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

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

This is similar to the PP_LOG macro, but the extra argument if not null ensures that operating system call failures are logged as well. 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
```

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, and 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 (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 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.

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 addresses.

`parse/`: This directory contains the code to do the generic parsing of addresses. It contains the general interface to X.400 and 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 the like.

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

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

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

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.

`LINEconsole/`: The source for a line based console program.

`lconsole/`: The source for an alternative line based console.

`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.

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

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

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.

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; /* the MTA */
        CHAN                  *li_chan; /* the channel its on */
        union {
                AUTH            *li_un_auth; /* auth stuff used in submit */
                char              *li_un_dir; /* the directory associated */
        } list_rchan_un;
        struct list_rchan      *li_next; /* next in the chain */
} LIST_RCHAN;
#define li_auth list_rchan_un.li_un_auth                                    10
#define li_dir list_rchan_un.li_un_dir

#define NULLIST_RCHAN          ((LIST_RCHAN *)0)
```

Figure 2.1: Channel List Structure

The use of the structure members are:

`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 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:

`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;
```

Two other useful routines are those to convert a string into a channel structure. The first one 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 first to allocate a channel list with one of the above functions, but not to fill in the channel part of it. This should be filled in with the routine:

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

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

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;
```

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

Figure 2.3: Management Parameter Structure

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;
```

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 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 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`, frees off all allocated variables within the structure.

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

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

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).

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 */           10
        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 */       20
#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;
```