guide: Layer Interfaces*
14	*ISODE Volume 14: Programmer's Guide: X.400 MTA*
15	*ISODE Volume 15: Programmer's Guide: X.400 UA*
16	*ISODE Volume 16: Programmer's Guide: Transport Services*
17	*ISODE Volume 17: Programmer's Guide: ASN.1 Library*
18	*ISODE Volume 18: Programmer's Guide: ASN.1 Tools*

IC R2 is released as a number of separate modules. Each module has its own update number which is incremented independently from other modules whenever an update is made to it. The version numbers of the various components may therefore differ from one another.

As of March 1994, the latest versions of the modules and their associated documentation are as follows:

Module	Version	Volume	Last Document Update
IC Base	R2.0	01	June 1993
		02	March 1994
		03	March 1994
		09	February 1994
		10	February 1994
		13	February 1994
		16	February 1994
		17	February 1993
		18	February 1993
IC DSA	R2.0	05	March 1994
IC DUA	R2.0	08	February 1993
		12	March 1994
IC MTA	R2.0	04	March 1994
	R2.0	07	February 1994
		14	February 1993
		15	February 1993
IC SNMP	R2.0	03	March 1994
Contributed	R2.0	-	-

The ISODE Consortium

The ISODE Consortium is a not-for-profit cooperative enterprise. Its mission is to promote and evolve the ISODE implementation of OSI. This is a major package of OSI applications, including source code and documentation, with a worldwide record of operational success.

All individuals and organisations handling ISODE source code are encouraged to join the ISODE Consortium. For more information about the ISODE Consortium, contact:

Headquarters

Address:	ISODE Consortium The Dome The Square Richmond TW9 1DT UK	Phone: Fax:	+44 81 332 9091 +44 81 332 9019

Email

Internet (RFC822)ic-info@isode.com

MHS (X.400): S=ic-info; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI;

If this is not possible, you may also reach the ISODE Consortium at:

US Office

Address:	ISODE Consortium PO Box 200195 Austin TX 78720 USA	Phone: Fax:	+1 (512) 338 3340 +1 (512) 338 3600

Availability

IC R2 is a source release, and is available in the following ways:

1. To organisational members of the ISODE Consortium as a benefit of membership.
2. To educational and non-commercial research organisations, under zero-cost licence from the ISODE Consortium or under licence from a Research Consortium member of the ISODE Consortium.
3. To any organisation as a 30 day evaluation copy from the ISODE Consortium.

Acquisition and use of IC R2 and related materials for any purpose requires a written licence agreement from ISODE Consortium Limited, or a written licence from an organisation licenced by ISODE Consortium Limited to grant such a licence.

Binary products based on IC R2 are available on a wide range of platforms from ISODE Consortium members. Information on these products and where to obtain them is available from the ISODE Consortium.

Bug Reporting

Bug reports on software releases made by the ISODE Consortium are welcomed. These may be sent to the Consortium by any means, but electronic mail to one of the bug reporting addresses is preferred. Please send proposed fixes with the reports if possible. For non-members of the ISODE Consortium, any reports will be acknowledged, but no further action is guaranteed. Any changes resulting from bug reports may be included in future releases.

The bug reporting addresses in Internet (RFC 822) format are:

- `bug-isode@isode.com` for general ISODE reports.
- `bug-quipu@isode.com` for reports on QUIPU (Directory).
- `bug-pp@isode.com` for reports on PP (MHS).
- `bug-snmp@isode.com` for reports on SNMP.
- `bug-doc@isode.com` for reports on the documentation.

The MHS (X.400) equivalents of these addresses are:

- `S=bug-isode; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-quipu; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-pp; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-snmp; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-doc; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`

Support

Basic support of ISODE Consortium releases is available to all consortium members as a benefit of membership. This includes upgrades and access to a database of bug reports and fixes. The ISODE Consortium also offers some additional support to companies building products specifically based on ISODE Consortium releases.

A range of ISODE-related support services are offered by ISODE Consortium members. A list of such services will also be available from the ISODE Consortium.

Chapter 1

Use of Processes

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 formal guidelines, but following existing practices 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 PP *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.

For some programs however, this sort of initialisation is not appropriate. In this case another interface may be better.

```
int pp_initialise (name, flags)
char *name;
int flags;
```

This routine will read in the PP *tailor* file, but will optionally not configure certain aspects depending on the flags. The name is the same as in `sys_init`. The flags can be any of the following:

Flag	Effect
TAI_NONE	No special tailoring is done.
TAI_LOGS	Reset all logs to use the PP log directory.
TAI_SIGNALS	Set handlers for certain signals.
TAI_ALL	All of the above.

The flags may be combined using the bitwise or operator.

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.

To add new items to be tailored, it is necessary to add some code to the parsing routines. In the simplest cases, it is only necessary to modify the tables and the switch statement in the file *pp/Lib/tai/tai_sys.c*. More complex tailoring is best done by creating a separate procedure, possibly in a new file. Such additional tailoring should not be undertaken lightly, as it will make the new version of PP incompatible 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 to 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 (as shown in Table 1.4.3). 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.

Setting	Logging level
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

1.4.2 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. With few exceptions, the logging macros described below use the `log_norm` structure.

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 parentheses, as shown. The `format` and any number of optional `arg` arguments produce the logging message.

The next 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;
char *what;
char *format;
```

This is similar to the `PP_LOG` macro, but it ensures that if the extra argument (`what`) is not `NULL`, system call failures are logged as well. This macro should be used where it is suspected that a system call has failed; for example: `PP\_SLOG (LLOG\_EXCEPTIONS, "memory", ("out of"));`

`level`: The logging level.

`what`: A character string indicating which system call failed, or the NULL pointer.

`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;
char   *format;
```

This macro is used for logging of extreme problems, particularly those 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 to the PP_OPER call are:

`what`: This is a string which is used to construct a system error message. The `what` argument is always present as PP_OPER () is not valid. A `what` value 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 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
```

This macro is used to record interesting events in protocol exchanges. It should be used for debugging 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 include: connecting to a host, terminating a connection, and the transfer of a message.

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

This macro is used throughout the source code to trace function calls and the general flow of the programs and libraries. It logs the information at the `LLOG_TRACE` level, and this macro is dependent on `PP_DEBUG` being defined. 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 (PDUs). 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 *pepsy*.

`pe`: The presentation element to be printed 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.

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

The logging package should provide enough of a consustent interface. Hwoever, if there is a need for logging calls not described here, they can be constructed from the more general `SLOG` macro.

1.5 Code Location

The PP code 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 routines. This directory contains resources for manipulating and processing RFC 822 addresses.

`charset/`: This directory contains support for converting between different character sets.

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

`io/`: This directory contains the majority of the routines concerned with writing out and reading in structures.

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

`parse/`: This directory contains the code to do generic address parsing. It contains the general interface to X.400 and RFC 822 address parsers.

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

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

`submisson/`: This directory contains routines for handling submissions, including those required to do the various submission tests.

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

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

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

`x400/`: This directory contains support routines for X.400 handling.

`x500/`: This directory contains support routines for interaction with the X.500 Directory.

1.5.2 Core Programs

The next major directory to be considered is the `Src/` directory which 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.

`lconsole/`: The source for a line based console.

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

`qad/`: This is the source for the QMGR assistance daemon.

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

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

1.5.3 Other Directories

The contents of the other significant directories are:

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

`doc/`: The documentation on PP.

`man/`: Manual pages on the PP programs.

`Uip/`: User interface programs, including those required for submitting messages to PP.

`Tools/`: Several programs useful for building up or verifying parts of the PP system.

`Chans/`: All the channel programs.

`Format/`: The reformatter programs.

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 relevant abstract services, namely:

- 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 commit or abort message.
6. Goto 2, unless there are no more messages.
7. Terminate submission.

2.2 The io_ Interface

The `io_` routines provide the full submission interface to PP. All routines have the prefix `io_` and all return a status value 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 to the *submit* process by the procedures defined below.

Channel Lists

A channel list is the structure used to record channels and hosts. The structure is defined in file *list_rchan.h* as follows:

```
typedef struct list\_rchan \{
    char                *li\_mta;      \/* the MTA *\/
    CHAN                *li\_chan;     \/* channel used to reach the MTA *\/
    union \{
            AUTH        *li\_un\_auth; \/* authentication used in submit *\/
            char        *li\_un\_dir;  \/* the associated directory *\/
    \} list\_rchan\_un;
    struct list\_rchan  *li\_next;     \/* next in the chain *\/
\} LIST\_RCHAN;
\#define   li\_auth   list\_rchan\_un.li\_un\_auth
\#define   li\_dir    list\_rchan\_un.li\_un\_dir
```

The use of the structure members is:

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

`li_chan`: The associated channel structure.

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

`li_dir`: A directory reference. This should not normally be used.

`li_next`: The next element in the list.

There are several routines to manipulate this structure.
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:

`mta`: The name of the MTA. This parameter is copied into new 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;
```

Two other useful routines convert a string into a channel structure. The first is a simple "string to structure" routine:

```
CHAN *ch_nm2struct (name)
char *name;
```

This converts from the given `name` into a channel structure, or returns `NULLCHAN` if there is no match.

For inbound channels, when filling in the queue structure with inbound parameters, it is necessary to allocate a channel list without filling in the channel part of it. This should be filled in with the second routine:

```
CHAN *ch_mta2struct (name, mta)
char *name, *mta;
```

This routine returns a channel structure based on a name and an MTA. It allows the particular channel to be selected dependent upon the MTA the message is received from.

Encoded Information Types (EITs)

EITs 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:

```
typedef struct list\_bpt \{
     char                   *li\_name;
     struct list\_bpt       *li\_next;
\} LIST\_BPT;
```

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

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

The new structure contains an allocated copy of `value`.

A linked list structure can be duplicated by using `list_bpt_dup`. This returns a list which can be modified without affecting the list passed as the argument. It is defined as:

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

A new element may be added to 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 an element holding a specific encoded information type by using `list_bpt_find`. This returns either the first matching 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 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 shown in this section. Values should be set to define the general policies of *submit* before any message is submitted. Once set, the values would not normally need to be changed.

```
struct prm_vars \{
    char   *prm\_logfile;
    int    prm\_;loglevel
    int    prm\_;opts
    char   *prm\_;passwd
\};
```

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

`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).

If changes are necessary, there are two support routines for manipulating this structure. The first, `prm_init`, initialises the structure to its default state. The second, `prm_free`, frees all allocated variables within the structure.

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

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

The Per-Message Information

The per-message information is defined in the file *q.h* and the structure is shown in Figure 2.1. Only the fields which should (or may) be filled in by interface programs are described in detail. Others are calculated by *submit*.

```
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;
        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 */
        EncodedIT       orig_encodedinfo; /* Original Encoded Info Types */
        int             priority;       /* Priority of message */
#define PRIO_NORMAL     0
#define PRIO_NONURGENT  1
#define PRIO_URGENT     2

        char            disclose_recips;
        char            implicit_conversion_prohibited;
        char            alternate_recip_allowed;
        char            content_return_request;

        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 */
        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;
        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;
                                /* 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;

        X400_Extension      *per_message_extensions;

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

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

/* —— PP calculated parameters */
        EncodedIT           encodedinfo;        /* Encoded information types */
        UTC                 queuetime;          /* Time that message was enqueued */
        UTC                 departime;          /* Time that message left the queue */
        int                 n_bodyparts;
        int                 n_forwarded;

        /* —— Queue Control File offsets for fixed length variables —— */
        off_t               nwarns_offset;      /* offset param for nwarns */
} Q_struct;
```

Figure 2.1: Queue Structure

The queue structure controls most of the per message parameters. There are a number of utility routines to help in manipulating this structure.

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

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

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

Typically an inbound channel will only fill in one or two parameters. Most can be safely defaulted unless the channel requires special processing for the message. It is important to set the `inbound` parameter and to fill in the list of encoded information types and possibly the content type. The rest are all optional.

The elements of the queue structure are now described. Some elements have an associated flag, with a name ending in `crit`, that indicates its criticality constraints. Values for these are:

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

These values may be logically chained (using "or") to set a combination of criticality constants.

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

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

As these values are commonly used, they are available in the following string variables.

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

An empty content type 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 member 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 *hdr_ipn_bp;
extern char *ia5_bp;
```

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).

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

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

`warninterval`: The interval of time between these 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 a recipient may reassign the message to a new recipient.

`dl_expansion_prohibited`: A flag indicating if the message may not be expanded by a distribution list.

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

`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 secure messaging is being used.

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

`message_origin_auth_check`: This element allows MTAs 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 it 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 by which this message has been expanded.

The Address structure

The address structure (illustrated in Figure 2.2) contains all the information about each address. The same structure is used for both Recipient and Originator addresses. Its definition is in the file *adr.h*. There are a number of support routines that help construct addresses.

The routine `adr_new` creates a new address structure with the minimum set of necessary values defaulted to permit submission as 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 address.

`type`: 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 do not 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 all of the allocated elements and the structures themselves.

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

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

```
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 */
        UTC       rd_time;
        int       rd_reason;
} Redirection;

typedef struct       adr_struct {
        int                 ad_no;              /* recipient number */
                                                /* PP's key to this adr */

                                                /* Next 3 parms may be modfiied */
                                                /* in Q.  Fixed text encoding */
        int                 ad_status;          /* recipient status */
        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 */
        int                 ad_explicitconversion;        /* explicit conversion */
        int                 ad_type;            /* address-type */

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

#define AD_RDM_MAX          4
        int                 ad_req_del[AD_RDM_MAX];       /* requested delivery method */
        char                ad_req_del_crit;

        char                ad_phys_forward; /* boolean - is physical                    */
                                                /* 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;

        int                 ad_reg_mail_type;
        char                ad_reg_mail_type_crit;
        char                *ad_recip_number_for_advice;
        char                ad_recip_number_for_advice_crit;

        OID                 ad_phys_rendition_attribs;
        char                ad_phys_rendition_attribs_crit;

        int                 ad_pd_report_request;
        char                ad_pd_report_request_crit;
        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;
        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 */
        int                 ad_subtype;         /* address-subtype */


        X400_Extension      *ad_per_recip_ext_list;
                                                /* bucket for new and private */
                                                /* odds and sods */

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

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

        /* -- should only be set for inbound x400 channels -- */
        char                *ad_r400orig;       /* original form of x400 address */

        /* -- next two should only be set for -- */
        /* -- unroutable originator addresses -- */
        char                *ad_r400DR;         /* x400 address for DRs */
        char                *ad_r822DR;         /* 822 address for DRs */

        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 */


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

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

        Aparse              *aparse;            /* parsing bucket */

        /* -- 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;

#define NULLADDR ((ADDR *)0)
```

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 2.1: Table of Explicit Conversion Values

Figure 2.2: Address Structure

Only the relevant 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	recipients awaiting delivery
AD_STAT_DONE	no delivery permitted to these recipients

`ad_dn`: The distinguished name contained in the address.

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

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

`ad_explicitconversion`: Is explicit conversion required? The value should be one of those defined in Table 2.1.

`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 alternatives.

`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. For example:

```
AD_PM_CNT_BUREAU ="bureau"
AD_PM_CNT ="counter-collect"
AD_PM_CNT_PHONE_ ="counter-collect+telephone"
AD_PM_CNT_TTX ="counter-collect+teletex"
AD_PM_CNT_TLX ="counter-collect+telex"
AD_PM_EXP ="express"
AD_PM_ORD ="ordinary"
AD_PM_SPEC ="special"
```

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

```
AD_RTM_NON_REG ="non-registered"
AD_RTM_REG ="registered"
AD_RTM_PERSON ="personal"
```

`ad_recip_number_for_advice`: The number for advice to call for delivery.

`ad_phys_rendition_attribute`: The physical rendition attributes for the recipient indicated by this address if the message is transfered to the postal service.

`ad_pd_report_request`: An option to supply a physical delivery report.

`ad_redirction_history`: The redirection history for this address in the form of a list of addresses with redirection timestamps.

`ad_message_token`: A security parameter for message tokens.

`ad_content_integrity`: A content integrity check parameter.

`ad_proof_delivery`: 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 subtype of the address if appropriate. At present, this is only used for RFC 822 addresses. 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`: The address as originally submitted.

`ad_next`: The link to the next address if this address is in a linked list.

2.2.2 Reply Value Structure

The reply value buffer structure, used in all handshakes with *submit*, to return a status and an explanatory string, is:

```
typedef struct} rp\_bufstruct     \/* For reading reply strings *\/
\{
    unsigned char  rp\_val;        \/* The value returned (one of those
defined in retcode.h) *\/
    char          rp\_line[256];  \/* An optional explanatory string *\/
\} RP\_Buf;
```

2.2.3 Procedure Calls

The submission interface is procedural and handshakes at most of the interaction points. This interaction is reasonably complex but allows the submission process 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 return codes that show status, in detail, are listed in Table 2.2. For determining the overall, situation, two macros are useful. These 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 value is generally “good” or “bad” respectively.

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

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 2.2: Return Codes

Initialisation

To initialise the submission system, it is necessary first 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 status (good or bad, as defined in Section 2.2.3), and fills in the `rp` structure if the response is bad (i.e. the invocation was unsuccessful).

Message Management Parameters

The first set of parameters to negotiate with *submit* are the management parameters. These are defined in the structure on page 11. This structure is transmitted 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 by *submit* 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 sender is passed over in the standardised address structure. This achieved by the `io_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 should be passed over, where it is checked by *submit*. This is 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 types of body parts presented are dependent on the type of message being transferred. To initialise for body part submission, the following function should be called:

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

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;
```

`name` is the name of this part. For an RFC 822 body part, it is usually something like `hdr.822` and `1.ia5` for the two parts of the message. For X.400, it is 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 `name` is actually a secret *link* (2) to another file in the queue. This is only for use by special channels and normally will be rejected by *submit*. If the link is false, then the the body part content is read from `name`.

All data is submitted 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 a temporary file in the queue created by *submit* as a result of 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 submission is brought to an end 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:

`OK`: The submission is successful.

`NOTOK`: The submission should be aborted.

2.2.4 Example

An example interaction with submit is shown in Figure 2.3.

```
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 2.3: 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, however, 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.

All routines return the constant `NOTOK` on failure and `OK` on success. The `NOTOK` is unrecoverable, and the interface must be restarted, from the beginning. All routines also take an `RP_Buf` pointer which is updated if the call fails. This may be used for diagnosis of any problem.

2.3.1 Initialisation

To initialise this interface, a client program calls the `pps_init` routine. This attempts to initialise an 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, or 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* daemon or a *submit* child process.

2.3.2 Addresses and Headers

The `pps_adr`, `pps_to` and `pps_cc` routines are called to supply recipient addresses for the message. `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 a file descriptor (returned by `open()`) 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 by `fopen()`). This is read up to the end of file.

These calls may be freely intermixed.

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 up 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, therefore, be run routinely with full logging.

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` macro 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 (for example, printing individual variables during different stages of program execution). Library routines, in particular, use this macro to cause logging of such detailed traces.

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.