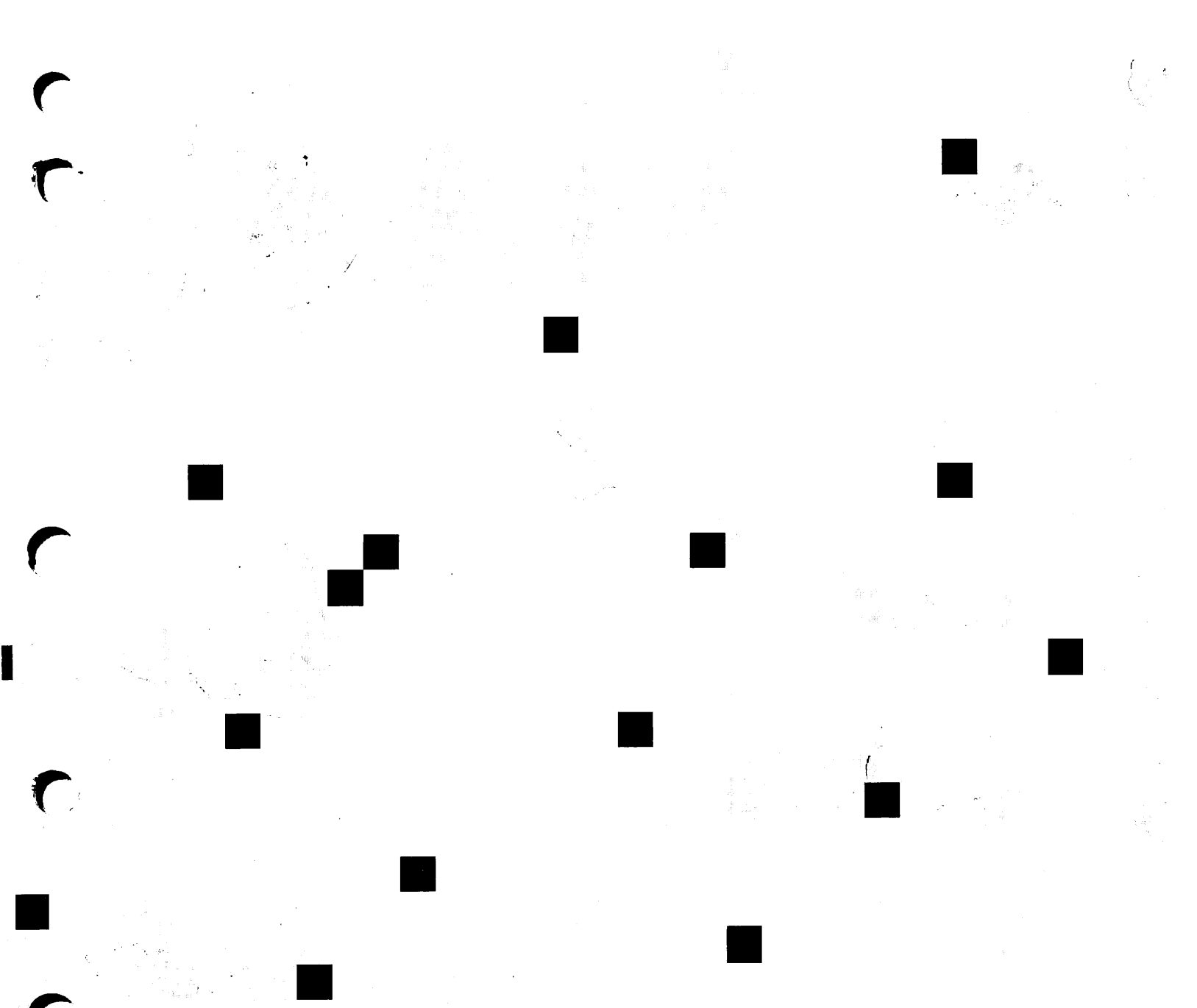


Prime®

X.400 Programmer's Guide

Release 1.1

DOC11277-1LA

X.400 Programmer's Guide

First Edition

Liz Parsons

This book documents the use of Prime X.400 at Release 1.1., which runs on PRIMOS® Master Disk Revision Levels 21.0.3 and above, and 22.0 and above.

Prime Computer, Inc., Prime Park, Natick, MA 01760

This document was prepared in the United Kingdom by Technical Publications Department, International Systems Marketing and Development, Willen Lake, Milton Keynes, MK15 0DB, United Kingdom.

PRINTING HISTORY

First Edition (DOC11277-1LA) March, 1989 for Release 1.1

CREDITS

Design: UK Technical Publications
Editorial: John Wells
Project Support: Alan Mynard
Illustration: Kevin Maguire
Document Preparation: Kevin Maguire
Production: Prime Technical Publications production unit

HOW TO ORDER TECHNICAL DOCUMENTS

To order copies of documents, or to obtain a catalog and price list:

United States Customers	*International*
Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).	Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-343-2320

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

CONTENTS

APPENDICES

ABOUT THIS BOOK

The X.400 Programmer's Guide is a reference to the X.400 Application Programming Interface (API). The book is written for programmers that use Prime X.400 library routines to develop mail applications. It gives a brief overview of Prime X.400, and describes the function and use of the X.400 application programming subroutines.

Chapter Contents

Chapter 1	Introduction, introduces the Prime X.400 product and OSI architecture.
Chapter 2	Programming Using the Prime X.400 API, introduces Prime X.400 concepts, describes the types of message available to User Agents (UAs) and gateways, and describes how a user programs the Prime X.400 API to construct, send, and receive messages correctly.
Chapter 3	Prime X.400 API Library, contains details of Prime X.400 library subroutines, in easy reference format.
Appendix A	Non-C Syntax API Library Routines, lists the PL1 parameter types that correspond to the C parameter types used in the API library routine descriptions in Chapter 3. It lists the PL1 syntax of each API library routine, and describes three API library routines (described in C) that are used for calling with non-C file units.
Appendix B	Example Application Program to Send a Message, lists the standard code used to send an X.400 message using the Prime X.400 API.
Appendix C	X.400 API Library Routine Return Values, lists the return values of each of the X.400 API library routines.

Related Documentation

Other Prime manuals which may be useful are

- *X.400 Administrator's Guide* (*DOC11276-1LA*)

Other manuals which you may find useful are

- *CCITT 1984 Red Book Volume VIII Fascicle VIII.7, Recommendations X.400 X.430*

Prime Documentation Conventions

The following conventions are used throughout this book. Examples illustrate how you use these commands and statements in typical applications.

Convention	*Explanation*	*Example*
UPPERCASE	Uppercase words indicate file names, and directory names.	SYSCOM
Italics	In text, italics indicate API routine arguments. In message header and envelope data structure data item descriptions, italics indicate description emphasis, message sequence types and body part types.	*key* *Reply Indication* *Content Type* *ForwardedIPMessage*
UPPERCASE/**Boldface**	Uppercase boldface words indicate C file pointer return values, keys which enable access to root data structures, and data items found in message envelope and header data structures.	**X4_OK** **X4_K_ROOT_BODY** **X4_K_SUBJECT**
UPPERCASE/*Italics*	Uppercase italic words indicate data structure types.	*X4_TIME*
Monospace	User examples and program listings are displayed in monospace.	`code = x4_logoff(pid);`

1

INTRODUCTION

OSI Communications

Open Systems Interconnection (OSI) is a set of internationally recognized recommendations, made by the International Organization for Standardization (ISO), and the International Telegraph and Telephone Consultative Committee (CCITT), that enable communications based upon a seven layer architectural model (illustrated in Figure 1-1).

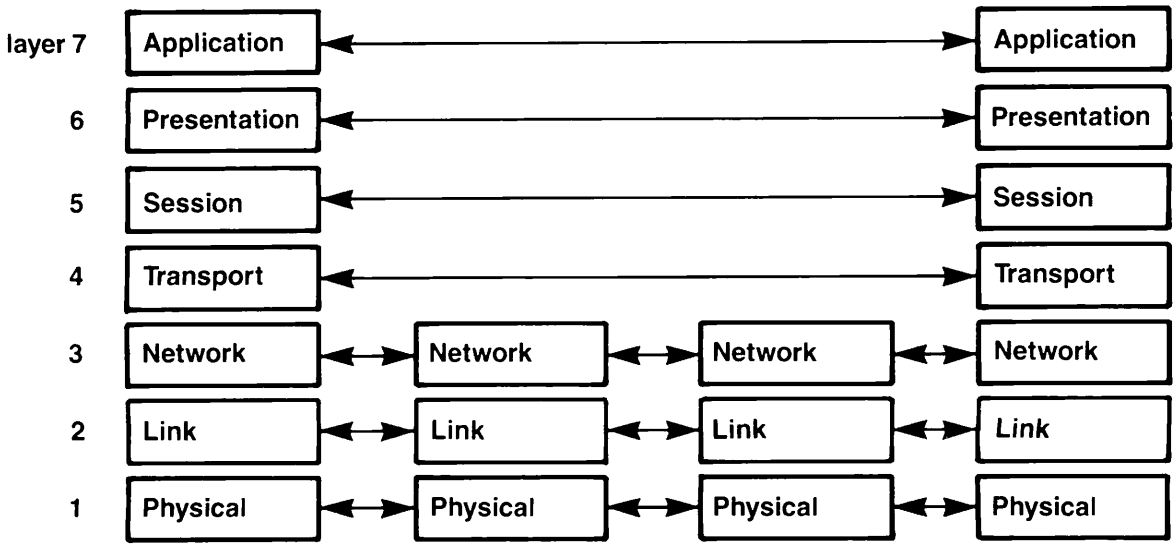


FIGURE 1-1. The OSI Reference Model

Each layer in the model is defined in terms of the service it provides to the layer above, the service it expects from the layer below, and the protocol used to communicate between equivalent entities within the same layer at different points within a network.

Overview of X.400

X.400 is a series of protocols that define a store-and-forward Message Handling System (MHS) for the exchange of messages between computer network users. X.400 is implemented in layer 7 of the OSI Reference Model (refer to Figure 1-1).

Figure 1-2 illustrates the architectural layers of X.400.

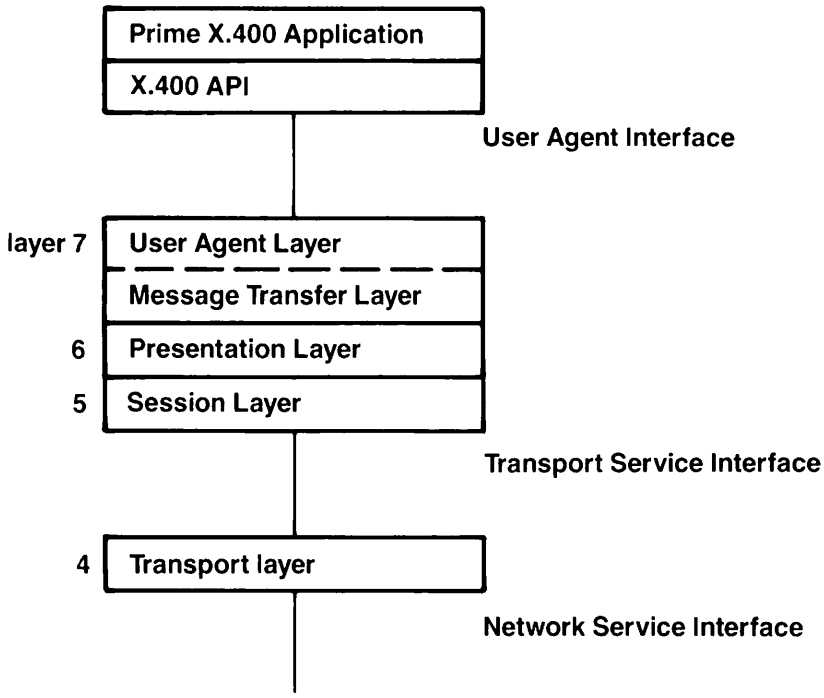


FIGURE 1-2. The Architectural Layers of X.400

The user agent interface is a message based interface that enables an X.400 application to send and receive messages via the user agent layer.

The X.400 Model

The X.400 series of definitions and protocols define a logical network model to which all X.400-compatible message handling systems must conform. The model comprises two types of software process; Message Transfer Agents (MTAs), and User Agents (UAs).

MTAs are store-and-forward nodes on an X.400 network. They act as servers for the exchange of messages across a network, cooperating with each other to ensure delivery.

They act as intermediaries between UAs to determine destinations, control routing, deliver messages, and signal errors.

UAs provide the link between users and MTAs. They interact with the sender, construct messages for submittal to MTAs, and display the messages to recipients at a destination node. UAs are implemented by mail applications.

The Prime X.400 Model

In accordance with the X.400 model, Prime X.400 comprises MTAs, that act as store-and-forward nodes for the exchange of messages across a network, and UAs, that interface with users to provide a Message Transfer Service (MTS). Figure 1-3 illustrates the main components of the Prime X.400 logical network.

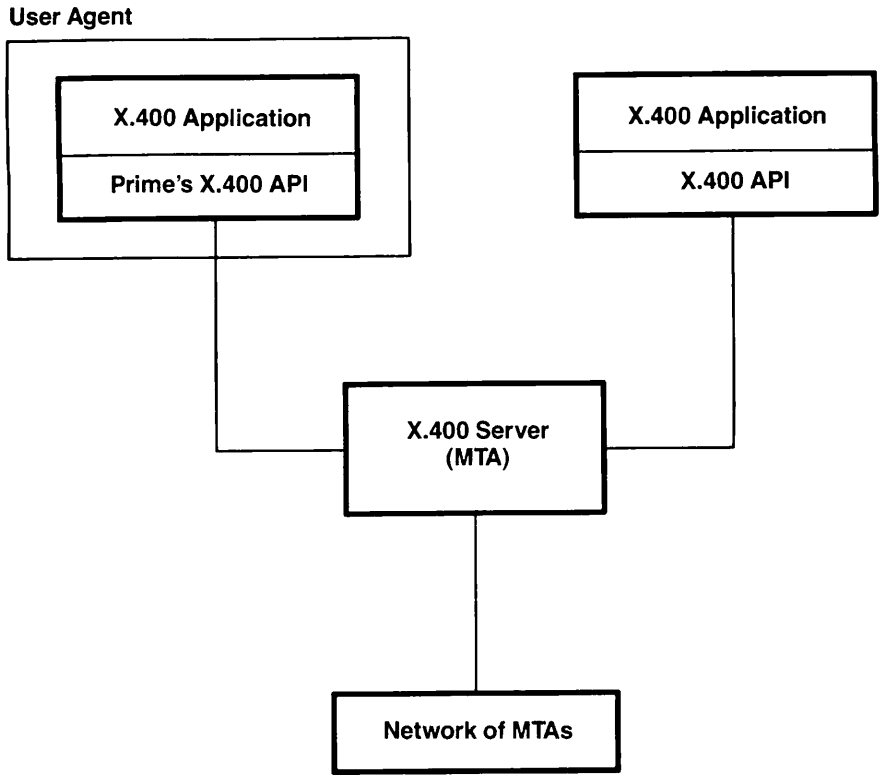


FIGURE 1-3. The Prime X.400 Logical Network

Prime X.400 UAs are implemented by X.400 applications, which use the services provided by the Application Programming Interface (API).

The Prime X.400 API is a set of library calls (subroutines) that:

- Establish a communication path to a Prime X.400 server process
- Establish a Prime X.400 session

- Allocate storage and initialize a message envelope data structure or message header data structure
- Release storage for the message envelope data structure or message header data structure
- Add fields to the message envelope data structure or message header data structure
- Send interpersonal messages using message envelope data structures and message header data structures
- Request that incoming messages, delivery notifications, and receipt notifications be read
- Fetch individual fields from message envelope data structures and message header data structures
- Action receipt of mail
- Terminate a Prime X.400 session
- Terminate a communication path to a Prime X.400 server process
- Convert X.409-encoded IA5 text body files to Prime ECS, and converts Prime ECS text files to X.409-encoded IA5 text body files
- Convert a Prime ECS character string to a T.61 character string, and converts a T.61 character string to a Prime ECS character string

2

PROGRAMMING USING THE PRIME X.400 API

This chapter introduces Prime X.400 concepts, describes the types of message available to User Agents (UAs) and Gateways, and describes how a user programs the Prime X.400 API to construct, send, and receive messages correctly.

Prime X.400 Concepts

This section introduces some Prime X.400 user agent, and gateway concepts.

Prime X.400 User Agent

An X.400 application (user agent) can establish multiple Prime X.400 sessions on a single communication path. On each session, the X.400 application can send and receive mail on behalf of a specific user that matches the configured address space in Prime's X.400 configuration file.

Prime X.400 Gateway

A gateway establishes a single Prime X.400 communication path and session. It uses this session to send and receive mail for multiple O/R addresses that match the configured address space for the gateway in Prime's X.400 configuration file.

X.400 Message Types for User Agents

The X.400 protocol provides for the following types of message for user agents:

- IPM Message
- Receipt Notification
- Delivery Notification

User agents can generate and receive IPM messages and receipt notifications, but can only receive delivery indications.

Message Type	*Function*
IPM Message Submission (IPMMS)	Interpersonal messages are transmitted over the message handling system to the recipients with an IPM message submission.
IPM Message Receipt (IPMMR)	Interpersonal messages transmitted over the message handling system are received by the recipient with an IPM message receipt.
Receipt Notification (RN)	Notification of the receipt of an interpersonal message is sent to the originator by the recipient with a receipt notification.
Receipt Notification Receipt (RNR)	A receipt notification sent by the recipient of an interpersonal message to the originator, is received by the originator with a receipt notification receipt.
Delivery Notification (DN)	Delivery of a message over the message handling system, to the recipient Message Transfer Agent (MTA), is confirmed to the originator with a delivery notification.

Note

X.400 message types for user agents are abbreviated in the table column headings later in this chapter.

X.400 Message Types for Gateways

The X.400 protocol provides for the following types of message for gateways:

- IPM Message
- Receipt Notification
- Delivery Notification

- Probe

Gateways can generate probe messages, and can generate and receive IPM messages, receipt notifications, and delivery notifications.

Message Type	*Function*
Delivery Notification Submission (DNS)	The recipient gateway of an incoming interpersonal message requesting a delivery notification sends a delivery notification submission to the originator.
Probe Submission (PS)	A gateway can check the ability of the message handling system to deliver an interpersonal message prior to actually sending the message with a probe submission.
Delivery Notification for Probes (DNP)	A gateway is notified of the results of a probe with a delivery notification.

Note

X.400 message types for gateways are abbreviated in the table column headings later in this chapter.

Other message types are the same as for user agents.

Message Structure

A message comprises an envelope, header and bodies. Figure 2-1, illustrates a standard message structure.

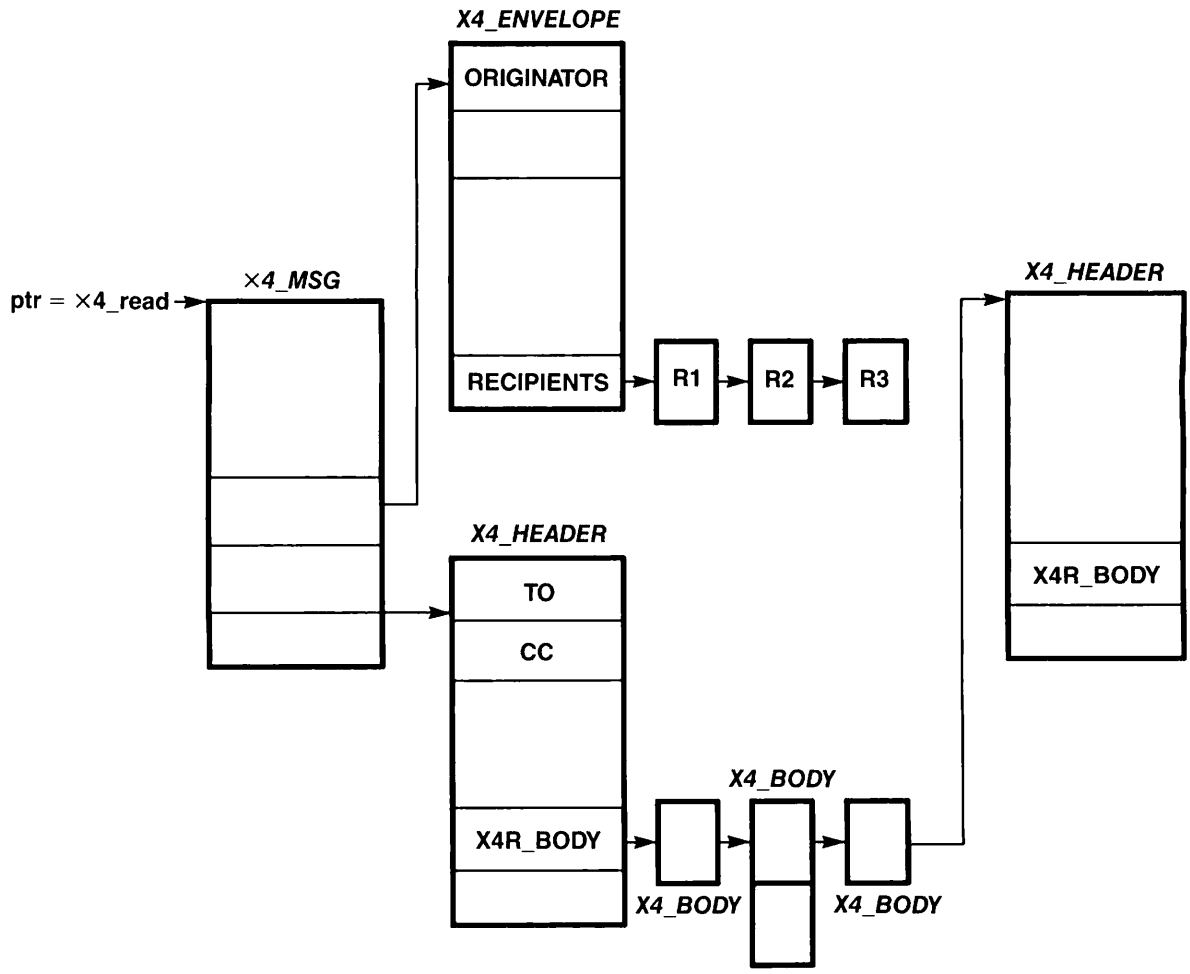


FIGURE 2-1. Message Structure

Each of the data items, listed in the message envelope and message header data structures, for each of the interpersonal message types, are described in API library routines; x4_get, and x4_put in Chapter 3, PRIME™ X.400 API LIBRARY. The following sections define the data items present in both the envelope and header data structures.

Message Envelope

The Message Envelope routes the body of the message from the originator to the recipients. The X.400-defined P1 protocol governs the flow of data between Message Transfer Agents, and conveys information that is an envelope for the Message.

Table 2-1, lists the data items associated with the message envelope data structure for each of the possible user agent interpersonal message types.

Note

The message type abbreviations used in this and following tables, are defined at the beginning of this chapter.

TABLE 2-1. Message Envelope Data Items for User Agent Message Types

	IPMMS	*IPMMR*	*RN*	*RNR*	*DN*
X4_K_CONTENT_ID	Y	Y	Y	Y	Y
X4_K_CONTENT_TYPE	Y	Y	Y	Y	-
X4_K_DEFERRED_DELIVERY	Y	Y	Y	Y	-
X4_K_ENCODED	Y	Y	Y	Y	-
X4_K_MPDU_ID	-	Y	-	Y	Y
X4_K_ORIGINATOR	-	Y	-	Y	Y
X4_K_PER_MESSAGE_FLAG	Y	Y	Y	Y	-
X4_K_PRIORITY	Y	Y	Y	Y	-
X4_K_RECIPIENT	Y	Y	Y	Y	-
X4_K_REPORTED_MESSAGE_ID	-	-	-	-	Y
X4_K_REPORTED_TRACE	-	-	-	-	Y
X4_K_TRACE	-	Y	-	Y	Y

Table 2-2 lists the data items associated with the message envelope data structure, for the additional gateway interpersonal message types.

TABLE 2-2. Message Envelope Data Items for Gateway Message Types

	DNS	*PS*	*DNP*
X4_K_CONTENT_ID	Y	Y	Y
X4_K_CONTENT_TYPE	-	Y	Y
X4_K_DEFERRED_DELIVERY	-	-	-
X4_K_ENCODED	-	Y	Y
X4_K_MPDU_ID	Y	Y	Y
X4_K_ORIGINATOR	Y	Y	Y
X4_K_PER_MESSAGE_FLAG	-	Y	Y
X4_K_PRIORITY	-	-	-
X4_K_RECIPIENT	Y	Y	Y
X4_K_REPORTED_MESSAGE_ID	Y	-	-
X4_K_REPORTED_TRACE	Y	-	-
X4_K_TRACE	Y	Y	Y
X4_K_LENGTH	-	Y	Y

Message Header

The Message Header is largely informative, and is not used by Prime X.400 for routing the Message Body. The X.400-defined P2 protocol governs the flow of data between the User Agent and the Message Transfer Agent, and conveys information that is a Header for the Message.

Table 2-3, lists the data items associated with the message header data structure for each of the possible user agent interpersonal message types.

TABLE 2-3. Message Header Data Items for Message Types

	IPMMS	*IPMMR*	*RN*	*RNR*
X4_K_ACTUAL_RECIPIENT	-	-	Y	Y
X4_K_AUTHORISE	Y	Y	-	-
X4_K_AUTO_FORWARD	-	Y	-	-
X4_K_BCC	Y	Y	-	-
X4_K_BODY	Y	Y	-	-
X4_K_CC	Y	Y	-	-
X4_K_DELIVERY_TIME	Y	Y	-	-
X4_K_ENCODED	-	-	Y	Y
X4_K_EXPIRES	Y	Y	-	-
X4_K_FROM	Y	Y	Y	Y
X4_K_IMPORTANCE	Y	Y	-	-
X4_K_IN_REPLY_TO	Y	Y	-	-
X4_K_INTENDED_RECIPIENT	-	-	Y	Y
X4_K_NON-RECEIPT_INFO	-	-	Y	Y
X4_K_OBSOLETES	Y	Y	-	-
X4_K_RECEIPT_INFO	-	-	Y	Y
X4_K_REF	Y	Y	Y	Y
X4_K_REPLY_BY	Y	Y	-	-
X4_K_REPLY_TO	Y	Y	-	-
X4_K_SENSITIVITY	Y	Y	-	-
X4_K_SUBJECT	Y	Y	-	-
X4_K_TO	Y	Y	Y	Y
X4_K_XREF	Y	Y	-	-

Message Body

The Message Body is the basic text of the message encoded in X.409 format. The CCITT X.400 recommendations support the following Body Types:

Body Type	*Description*
IA5Text	ASCII.
G3Fax	A sequence of bit strings, each representing a page of Group 3 facsimile information, encoded according to Recommendation T.4.

TIF0	A document, of a structure that is defined in Recommendation T.73, and that conforms to TIF (Text Interchange Format) 0 application rules.
TTX	Teletex.
NationallyDefined	Anything at all.
ForwardedIPMessage	A Message contained within the body of another Message, to be distributed to a further set of recipients. It optionally includes the original Message Header information.
SFD	A simple formatable document.
TIF1	A document, of a structure that is defined in Recommendation T.73, and that conforms to TIF1 application rules.

Prime X.400 provides encoding and decoding routines supporting IA5Text body types. Applications wishing to use other body types must perform their own body processing.

Data Structures

The file STRUC.H.INS.C in the top-level directory SYSCOM contains the definition of all the data structures used by Prime X.400 messages, and the keys that are used to identify these structures.

There is one structure for each primitive component of a message. Each structure starts with a standard substructure indicating the type of structure and whether it contains valid data. This substructure is defined as:

```
typedef struct {
        short id;       /* Data structure ID. */
        short rev;      /* Revision number */
        short valdata; /* TRUE (non-zero) if structure contains valid data */
}
X4_STRUC;
```

The *id* is set to the key for the main structure. The *rev* is set to the value that indicates the particular revision of the structure. The latest revision number is given by the key **X4_REV** found in the file STRUC.H.INS.C in the top-level directory SYSCOM. The *valdata* is a boolean value where 0 represents FALSE, and any other value represents TRUE. This is set to TRUE if the main structure contains valid data.

Consider the following example primitive data structure, which is the priority indication for a message:

```
typedef struct {
        X4_STRUC struc;
        short    value;
}
X4_PRIORITY;
```

The *structid* would be set to **X4_ID_PRIORITY**, and *struc.rev* to **X4_REV.** If *struct.valdata* is non-zero, then *value* contains the priority of a message. A default priority message would be indicated by the absence of an *X4_PRIORITY* structure in the message, or a structure present with the *struct.valdata* field set FALSE.

The more complex message components comprise structures containing these primitive data structures, and lists of structures. For example, the standard attributes of an O/R name are specified by the following structure:

```
typedef struct {
        X4_STRUC struc;
        X4_COUNTRYNAME cname; /* optional */
        X4_ADMD admd; /* optional */
        X4_X121 x121; /* optional */
        X4_TERM term; /* optional */
        X4_PRMD prmd; /* optional */
        X4_ORGNAME orgname; /* optional */
        X4_UNIQUEUAID uaid; /* optional */
        X4_NAME name; /* optional */
        X4R_ORGUNIT orgunit; /* optional */
}
X4_STDATT;
```

The *struct* is the standard header describing the attribute structure. The following are all primitive data structures; *cname*, *admd*, *x121*, *term*, *prmd*, *orname*, *uaid*, and *name*. The special structure *orgunit* defines the root of a list of organization unit primitive structures. Prime X.400 provides routines for manipulating the elements of this list; x4_enchain adds a primitive structure to a list; x4_find locates a particular entry in a list. The API routine x4_get can be used to retrieve successive elements of a list.

Prime X.400 data structures can be initialized (all *valdata* fields set to FALSE) using the routine x4_init. They can be dynamically allocated and initialized using the routine x4_alloc.

An X.400 message is built by adding Prime X.400 data structures to a message header and envelope (Refer to Appendix B, EXAMPLE APPLICATION PROGRAM TO SEND A MESSAGE). This operation is performed using x4_put. When reading a message, the elements of the envelope and header are extracted using the routine x4_get. Repetitive

calls of x4_get returns the same structure for nonlist items, and successive list elements for list structures.

Note

When using x4_get to process a list, a NULL pointer is returned *and an* **X4_ERR_END_OF_LIST** error raised when the end of the list is reached. This error must be cleared using x4_clear before any further API routines are made. The process of scanning a list using x4_get also changes pointers within the list. If a second pass of the list is required, you should use x4_find to reset these pointers to the beginning of the list.

Two primitive data structures *X4_SUBJECT*, and *X4_FREEFORMNAME* represent the subject of a mail item, and the free-form name of a mail user respectively. The data element in these structures is a character string. This character string is *not* a standard Prime ECS string, but is encoded according to Prime's implementation of T.61 (the most significant bit is the reverse of the standard T.61 encoding). Two API library routines are provided to convert this string from Prime ECS to T.61, and from T.61 to Prime ECS (refer to Chapter 3, PRIME X.400 API LIBRARY, routines x4_enct61 and x4_dect61).

Using the Prime X.400 API Routines

Prime X.400 is an implementation of the X.400 OSI message handling system, and includes a series of thirty API library routines. These routines are listed and described in Chapter 3, PRIME X.400 API LIBRARY.

The API library routines are provided to help the programmer create a user application program, that interactively sends and receives messages.

Each of the following subsections explain the Prime X.400 API routines, with examples of their execution in a typical user application program.

Error Handling

Some API routines return error codes as their results, others return NULL pointers indicating an error. In each case, full details of the error can be obtained by executing the x4_error API library routine. For example:

```
if (x4_error(&error, &qualifier))
   printf("Error %d Qualifier %d.\n", error, qualifier);
```

The two parameters *error*, and *qualifier*, contain keys indicating the error that has occurred. The values of *error* are defined in the file X4_ERROR.H.INS.C. The *qualifier* parameter contains a key, the value of which depends on the value of *error*.

Errors can be cleared using API library routine x4_clear:

```
x4_clear();
```

The API library routine x4_clear sets the two parameters *error* and *qualifier* in x4_error to zero.

Once an error has occurred, all subsequently called API routines return the same error, until it has been cleared using x4_clear.

Refer to Appendix C, X.400 API LIBRARY ROUTINE RETURN VALUES, for a list of the return values produced by the API library routines.

Establishing a Communication Path to a Prime X.400 Server

To communicate with Prime X.400, the user application program must establish an Inter Server Communication (ISC) session with the Prime X.400 server. This is achieved using the x4_open_uai API library routine

```
ptr=x4_open_uai("", 1);
```

Note

A gateway application would use the x4_open_gwi routine to establish an ISC session with the Prime X.400 server.

The arguments to x4_open_uai are *server_node* (char *), and *retired* (int). The argument *retired* is present to maintain compatibility with previous versions of the API. Its value is not used.

To terminate an ISC communication session, the API library routine x4_close must be called:

```
code = x4_close();
```

code is an integer value, where 0 indicates that the termination was successful, and a nonzero value indicates an error (which can be explained using API library routine x4_error).

Establishing a Prime X.400 Session

When a communication path to a Prime X.400 server has been successfully established, the user application program, connects to one or more user agents (depending upon the parameter). In order to send and receive messages on behalf of a user, the user application program must first establish a session to the Prime X.400 server. This is achieved using API library routine x4_logon:

```
pid = (X4_MSG *) x4_logon(user_name, mail_directory, mode);
```

The pointer *pid*, points to a structure of type *X4_MSG*. After a successful logon, this structure contains the number of outstanding messages waiting to be read. The pointer must be given as input to other API routines. When executing x4_logon, Prime X.400 searches the servers configuration table for a match against the *user_name* (char *). The argument *mail_directory* (char *) is the name of a user directory, or sub directory, where messages can be sent to, or received from. The argument *mode* (int), can be **X4_SEND**, **X4_RECEIVE**, or both (logical OR). The API library routine x4_logoff is used to terminate a Prime X.400 session started by x4_logon:

```
code = x4_logoff(pid);
```

The argument *pid*, is the same pointer received from x4_logon. *code* (int) defines the success or failure of the Prime X.400 session termination.

Reading Data

Incoming messages are received from Prime X.400 using the API library routine x4_read:

```
msg_ptr = (X4_MSG *) x4_read(300000);
```

The pointer *msg_ptr* references a structure of type *X4_MSG* which contains information on the message types, and pointers to the header and envelope data structures. Message types can be; interpersonal messages, delivery reports, or replies (each requiring separate treatment). The argument to x4_read is a wait period, specified in milliseconds.

Retrieving Information

Once a message has been received the information in it can be retrieved. This information will be data items such as where the message came from, who sent it, what type of message it is, and so on; each parameter requiring special attention depending on validation fields. Information is retrieved using the API library routine x4_get, which returns a pointer to various structures that depend on the value of the key.

The arguments to x4_get are the pointer to the envelope or header (from x4_read), and a key indicating the item required. The routine x4_get returns a pointer to a declared structure. In the example below, the recipient is being extracted from the envelope. The returned structure has *valdata* (int) set to 1 if there is valid data in the string. In the example, the user application program would print "To: ", the recipients first name (or " ") and then the last name (or " ").

```
x4_P1Recipient *recipient;

recipient = (x4_P1Recipient *) x4_get(env_ptr, X4_RECIPIENT);
if (recipient != NULL)
   printf("\nTo: %s %s  ",
      (recipient->orname.stdatt.name.forname.struc.valdata) ?
       recipient->orname.stdatt.name.forname.string     : "\"\"",
      (recipient->orname.stdatt.name.surname.struc.valdata) ?
       recipient->orname.stdatt.name.surname.string     : "\"\"");
```

The routine x4_get may return linked lists of information in the previous, and next standard formats, particularly when multiple occurrences are permitted (as for when there are multiple recipients).

Decoding and Encoding Files

At this stage in the user application program when an interpersonal message containing IA5 text has been received, the X.409-encoded IA5 text body file should be decoded to a Prime ECS (Extended Character Set) file. This is achieved using the API library routine x4_decia5:

```
x4_decia5(dest, src);
```

This routine decodes the body file accessed by the file pointer *src*, and saves the decoded contents in the file accessed by the file pointer *dest*.

Accepting or Rejecting Mail

If decoding and copy was successful, the application should accept the message through the API library routine x4_accept, which deletes the message from Prime X.400's reliable transfer store.

```
x4_accept(pid);
```

If the copy is not successful, API library routine x4_reject is called:

```
x4_reject(pid);
```

In both API library routines, the argument *pid* is received from x4_logon, pointing to the structure *X4_MSG*.

Sending a Message

Sending a message is accomplished in a similar manner to receive:

Encode using x4_encia5, put data items into header and envelope using x4_put, and send using x4_send.

Encoding a Prime ECS file to an X.409-encoded IA5 text body file is achieved using API library routine x4_encia5:

```
x4_encia5(dest, src);
```

This routine reads the file accessed by the file pointer *src*, then writes an X.409-encoded IA5 text body to the file accessed by the file pointer *dest*.

Terminating Connections

API routine x4_logoff is used to disconnect from a particular user agent. API routine x4_close terminates the communication path to the Prime X.400 server.

Handling User Agent Messages with the API

IPM Message Submission

The API routine x4_send is used to submit an IPM message.

An X.400 application, having established a communication path to a Prime X.400 server process, and a Prime X.400 session to a user, uses the X.400 API library routine x4_send to send a message over the message handling system (the message is submitted to Prime X.400 using information stored in the nominated envelope data structure, and header data structure).

The following API library routines are used to send data over the message handling system:

```
x4_open_uai(server_node, retired)
x4_logon(user_name, directory, mode)
x4_put(struct, key, arg) /* To build an IPM message */
x4_send(logon_ptr, envelope, header)
```

This IPM message submission sequence can be terminated by ending the Prime X.400 session, using x4_logoff, then closing the communication path to the Prime X.400 server process, using x4_close.

IPM Message Receipt

The API routine x4_read is used to check for an IPM message receipt.

An X.400 application, having established a communication path to a Prime X.400 server process, and a Prime X.400 session to a user, uses the X.400 API library routine x4_read to wait for, and read, an incoming interpersonal message.

When the application has finished processing the mail, the API library routine x4_accept or x4_reject must be called to accept, or reject respectively, responsibility for the mail. In either case the message is deleted from the Prime X.400 reliable transfer store.

If the recipient X.400 application terminates the Prime X.400 session, using x4_logoff without calling x4_accept or x4_reject, Prime X.400 saves the message, and attempts to deliver it the next time the user establishes a Prime X.400 session, and calls x4_read.

Individual data fields within a message can be retrieved or added, at any time during mail processing, by calls to x4_get, or x4_put respectively. Once a message has been accepted, or rejected, the recipient X.400 application can call x4_read to wait for any other incoming messages.

The recipient X.400 application can reply to a message that requests receipt notification with x4_reply, prior to accepting or rejecting the message, or terminating the Prime X.400 session.

The following API library routines are used to read data transmitted over the message handling system:

```
x4_open_uai(server_node, retired)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_get(struct, key)/x4_put(struct, key, arg)
x4_accept(logon_ptr)/x4_reject(logon_ptr)/x4_logoff(logon_ptr)
```

Receipt Notification

The API routine x4_reply is used to generate a receipt notification.

If the originator of a message requests a reply, the recipient X.400 application must build a message and call x4_reply, which sends the receipt notification to the originating X.400 application. A receipt notification can be made prior to the user accepting or rejecting the message after x4_read, or after the message has been accepted, and, before another call to x4_read.

The following API library routines are used to build a receipt notification to data transmitted over the message handling system:

```
x4_open_uai(server_node, retired)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_get(struct, key)/x4_put(struct, key, arg) /* To build a receipt notification */
x4_reply(logon_ptr, envelope, header)
x4_accept(logon_ptr)/x4_reject(logon_ptr)/x4_logoff(logon_ptr)
```

Receipt Notification Receipt

The API routine x4_read is used to check for a receipt notification receipt.

An X.400 application, having sent a message requesting a receipt notification, at some time later, calls x4_read to read the receipt from Prime X.400. The application can use x4_get to retrieve such data items as **X4_K_ACTUAL_RECIPIENT**, that indicate who sent the receipt notification. The reply indication must be accepted or rejected as with IPM message receipts.

The following API library routines are used to read a receipt notification to data transmitted over the message handling system:

```
x4_open_uai(server_node, retired)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_get(struct, key)/x4_put(struct, key, arg)
x4_accept(logon_ptr)/x4_reject(logon_ptr)/x4_logoff(logon_ptr)
```

Delivery Notification

The API routine x4_read is used to check for a delivery notification.

An X.400 application, having sent a message requesting a delivery notification, at some time later, calls x4_read to read the delivery notification.

The application can use x4_get to retrieve such data items as **X4_K_TRACE**, that indicate the passage of the message. The delivery notification must be accepted or rejected as with IPM message receipts.

The following API library routines are used to read a delivery notification:

```
x4_open_uai(server_node, retired)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_get(struct, key)/x4_put(struct, key, arg)
x4_accept(logon_ptr)/x4_reject(logon_ptr)/x4_logoff(logon_ptr)
```

Handling Gateway Messages with the API

The message types available to gateways, with the exception of delivery notification submission, probe submission, and delivery notification for probes, are the same as the message types available to user agents.

Delivery Notification Submission

The API routine x4_drnotify is used to generate a delivery notification.

If the originator of a message requests a delivery notification, the recipient gateway calls x4_drnotify which sends a delivery notification to the originator.

The following API library routines are called when sending a delivery notification:

```
x4_open_gwi(server_node)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_put(struct, key, arg) /* To build a delivery notification */
x4_drnotify(logon_ptr, envelope)
```

Probe Submission

The API routine x4_probe is used by a gateway to generate a probe.

If a gateway wishes to send a large message, it is wise to send a probe first to check that the recipient MTA is accepting messages.

Having sent a probe submission and received a positive delivery notification, it is then possible for the gateway application to send the message over the message handling system. However, delivery is not guaranteed, even if the delivery notification is positive.

The following API library routines are used when sending probes:

```
x4_open_gwi(server_node)
x4_logon(user_name, directory, mode)
x4_put(struct, key, arg) /* To build a probe */
x4_probe(envelope)
```

Delivery Notification for Probes

The API routine x4_read is used to check for a delivery notification.

A gateway application, having sent a probe, at some time later, calls x4_read to read the delivery notification from Prime X.400 indicating the validity of the X.400 route. The application can use x4_get to retrieve such data items as **X4_K_TRACE**, that indicate the passage of the message. The delivery notification must be accepted or rejected as with IPM message receipts.

The following API library routines are used to read a delivery notification for probes:

```
x4_open_gwi(server_node)
x4_logon(user_name, directory, mode)
x4_read(wait)
x4_get(struct, key)
x4_accept(logon_ptr)/x4_reject(logon_ptr)/x4_logoff(logon_ptr)
```

3

PRIME X.400 API LIBRARY

Introduction

This chapter lists all the Prime X.400 library subroutines. Each section describes the function, C syntax, purpose, and values returned for the subroutine.

If the user wishes to use PRIMOS® file units rather than C file pointers with the subroutines, refer to Appendix A, NON-C SYNTAX API LIBRARY ROUTINES.

Summary of Routines

x4_accept	Accepts responsibility for the incoming message
x4_alloc	Allocates and initializes a Prime X.400 data structure
x4_clear	Clears a Prime X.400 error condition
x4_close	Terminates a communication path to a Prime X.400 server process
x4_copy	Copies a Prime X.400 data structure
x4_decia5	Converts an X.409-encoded IA5 text file to Prime ECS
x4_dect61	Converts a T.61 character string to a Prime ECS character string
x4_drnotify	Sends a delivery notification from a gateway
x4_dump	Produces a formatted diagnostic print of a Prime X.400 data structure
x4_enchain	Adds a record to the end of a Prime X.400 linked list
x4_encia5	Converts a Prime ECS text file to X.409-encoded IA5 text
x4_enct61	Converts a Prime ECS character string to a T.61 character string
x4_error	Returns the current error status code and qualifier
x4_find	Locates items within a list data structure
x4_get	Returns the address of a data item from a nominated data structure
x4_getgdi	Returns the MTA global domain identifier
x4_getmta	Returns the MTA name
x4_init	Initializes a Prime X.400 data structure
x4_kill	Releases storage for all items in a list data structure
x4_logoff	Terminates a Prime X.400 session
x4_logon	Establishes a Prime X.400 session
x4_open_gwi	Establishes a communication path to a Prime X.400 server for use by a gateway
x4_open_uai	Establishes a communication path to a Prime X.400 server for use by a user

x4_probe	Sends a probe from a gateway
x4_put	Adds a data item to a Prime X.400 data structure
x4_read	Initiates a read of an awaiting message
x4_reject	Rejects responsibility for the incoming message
x4_release	Releases storage for a Prime X.400 data structure
x4_reply	Sends a message reply of type reply request
x4_send	Sends a message of type data request

Also

x4_save — Saves an X400 structure to a file

```
int x4_save (fp, struc)
FILE *fp;
char *struc;
```

x4_restore — Restores a saved X400 structure from a file

```
char *x4_restore (fp);
FILE *fp;
```

X4_ACCEPT

Function

Accepts responsibility for the last message read by x4_read, for a specified Prime X.400 logon session.

This routine and the API library routine x4_reject, are used to accept or reject mail.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_accept(logon_ptr)

char *logon_ptr;
```

Description

The x4_accept call acknowledges that the X.400 application has successfully handled the last incoming message for the specified Prime X.400 logon session (*logon_ptr*), and that the message can be deleted from the Prime X.400 reliable transfer store.

Prime X.400 deletes the stored message.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_ISC_ERR	An Inter Server Communication (ISC) error has occurred. The error qualifier contains the ISC error code.
X4_ERR_NO_READ	The user does not have an unanswered x4_read request.
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_ALLOC

Function

Allocates and initializes a Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_alloc(struc_id, version)

int struc_id;
int version;
```

Description

The x4_alloc call returns a pointer to an initialized Prime X.400 data structure. The version number must be **X4_REV**.

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_NO_RESOURCE	There were insufficient resources to allocate the data structure. The error qualifier contains the data structure ID.

X4_CLEAR

Function

Clears a Prime X.400 error condition.

This routine is used in error handling, with the API library routine x4_error.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

void x4_clear()
```

Description

The x4_clear call resets a Prime X.400 error condition. This routine must be called after a Prime X.400 error has occurred, otherwise all succeeding Prime X.400 API library calls return the same error.

Returns

The routine returns the following value:

Value	*Meaning*
X4_OK	The operation was successful.

X4_CLOSE

Function

Terminates a communication path to a Prime X.400 server process.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_close()
```

Description

The x4_close call terminates a previously opened communication path between a X.400 application and a Prime X.400 server.

Any logged-on Prime X.400 sessions are automatically logged off by this routine.

This routine has the opposite effect to x4_open_uai and x4_open_gwi API library calls.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_ISC_ERR	An Inter Server Communication (ISC) error has occurred. The error qualifier contains the ISC error code.
X4_ERR_NOT_OPEN	The user does not have a communication path open to Prime X.400.

X4_COPY

Function

Copies a Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_copy(struc1, struc2)

char *struc1;
char *struc2;
```

Description

The x4_copy call copies the contents of *struc2* to *struc1*.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_COPY	*struc2* is not the same type of data structure as *struc1*.
X4_ERR_BAD_REV	*struc2* or *struc1* contains an invalid version ID.
X4_ERR_BAD_STRUC	*struc2* or *struc1* is an invalid data structure ID.
X4_ERR_NO_DATA	*struc2* does not contain data (*valdata* field set false).

X4_DECIA5

Function

Decodes an X.409-encoded IA5 text body file into Prime ECS.

This routine and the API library routine x4_encia5, are used to decode, and encode files.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_decia5(dest, src)

FILE *dest;
FILE *src;
```

Description

The x4_decia5 call reads the open file accessed by the file pointer *src*, and strips out the X.409 encoding and converts the contents to Prime ECS. The result is written to the open file accessed by the file pointer *dest*.

If the user wishes to use PRIMOS file units rather than C file pointers with this call, refer to Appendix A, NON-C SYNTAX API LIBRARY ROUTINES.

Returns

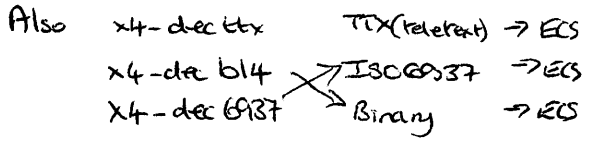
The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_EXIA5_STR	X.409 IA5 string expected. The error qualifier contains the type found.
X4_ERR_EXOCT_STR	X.409 octet string expected. The error qualifier contains the type found.
X4_ERR_EXSEQ	X.409 sequence expected. The error qualifier contains the type found.
X4_ERR_EXSET	X.409 set expected. The error qualifier contains the type found.

X4_ERR_EXTAG_INT X.409 tagged integer expected. The error qualifier contains the type found.

X4_ERR_FILE_ERR A file system error has occurred. The error qualifier contains the PRIMOS error code. The C library variable *errno* can also be set.

X4_ERR_UXSIZE Unexpected X.409 size. The error qualifier contains the size found.



X4_DECT61

Function

Decodes a T.61 character string to a Prime ECS character string.

C Syntax

```
#include <x4_error.h>

int x4_dect61(dest, src)

char *dest;
char *src;
```

Description

The x4_dect61 call converts the character string given by *src* from T.61 to a Prime ECS character string in *dest*.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_INVALID_CHARS	The source string contains characters that cannot be converted to Prime ECS.

X4_DRNOTIFY

Function

Sends a delivery notification from a gateway.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_drnotify(logon_ptr, envelope)

char *logon_ptr;
char *envelope;
```

Description

The x4_drnotify call sends a delivery notification in response to a previously received message. This routine can only be used when the gateway interface is in use. The routine returns a pointer to a static *X4_MPDUSTRING* data structure containing the MPDU identifier assigned by Prime X.400.

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_MDNP	A mandatory descriptor is missing from the envelope data structure provided. The error qualifier contains the structure ID of the missing descriptor.
X4_ERR_NO_RESOURCE	Prime X.400 is unable to accept this request. The error qualifier contains the reason for rejection: 1 = server reconfiguring, 2 = invalid header or envelope, 3 = X.400 server error.
X4_ERR_NOT_GWI	A communication path to a Prime X.400 server has been established using the x4_open_uai call rather than x4_open_gwi.

X4_ERR_NOT_OPEN	The gateway does not have a session open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_DUMP

Function

Produces a formatted diagnostic print of a Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_dump(fp, struc)

FILE *fp;
char *struc;
```

Description

The x4_dump call produces a formatted listing of the specified Prime X.400 data structure on the nominated open file unit.

If the user wishes to use PRIMOS file units with this call rather than C file pointers, refer to Appendix A, NON-C SYNTAX API LIBRARY ROUTINES.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.

X4_ENCHAIN

Function

Adds a Prime X.400 *list* data structure to the end of a linked list.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_enchain(root, list)

char *root;
char *list;
```

Description

The x4_enchain call adds a Prime X.400 *list* data structure to the end of the linked list indicated by the Prime X.400 root data structure.

Returns

The routine returns the following value:

Value	*Meaning*
X4_OK	The operation was successful.

X4_ENCIA5

Function

Encodes a Prime ECS text file as an X.409 encoded IA5 text body file.

This routine and the API library routine x4_decia5, are used to decode, and encode files.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_encia5(dest, src)

FILE *dest;
FILE *src;
```

Description

The x4_encia5 call reads the open file accessed by the file pointer *src*, then writes an X.409 encoded IA5 text body to the open file accessed by the file pointer *dest*.

If the user wishes to use PRIMOS file units rather than C file pointers with this call, refer to Appendix A, NON-C SYNTAX API LIBRARY ROUTINES.

Returns

The routine returns the following value:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_FILE_ERR	A file system error has occurred. The error qualifier contains the PRIMOS error code. The C library variable *errno* can also be set.

Also x4_encTtx ECS → TTX (teletext)
x4_encbin ECS → ISO6937
x4_enc6937 ECS → Binary

X4_ENCT61

Function

Encodes a Prime ECS character string to a T.61 character string.

C Syntax

```
#include <x4_error.h>

int x4_enct61(dest, src, maxlen)

char *dest;
char *src;
int maxlen;
```

Description

The x4_enct61 call converts the character string given by *src* from Prime ECS to a T.61 character string in *dest*. The input parameter *maxlen* gives the maximum resulting T.61 string length allowed.

Note

A T.61 string can be twice as long as the source ECS string.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_INVALID_CHARS	The source string contains characters that cannot be converted to T.61.
X4_ERR_TOO_LONG	The resulting T.61 string is longer than the maximum specified by *maxlen*.

X4_ERROR

Function

Returns the current error status code and qualifier.

This routine and the API library routine x4_clear, are used in error handling.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_error(error, qualifier)

int *error;
int *qualifier;
```

Description

The x4_error call returns the current error status code and qualifier.

Returns

The routine returns a non_zero (logical true) value if an error has occurred, or zero (logical false) if no error has occurred.

X4_FIND

Function

Locates items within a *list* data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_find(root, key)

char *root;
int key;
```

Description

The x4_find call returns a pointer to an item within the *list* data structure indicated by root.

Key may take the following values:

X4_K_FIRST	Returns a pointer to the first item in the list
X4_K_LAST	Returns a pointer to the last item in the list
X4_K_NEXT	Returns a pointer to the next item in the list
X4_K_PREVIOUS	Returns a pointer to the previous item in the list

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_KEY	The user has specified an invalid key.
X4_ERR_BAD_REV	An invalid data structure version was provided.
X4_ERR_BAD_STRUC	The user has specified an invalid data structure ID.
X4_ERR_END_OF_LIST	There are no more items in this list.
X4_ERR_LIST_EMPTY	There is no data item present for this list.

X4_GET

Function

Returns a data item from a message envelope data structure or message header data structure.

This routine is used to retrieve data.

C Syntax

```
#include <x4_struc.h>

#include <x4_keys.h>

#include <x4_error.h>

char *x4_get(struct, key)

char *struct;
int key;
```

Description

The x4_get call returns a pointer to an individual field within the message header data structure, or message envelope data structure, depending upon the type indicated by *struct*.

The following data items are found in the message envelope data structure, depending on the value of *key*:

Item	*Description*
X4_K_CONTENT_ID	The *UA Content Identifier*, provided by the UA and carried back to the originator (in a delivery indication) by the message transfer layer. It consists of a data structure of type *X4_UACONTENTID*. This parameter is limited to 16 characters in length.
X4_K_CONTENT_TYPE	A *Content Type* parameter, supplied by the originating UA, which identifies the convention that governs the structure of the contents. It consists of a data structure of type *X4_CONTENTTYPE*. The only defined value is *X4_CT_P2*, which identifies the P2 protocol for interpersonal messaging (as specified in CCITT recommendation X.420).
X4_K_DEFERRED_DELIVERY	A P1 field that specifies the earliest time that the message

can be delivered to the recipient. It consists of a data structure of type *X4_TIME*.

X4_K_ENCODED The encoding format used in the body of the message. It consists of a data structure of type *X4_ENCODED*.

X4_K_MPDU_ID The *Message Protocol Data Unit Identifier*, assigned by the originator UA. It consists of a data structure of type *X4_MPDUID*.

X4_K_ORIGINATOR The P1 originator name. It consists of a data structure of type *X4_ORNAME*.

X4_K_PER_MESSAGE_FLAG
A P1 options field which applies to all recipients of the message. The field consists of an *X4_PERMESSAGEFLAG* type data structure. This can be *X4_PMF_DISCLOSERECIPIENTS* (that indicates whether the O/R names of all recipients should be indicated to each recipient UA when the message is delivered), or *X4_PMF_CONVERSIONPROHIBITED* (that indicates whether the conversion is to be inhibited), or *X4_PMF_ALTERNATERECIPIENTALLOWED* (that indicates whether the alternate recipient allowed service is requested), or *X4_PMF_CONTENTRETURNREQUEST* (that indicates whether the content of the message is to be returned with any non-delivery notification).

X4_K_PRIORITY The P1 priority field. It consists of a data structure of type *X4_PRIORITY*, and can be *X4_P_NORMAL*, *X4_P_NONURGENT*, or *X4_P_URGENT*.

X4_K_RECIPIENT A P1 field that specifies the names of recipients for the message. This information is used for routing the message. It can occur more than once, and consists of a data structure of type *X4_P1RECIPIENT*. If the envelope is a *delivery notification*, then this field describes the reported recipients of the original message, and consists of a list of data structures of type *X4_REPORTEDP1RECIPIENT*.

X4_K_REPORTED_MPDU_ID
The message protocol data unit identifier of the message that is the subject of a *Delivery Notification*. It consists of a data structure of type *X4_MPDUID*.

X4_K_REPORTED_TRACE
Trace information associated with a message which is the subject of a *Delivery Notification*. It consists of a data structure of type *X4_TRACE*.

X4_K_TRACE Information (list of MTAs) of the passage of a message through the message transfer layer. It consists of a data structure of type *X4_TRACE*.

The following data items are found in the message header data structure, depending on the value of *key*:

Item	*Description*

X4_K_ACTUAL_RECIPIENT
A P2 field that is returned in a *Receipt Notification Receipt*, and that indicates the actual recipient who received the message. It consists of a data structure of type *X4_ORDESCRIPTOR*.

X4_K_AUTHORISE An optional P2 field that describes the user who authorized the message to be sent. There may be more than one authorizing user specified. The field consists of a data structure of type *X4_ORDESCRIPTOR*, and is not validated by Prime X.400.

X4_K_AUTO_FORWARD
Indicates that the message has been redirected by the original recipient message transfer agent. It consists of a data structure of type *X4_AUTOFORWARD*.

X4_K_BCC A P2 descriptor that identifies a blind copy recipient. That is, a recipient whose name is not disclosed to primary or copy recipients. It can occur once, several times, or not at all. It consists of the same fields as the primary recipient.

X4_K_BODY A field that describes the type of each body part within the file body. If a body part is of type *ForwardedIPMessage*, then it contains a reference to a separate message header data structure for the forwarded message. Such enclosures can be repeated.

X4_K_CC A P2 descriptor that identifies a copy recipient of the X.400 message. It can occur once, several times, or not at all. It consists of the same fields as the primary recipient.

X4_K_DELIVERY_TIME
A field that provides the message delivery time at the forwarding agent, if the body part is of type *ForwardedIPMessage*. This field is optional, and consists of a data structure of type *X4_TIME*.

X4_K_ENCODED A field that indicates the converted encoded information types of the message. It consists of a data structure of type *X4_ENCODED*.

X4_K_EXPIRES A P2 field that indicates the date and time by which the originator considers the message to be no longer valid, or useful. It is optional, and consists of a data structure of type *X4_TIME*.

X4_K_FROM A P2 field that identifies the user that submitted the X.400 message. It consists of a data structure of type *X4_ORDESCRIPTOR*, and is for information only. Prime X.400 does not validate this field.

X4_K_IMPORTANCE A P2 descriptor that gives an indication of the importance of the message being sent. It consists of a data structure of type *X4_IMPORTANCE*. Allowable values are *X4_IMP_LOW*, *X4_IMP_NORMAL* or *X4_IMP_HIGH*. If not present, a default value of *X4_IMP_NORMAL* is supplied.

X4_K_IN_REPLY_TO A P2 field that identifies a previous message to which this message is a reply. It is optional and consists of a data structure of type *X4_REF*.

X4_K_INTENDED_RECIPIENT
A P2 field that is returned in a *Receipt Notification Receipt*, and that indicates the intended recipient for the Message (where this is different to the actual recipient). It consists of a data structure of type *X4_ORDESCRIPTOR*.

X4_K_NON_RECEIPT_INFO
A field that provides information regarding nonreceipt of the message by the recipient UA. It consists of a data structure of type *X4_NONRECEIPTINFO*.

X4_K_OBSOLETES A P2 descriptor that identifies any previous messages that are made obsolete by this message. It can occur once, several times, or not at all. It consists of a data structure of type *X4_REF*.

X4_K_RECEIPT_INFO A field that provides information regarding receipt of the message by the recipient UA. It consists of a data structure of type *X4_RECEIPTINFO*.

X4_K_REF A P2 field that contains the message protocol data unit identifier, supplied by the originating X.400 application. It consists of a data structure of type *X4_REF*.

X4_K_REPLY_BY A P2 descriptor that gives the date and time by which a reply to this message should be sent. It is optional, and consists of a data structure of type *X4_TIME*.

X4_K_REPLY_TO A P2 descriptor that gives the names of users to whom the reply should be sent. It can occur once, several times, or

not at all. It consists of an *X4_ORDESCRIPTOR*, which must contain an *X4_ORNAME*.

X4_K_SENSITIVITY A P2 field that gives an indication of the sensitivity of the message being sent. It consists of a data structure of type *X4_SENSITIVITY*. Allowable values are *X4_SEN_PERSONAL*, *X4_SEN_PRIVATE* or *X4_SEN_COMPANYCONFIDENTIAL*. If not present, a value of *X4_SEN_PERSONAL* is supplied.

X4_K_SUBJECT A P2 descriptor that describes the subject of the X.400 message being sent. It can occur once, several times, or not at all. It consists of a data structure of type *X4_SUBJECT*.

X4_K_TO A P2 descriptor that identifies the primary recipient of the message. It can occur more than once. It consists of a data structure of type *X4_RECIPIENT*, which comprises an *X4_ORDESCRIPTOR*, an *X4_REPORT_REQUEST*, and an *X4_REPLY_REQUEST*. The *X4_REPORT_REQUEST* enables the user to select receipt notification, or nonreceipt notification, from the recipient UA. The *X4_REPLY_REQUEST* enables the user to request the recipient to acknowledge receipt by sending a reply.

X4_K_XREF A P2 descriptor that identifies any previous X.400 messages that are cross referenced by this X.400 message. It can occur once, several times, or not at all. It consists of a data structure of type *X4_REF*.

The following keys are provided to enable access to the root structures in the header data structure, and envelope data structure:

X4_K_ROOT_AUTHORISE
Accesses the root to the list of P2 authorize fields

X4_K_ROOT_BCC Accesses the root to the list of P2 BCC recipients

X4_K_ROOT_BODY Accesses the root to the list of *X4_BODY* structures, that describe the type of each body part within the message

X4_K_ROOT_CC Accesses the root to the list of P2 CC recipients

X4_K_ROOT_OBSOLETES
Accesses the root to the list of *X4_REF* structures, which identify the previous messages that have been made obsolete by this message

X4_K_ROOT_RECIPIENT
Accesses the root to the list of *X4_REPORTEDP1RECIPIENT* structures, that describe the reported recipients of the original message

X4_K_ROOT_REPLY_TO
Accesses the root to the list of *X4_ORDESCRIPTOR* structures, that identify the users to which a reply should be sent

X4_K_ROOT_REPORTED_TRACE
Accesses the root to the list of *X4_TRACE* structures, that make up the intermediate trace list

X4_K_ROOT_TO Accesses the root to the list of P2 primary recipient lists

X4_K_ROOT_TRACE Accesses the root to the list of *X4_TRACE* structures

X4_K_ROOT_XREF Accesses the root to the list of *X4_REF* structures, that identify which previous messages are cross referenced by this one

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_KEY	The user has specified an invalid key.
X4_ERR_BAD_REV	An invalid data structure version ID was provided.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_END_OF_LIST	There are no further data items of the list type requested.
X4_ERR_LIST_EMPTY	There is no data item present for the list type requested.
X4_ERR_NO_DATA	There is no data item present of the type requested.

X4_GETGDI

Function

Returns the MTA global domain identifier.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_getgdi();
```

Description

This routine returns a pointer to a dynamically allocated *X4_GLOBALDOMAINID* structure that contains the global domain identifier (Country, ADMD, and PRMD) of the MTA to which there is an open session. Combined with the MPDUID String returned by x4_send, the GDI gives the full MPDUID of a sent message.

It is the calling applications responsibility to free the allocated structure (using *free*) when it is no longer required.

Returns

If the routine returns a NULL pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_BAD_MESSAGE	An invalid message format has been received.

X4_GETMTA

Function

Returns the MTA name.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_getmta();
```

Description

This routine returns a pointer to a static string containing the name of the MTA to which there is an open session.

Returns

If the routine returns a NULL pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_BAD_MESSAGE	An invalid message format has been received.

X4_INIT

Function

Initializes a Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_init(struc, key, version)

char *struc;
int key;
int version;
```

Description

The x4_init call initializes a Prime X.400 data structure of type *struc*. The value of *key* qualifies the data structure, and is one of the X4_ID keys found in the X4_STRUC.H.INS.C file in the top-level directory SYSCOM. The version must be passed. The latest version which should be supplied in all normal circumstances, is X4_REV which is held in the X4_STRUC.H.INS.C file in the top-level directory SYSCOM. If the value of *key* is incorrect, the routine returns **X4_ERR_BAD_STRUC**.

The content of individual fields can be added using the x4_put library call.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.

X4_KILL

Function

Releases storage for all items in a list data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_kill(root)

char *root;
```

Description

The x4_kill call releases storage for all data items in a list data structure indicated by *root*. The root data structure itself, is updated to indicate an empty list.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_STRUC	The data structure provided was not a root data structure.

X4_LOGOFF

Function

Disconnects a user session between an X.400 application, and a Prime X.400 server process.

This routine and the API library routine x4_logon, are used in establishing, and terminating, a Prime X.400 session.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_logoff(logon_ptr)

char *logon_ptr;
```

Description

The x4_logoff call terminates a previously opened session between an X.400 application, and Prime X.400 server process.

This routine has the opposite effect to the x4_logon API library call.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_ISC_ERR	An Inter Server Communication (ISC) error has occurred. The error qualifier contains the ISC error code.
X4_ERR_NOT_OPEN	An ISC session was not open.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_LOGON

Function

Establishes a user session between an X.400 application, and a Prime X.400 server process.

This routine and the API library routine x4_logoff, are used in establishing, and terminating, a Prime X.400 session.

C Syntax

```
#include <x4_struc.h>

#include <x4_keys.h>

#include <x4_error.h>

char *x4_logon(user_name, directory, mode)

char *user_name;
char *directory;
int mode;
```

Description

The x4_logon call establishes a user session between an X.400 application, and a Prime X.400 server process. Prime X.400 searches the configuration file for a match against the *user_name*. If the match is successful, the configuration file contains the X.400 ORAddress for this *user_name*. This ORAddress is used as the P1 originator field for all transmitted messages from this user.

The x4_logon call returns a pointer to a Prime X.400 logon data structure (of type *X4_MSG*). This pointer is used as an argument in subsequent API calls to identify this Prime X.400 session. This structure contains a count of the number of mail items waiting for the user to read.

The *directory* name provided is the destination location for incoming mail body parts, and the default source location for outgoing mail body parts. If null, it defaults to a sub-directory called X400_MAIL in the users' origin directory.

The *mode* can be **X4_K_RECEIVE**, **X4_K_SEND**, or both (logical OR).

Returns

If the routine returns a null value, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_RESPONSE	An unrecognized message type from the Prime X.400 server. The error qualifier contains the message type in question.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_LOGGED_ON	The user is already logged on.
X4_ERR_NAC	The user does not have access rights to this X.400 user name.
X4_ERR_NO_RESOURCE	Prime X.400 has no resource available to support this user.
X4_ERR_NOT_OPEN	An ISC session is not open.
X4_ERR_RECONFIGURING	The Prime X.400 server is reconfiguring.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.
X4_ERR_TERMINATED	Prime X.400 has closed down.
X4_ERR_UNKNOWN_USER	The user name is not present in the configuration file being used by Prime X.400.

X4_OPEN_GWI

Function

Establishes a gateway communication path to a Prime X.400 server process.

This routine and the API library routine x4_close, are used to establish and terminate a communication path to a Prime X.400 server.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_open_gwi(server_node)

char *server_node;
```

Description

The x4_open_gwi call establishes a communication path for gateways, between an X.400 application and a Prime X.400 server on a specified processor node.

The argument *server_node* is the Primenet node name of the node where the server resides, or NULL for the local node.

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_OPEN	The user already has a path open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.
X4_ERR_BAD_OPEN	Attempt to ~~connect to~~ open a session on another node that is an incompatible version.

X4_OPEN_UAI

Function

Establishes a user communication path to a Prime X.400 server process.

This routine, and the API library routine x4_close, are used to establish and terminate a communication path to a Prime X.400 server.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_open_uai(server_node, retired)

char *server_node;
int retired;
```

Description

The x4_open_uai call establishes a communication path for users, between an X.400 application and a Prime X.400 server on a specified processor node.

The argument *retired* is present to maintain compatibility with previous versions of the API. Its value is not used.

The argument *server_node* is the Primenet node name of the node where the server resides, or NULL for the local node.

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_OPEN	The user already has a path open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.
X4-ERR-BAD-OPEN	Attempt to open a session on another node that is an incompatible version.

X4_PROBE

Function

Sends a message probe from a gateway.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_probe(envelope)

char *envelope;
```

Description

The x4_probe call checks the validity of an X.400 route, using information previously stored in a message *envelope* data structure, before sending a message. The routine returns a pointer to the MPDU identifier structure (*X4_MPDUSTRING*) assigned by Prime X.400.

This routine can only be called by a logged on gateway user.

Returns

If the routine returns a null pointer, then x4_error returns following value:

Value	*Meaning*
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.

X4_PUT

Function

Adds a data item to a message envelope data structure, or message header data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_keys.h>

#include <x4_error.h>

int x4_put(struct, key, arg)

char *struct;
int key;
char *arg;
```

Description

The x4_put call adds the data item referenced by *arg*, to the message envelope data structure, or message header data structure, referenced by *struct*.

The following data items are found in the message envelope data structure, depending on the value of *key*:

Item	*Description*
X4_K_CONTENT_ID	The *UA Content Identifier*, provided by the UA and carried back to the originator (in a *delivery notification*) by the message transfer layer. It consists of a data structure of type *X4_UACONTENTID*. This parameter is limited to 16 characters in length.
X4_K_CONTENT_TYPE	A *Content Type* parameter, supplied by the originating UA, which identifies the convention that governs the structure of the contents. It consists of a data structure of type *X4_CONTENTTYPE*. The only defined value is *X4_CT_P2*, which identifies the P2 protocol for interpersonal messaging (as specified in CCITT recommendation X.420).
X4_K_DEFERRED_DELIVERY	A P1 field that specifies the earliest time that the message can be delivered to the recipient. It consists of a data structure of type *X4_TIME*.

X4_K_ENCODED The encoding format used in the body of the message. It consists of a data structure of type *X4_ENCODED.*

X4_K_MPDU_ID The *Message Protocol Data Unit Identifier*, assigned by the originator UA. It consists of a data structure of type *X4_MPDUID.*

X4_K_ORIGINATOR The P1 originator name. It consists of a data structure of type *X4_ORNAME.*

X4_K_PER_MESSAGE_FLAG
A P1 options field which applies to all recipients of the message. The field consists of an *X4_PERMESSAGEFLAG* type data structure. This can be *X4_PMF_DISCLOSERECIPIENTS* (that indicates whether the O/R names of all recipients should be indicated to each recipient UA when the message is delivered), or *X4_PMF_CONVERSIONPROHIBITED* (that indicates whether the conversion is to be inhibited), or *X4_PMF_ALTERNATERECIPIENTALLOWED* (that indicates whether the alternate recipient allowed service is requested), or *X4_PMF_CONTENTRETURNREQUEST* (that indicates whether the content of the message is to be returned with any non-delivery notification).

X4_K_PRIORITY The P1 priority field. It consists of a data structure of type *X4_PRIORITY*, and can be *X4_P_NORMAL*, *X4_P_NONURGENT*, or *X4_P_URGENT.*

X4_K_RECIPIENT A P1 field that specifies the names of recipients for the message. This information is used for routing the message. It can occur more than once, and consists of a data structure of type *X4_P1RECIPIENT.* If the envelope is a *delivery notification*, then this field describes the reported recipients of the original message, and consists of a list of data structures of type *X4_REPORTEDP1RECIPIENT.*

X4_K_REPORTED_MPDU_ID
The message protocol data unit identifier of the message that is the subject of a *Delivery Notification.* It consists of a data structure of type *X4_MPDUID.*

X4_K_REPORTED_TRACE
Trace information associated with a message which is the subject of a *Delivery Notification.* It consists of a data structure of type *X4_TRACE.*

X4_K_TRACE Information (list of MTAs) of the passage of a message through the message transfer system. It consists of a data structure of type *X4_TRACE.*

The following data items are found in the message header data structure, depending on the value of *key*:

Item	*Description*
X4_K_ACTUAL_RECIPIENT	A P2 field that is returned in a *Receipt Notification Receipt*, and that indicates the actual recipient who received the message. It consists of a data structure of type *X4_ORDESCRIPTOR.*
X4_K_AUTHORISE	An optional P2 field that describes the user who authorized the sending of the message. There may be more than one authorizing user specified. The field consists of a data structure of type *X4_ORDESCRIPTOR*, and is not validated by Prime X.400.
X4_K_BCC	A P2 descriptor that identifies a blind copy recipient. That is, a recipient whose name is not disclosed to primary or copy recipients. It can occur once, several times, or not at all. It consists of the same fields as the primary recipient.
X4_K_BODY	A field that describes the type of each body part within the body file. If the body part is of type *ForwardedIPMessage*, then it contains a reference to a separate message header data structure for the forwarded message. Such enclosures can be repeated.
X4_K_CC	A P2 descriptor that identifies a copy recipient of the X.400 message. It can occur once, several times, or not at all. It consists of the same fields as the primary recipient.
X4_K_DELIVERY_TIME	A field that provides the message delivery time at the forwarding agent, if the body part is of type *ForwardedIPMessage.* This field is optional, and consists of a data structure of type *X4_TIME.*
X4_K_ENCODED	A field that indicates the converted encoded information types of the message. It consists of a data structure of type *X4_ENCODED.*
X4_K_EXPIRES	A P2 field that indicates the date and time by which the originator considers the message to be no longer valid and useful. It is optional, and consists of a data structure of type *X4_TIME.*
X4_K_FROM	A P2 field that identifies the user that submitted the X.400 message. It consists of a data structure of type *X4_ORDESCRIPTOR*, and is for information only. Prime X.400 does not validate this field.

X4_K_IMPORTANCE A P2 descriptor that gives an indication of the importance of the message being sent. It consists of a data structure of type *X4_IMPORTANCE*. Allowable values are *X4_IMP_LOW*, *X4_IMP_NORMAL* or *X4_IMP_HIGH*. If not present, a default value of *X4_IMP_NORMAL* is supplied.

X4_K_IN_REPLY_TO A P2 field that identifies a previous message to which this message is a reply. It is optional and consists of a data structure of type *X4_REF*.

X4_K_INTENDED_RECIPIENT
A P2 field that is returned in a *Receipt Notification Receipt*, and that indicates the intended recipient for the Message (where this is different to the actual recipient). It consists of a data structure of type *X4_ORDESCRIPTOR*.

X4_K_NON_RECEIPT_INFO
A field that provides information regarding nonreceipt of the message by the recipient UA. It consists of a data structure of type *X4_NONRECEIPTINFO*.

X4_K_OBSOLETES A P2 descriptor that identifies any previous messages that are made obsolete by this message. It can occur once, several times, or not at all. It consists of a data structure of type *X4_REF*.

X4_K_RECEIPT_INFO A field that provides information regarding receipt of the message by the recipient UA. It consists of a data structure of type *X4_RECEIPTINFO*.

X4_K_REF A P2 field that contains the interpersonal message identifier supplied by the originating X.400 application. It consists of a data structure of type *X4_REF*.

X4_K_REPLY_BY A P2 descriptor that gives the date and time by which a reply to this message should be sent. It is optional, and consists of a data structure of type *X4_TIME*.

X4_K_REPLY_TO A P2 descriptor that gives the names of users to whom the reply should be sent. It can occur once, several times, or not at all. It consists of an *X4_ORDESCRIPTOR*, which must contain an *X4_ORNAME*.

X4_K_SENSITIVITY A P2 field that gives an indication of the sensitivity of the message being sent. It consists of a data structure of type *X4_SENSITIVITY*. Allowable values are *X4_SEN_PERSONAL*, *X4_SEN_PRIVATE* or *X4_SEN_COMPANYCONFIDENTIAL*. If not present, a value of *X4_SEN_PERSONAL* is supplied.

X4_K_SUBJECT A P2 descriptor that describes the subject of the X.400 message being sent. It can occur once, several times, or not at all. It consists of a data structure of type *X4_SUBJECT*.

X4_K_TO A P2 descriptor that identifies the primary recipient of the message. It can occur more than once. It consists of a data structure of type *X4_RECIPIENT*, which comprises an *X4_ORDESCRIPTOR*, an *X4_REPORT_REQUEST*, and an *X4_REPLY_REQUEST*. The *X4_REPORT_REQUEST* enables the user to select receipt notification or nonreceipt notification. The *X4_REPLY_REQUEST* enables the user to request the recipient to acknowledge receipt by sending a reply.

X4_K_XREF A P2 descriptor that identifies any previous X.400 messages that are cross referenced by this X.400 message. It can occur once, several times, or not at all. It consists of a data structure of type *X4_REF*.

The following keys are provided to enable access to the root structures in the header data structure, and envelope data structure:

X4_K_ROOT_AUTHORISE
Accesses the root to the list of P2 authorize fields

X4_K_ROOT_BCC Accesses the root to the list of P2 BCC recipients

X4_K_ROOT_BODY Accesses the root to the list of *X4_BODY* structures, that describe the type of each body part within the message

X4_K_ROOT_CC Accesses the root to the list of P2 CC recipients

X4_K_ROOT_OBSOLETES
Accesses the root to the list of *X4_REF* structures, which identify the previous messages that have been made obsolete by this message

X4_K_ROOT_RECIPIENT
Accesses the root to the list of *X4_REPORTEDP1RECIPIENT* structures, that describe the reported recipients of the original message

X4_K_ROOT_REPLY_TO
Accesses the root to the list of *X4_ORDESCRIPTOR* structures, that identify the users to which a reply should be sent

X4_K_ROOT_REPORTED_TRACE
Accesses the root to the list of *X4_TRACE* structures, that make up the intermediate trace list

X4_K_ROOT_TO	Accesses the root to the list of P2 primary recipient lists
X4_K_ROOT_TRACE	Accesses the root to the list of *X4_TRACE* structures
X4_K_ROOT_XREF	Accesses the root to the list of *X4_REF* structures, that identify which previous messages are cross referenced by this one

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_KEY	The user has specified an invalid key.
X4_ERR_BAD_REV	An invalid data structure version ID was provided.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.

X4_READ

Function

Initiates a read of the waiting message.

This routine is used to read data.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_read(wait)

int wait;
```

Description

The x4_read call waits for a signal from Prime X.400 for incoming messages, and returns a pointer to an *X4_MSG* data structure.

Wait is the maximum wait period (specified in milliseconds). Zero causes the routine to return immediately if there is no mail to read. A positive value causes the routine to wait for the indicated period before returning, if there is no mail immediately available. If a mail item arrives during this period, the routine returns. A negative value causes an indefinite wait.

Individual data fields may be retrieved by subsequent calls to x4_get.

When users have finished processing this mail, they must call x4_accept or x4_reject (in which case Prime X.400 deletes the stored message), or x4_logoff (in which case Prime X.400 attempts to deliver it the next time the user establishes a Prime X.400 session).

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_MESSAGE	An invalid message format has been received.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.

X4_ERR_NO_MESSAGE
There is no message waiting.

X4_ERR_NOT_OPEN The user does not have a session open to Prime X.400.

X4_ERR_SYN_ERR An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_REJECT

Function

Rejects responsibility for a received message.

This routine and the API library routine x4_accept, are used to accept or reject mail.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_reject(logon_ptr)

char *logon_ptr;
```

Description

The x4_reject call informs Prime X.400 that the X.400 application is unable to handle the incoming message.

Prime X.400 deletes the stored message.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_ISC_ERR	An Inter Server Communication (ISC) error has occurred. The error qualifier contains the ISC error code.
X4_ERR_NO_READ	The user does not have an unanswered x4_read request.
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.
X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_RELEASE

Function

Releases storage for a Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4_release(struct)

char *struct;
```

Description

The x4_release call releases storage for a valid initialized Prime X.400 data structure.

Does not delete body files. These are the caller's responsibility.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.

X4_REPLY

Function

Sends a message reply.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_reply(logon_ptr, envelope, header)

char *logon_ptr;
char *envelope;
char *header;
```

Description

The x4_reply call acknowledges a previously read message, using the information stored in the message *envelope* data structure.

This routine returns a pointer to a static *X4_MPDUSTRING* data structure that contains the MPDU identifier assigned by Prime X.400.

Returns

If the routine returns a null pointer, then X4_error returns one of the following:

Value	*Meaning*
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_MDNP	A mandatory descriptor is missing from the envelope or header structure provided. The error qualifier contains the structure ID of the missing descriptor.
X4_ERR_NO_RESOURCE	Prime X.400 is unable to accept this request. The error qualifier contains the reason for rejection: 1 = server reconfiguring, 2 = invalid header or envelope, 3 = X.400 server error.
X4_ERR_NOT_OPEN	The user does not have a session open to Prime X.400.

X4_ERR_SYN_ERR	An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_SEND

Function

Sends a message.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

char *x4_send(logon_ptr, envelope, header)

char *logon_ptr;
char *envelope;
char *header;
```

Description

The x4_send call submits a message to Prime X.400 using the information previously stored in the nominated data structure.

The routine returns a pointer to a static *X4_MPDUSTRING* data structure containing the MPDU identifier assigned by Prime X.400.

Returns

If the routine returns a null pointer, then x4_error returns one of the following values:

Value	*Meaning*
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_FIELD_ERROR	One of the envelope or header fields contains illegal characters. The error qualifier contains the structure ID of the field in error.
X4_ERR_INVALID_CHARS	A message field contains characters that are not valid for the attribute. The error qualifier contains the structure ID in error.
X4_ERR_ISC_ERR	An ISC error has occurred. The error qualifier contains the ISC error code.
X4_ERR_MDNP	A mandatory descriptor is missing from the envelope or header structure provided. The error qualifier contains the structure ID of the missing descriptor.

X4_ERR_NO_RESOURCE
Prime X.400 is unable to accept this request. The error qualifier contains the reason for rejection: 1 = Server reconfiguring, 2 = Invalid header or envelope, 3 = X.400 server error.

X4_ERR_NOT_OPEN The user does not have a session open to Prime X.400.

X4_ERR_SYN_ERR An ISC synchronizer error has occurred. The error qualifier contains the synchronizer error code.

X4_ERR_MAX_NESTING Exceeded nesting of forwarded messages. Current max is 5.

APPENDICES

A

NON-C SYNTAX API LIBRARY ROUTINES

Introduction

This appendix lists the PL1 parameter types that correspond to the C parameter types used in the API library routine descriptions in Chapter 3, PRIME X.400 API LIBRARY. It lists the PL1 syntax of each API library routine, and describes three API library routines (described in C) that are used for calling with non-C file units.

Non-C API Library Routines

This section describes three API library routines, described in C, that are used for calling with non-C file units, that is, PRIMOS® file units.

X4P$DECIA5

Function

Decodes an X.409-encoded IA5 text body file to a Prime ECS file.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4p$decia5(tofu, fromfu)

int tofu;
int fromfu;
```

Description

This routine returns **X4_OK** if the operation was successful.

tofu is the PRIMOS file unit (as returned by SRCH$$) of the destination file. *fromfu* is the PRIMOS file unit (as returned by SRCH$$) of the file to be decoded.

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_EXIA5_STR	X.409 IA5 string expected. The error qualifier contains the type found.
X4_ERR_EXOCT_STR	X.409 octet string expected. The error qualifier contains the type found.
X4_ERR_EXSEQ	X.409 sequence expected. The error qualifier contains the type found.

X4_ERR_EXSET	X.409 set expected. The error qualifier contains the type found.
X4_ERR_EXTAG_INT	X.409 tagged integer expected. The error qualifier contains the type found.
X4_ERR_FILE_ERR	A PRIMOS file error has occurred, the qualifier is the PRIMOS error.
X4_ERR_UXSIZE	Unexpected X.409 size. The error qualifier contains the size found.

X4P$DUMP

Function

Produces a formatted diagnostic print of a specified Prime X.400 data structure.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4p$dump(dmpfu, struc)

int dmpfu;
char *struc;
```

Description

This routine produces a formatted listing of the specified Prime X.400 data structure. *dmpfu* is the PRIMOS file unit (as returned by SRCH$$) of a file to which the dump is written.

If *dmpfu* has a value -1, or -2, then the dump is directed as follows:

-1 the dump output is directed to STDOUT
-2 the dump output is directed to STDERR

Returns

The routine returns one of the following values:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_BAD_STRUC	An invalid data structure ID was provided.
X4_ERR_FILE_ERR	A PRIMOS file error has occurred, the qualifier gives the PRIMOS error, or is 0, which indicates that *dmpfu* is an illegal file unit value, that is, dmpfu <-2.

X4P$ENCIA5

Function

Encodes a Prime ECS text file as an X.409 encoded IA5 text body file.

C Syntax

```
#include <x4_struc.h>

#include <x4_error.h>

int x4p$encia5(tofu, fromfu)

int tofu;
int fromfu;
```

Description

This routine returns **X4_OK** if the operation was successful.

fromfu is the PRIMOS file unit (as returned by SRCH$$) of the file to be encoded. *tofu* is the PRIMOS file unit (as returned by SRCH$$) of the destination file (the X.409-encoded IA5 text body file).

Returns

The routine returns the following value:

Value	*Meaning*
X4_OK	The operation was successful.
X4_ERR_FILE_ERR	A PRIMOS file error has occurred, the qualifier is the PRIMOS error.

Parameter Types

This section lists the C parameter types used in the API library routine descriptions in Chapter 3, PRIME X.400 API LIBRARY, and their equivalent PL1 parameter types.

C Types	*PL1 Types*
int	fixed binary(31)
char[n]	CHAR(*)
char*	3-word pointer
int*	ADDR(*variable*)

PL1 Syntax API Library Routines

This section describes how to declare and call each API library routine, using PL1. These routines are described in Chapter 3, PRIME X.400 API LIBRARY.

X4_ACCEPT

PL1 Syntax:

```
dcl x4_accept entry(ptr) returns(fixed bin(31));
```

X4_ALLOC

PL1 Syntax:

```
dcl x4_alloc entry(fixed bin(31), fixed bin(31)) returns (ptr);

dcl struc ptr;

struc = x4_alloc ((id), (rev));
```

Example:

```
struc = x4_alloc((X4_ID_IPM_HEADER), (2));
```

X4_CLEAR

PL1 Syntax:

```
dcl x4_clear entry();

call x4_clear();
```

X4_CLOSE

PL1 Syntax:

```
dcl x4_close entry() returns(fixed bin(31));

dcl status fixed bin(31);

status = x4_close();
```

X4_COPY

PL1 Syntax:

```
dcl x4_copy entry(ptr, ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc1 ptr;
dcl struc2 ptr;

status = x4_copy(struc1, struc2);
```

X4_DECT61

PL1 Syntax:

```
dcl x4_dect61 entry(ptr, ptr) returns fixed bin(31);

dcl err fixed bin(31);
dcl src char(n);
dcl dest char(n);

err = x4_dect61(ADDR(dest), ADDR(src));
```

X4_DRNOTIFY

PL1 Syntax:

```
dcl x4_drnotify entry(ptr, ptr) returns(ptr);

dcl struc1 ptr;
dcl pid ptr;
```

```
struc2 - x4_drnotify(pid, struc1);
```

X4_ENCHAIN

PL1 Syntax:

```
dcl x4_enchain entry(ptr, ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc1 ptr;
dcl struc2 ptr;

status = x4_enchain(ADDR(root), struc2);
```

Example:

```
status = x4_enchain(ADDR(struc1->X4_ORNAME.stdatt.orgunit), struc2);
```

X4_ENCT61

PL1 Syntax:

```
dcl x4_enct61 entry(ptr, ptr, fixed bin(31)) returns fixed bin(31);

dcl err fixed bin(31);
dcl src char(n);
dcl dest char(n);

err = x4_enct61(ADDR(dest), ADDR(src),(n));
```

X4_ERROR

PL1 Syntax:

```
dcl x4_error entry(ptr, ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl ecode fixed bin(31);
dcl secode fixed bin(31);

status = x4_error(ADDR(ecode), ADDR(secode));
```

X4_FIND

PL1 Syntax:

```
dcl x4_find entry(ptr, fixed bin(31)) returns(ptr);
```

```
dcl struc1 ptr;
dcl struc2 ptr;

struc2 = x4_find(struc1, (key));
```

Example:

```
struc2 = x4_find(struc1, (X4_K_NEXT));
```

X4_GET

PL1 Syntax:

```
dcl x4_get entry(ptr, fixed bin(31)) returns(ptr);

dcl struc1 ptr;
dcl struc2 ptr;

struc2 = x4_get(struc1 (key));
```

Example:

```
struc2 = x4_get(struc1, (X4_K_CC));
```

X4_GETGDI

PL1 Syntax:

```
dcl x4_getgdi entry returns(ptr);

dcl gdi ptr;

gdi = x4_getgdi();
```

X4_GETMTA

PL1 Syntax:

```
dcl x4_getmta entry returns(ptr);

dcl mta ptr;

mta = x4_getmta();
```

X4_INIT

PL1 Syntax:

```
dcl x4_init entry(ptr, fixed bin(31), fixed bin(31)) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc1 ptr;

status = x4_init(struc1, (id), (rev));
```

Example:

```
status = x4_init(struc1, (X4_ID_RECIPIENT), (2));
```

X4_KILL

PL1 Syntax:

```
dcl x4_kill entry(ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc ptr;

status = x4_kill(struc);
```

X4_LOGOFF

PL1 Syntax:

```
dcl x4_logoff entry(ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl pid ptr;

status = x4_logoff(pid);
```

X4_LOGON

PL1 Syntax:

```
dcl x4_logon entry(ptr, ptr, fixed bin(31)) returns(ptr);

dcl pid ptr;
dcl user char(n);
dcl dir char(n);

pid = 4_logon(ADDR(user), ADDR(dir), (mode));
```

Example:

```
pid = x4_logon(ADDR(user), ADDR(dir), (X4_K_RECEIVE));
```

X4_OPEN_GWI

PL1 Syntax:

```
dcl x4_open_gwi entry(ptr) returns(ptr);

dcl pid ptr;
dcl server char(n);

pid = x4_open_gwi(ADDR(server));
```

X4_OPEN_UAI

PL1 Syntax:

```
dcl x4_open_uai entry(ptr, fixed bin(31)) returns(ptr);

dcl pid ptr;
dcl server char(n);
dcl retired fixed bin(31);

pid = x4_open_uai(ADDR(server), (retired));
```

X4_PROBE

PL1 Syntax:

```
dcl x4_probe entry(ptr) returns(ptr);

dcl struc1 ptr;
dcl struc2 ptr;

struc2 = x4_probe(struc1);
```

X4_PUT

PL1 Syntax:

```
dcl x4_put entry(ptr, fixed bin(31), ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc1 ptr;
dcl struc2 ptr;
```

```
status = x4_put(struc1, (key), struc2);
```

Example:

```
status = x4_put(struc1, (X4_K_TO), struc2);
```

X4_READ

PL1 Syntax:

```
dcl x4_read entry(fixed bin(31)) returns(ptr);

dcl delay fixed bin(31);
dcl struc1 ptr;

struc1 = x4_read((delay));
```

X4_REJECT

PL1 Syntax:

```
dcl x4_reject entry(ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl pid ptr;

status = x4_reject(pid);
```

X4_RELEASE

PL1 Syntax:

```
dcl x4_release entry(ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl struc1 ptr;

status = x4_release(struc1);
```

X4_REPLY

PL1 Syntax:

```
dcl x4_reply entry(ptr, ptr, ptr) returns(ptr);

dcl pid ptr;
dcl struc1 ptr;
dcl struc2 ptr;
dcl struc3 ptr;
```

```
struc3 = x4_reply(pid, struc1, struc2);
```

X4_SEND

PL1 Syntax:

```
dcl x4_send entry(ptr, ptr, ptr) returns(ptr);

dcl pid ptr;
dcl struc1 ptr;
dcl struc2 ptr;
dcl struc3 ptr;

struc3 = x4_send(pid, struc1, struc2);
```

X4P$DECIA5

PL1 Syntax:

```
dcl x4p$decia5 entry(fixed bin(31), fixed bin(31)) returns(fixed bin(31));

dcl status fixed bin(31);
dcl fileunitto fixed bin(31);
dcl fileunitfrm fixed bin(31);

status = x4p$decia5((fileunitto), (fileunitfrm));
```

where:

fileunitto	PRIMOS file unit as returned by SRCH$$ of destination file
fileunitfrm	PRIMOS file unit as returned by SRCH$$ of source file

X4P$DUMP

PL1 Syntax:

```
dcl x4p$dump entry(fixed bin(31), ptr) returns(fixed bin(31));

dcl status fixed bin(31);
dcl fileunit fixed bin(31);
dcl struc ptr;

status = x4p$dump((fileunit), struc);
```

where:

fileunit	PRIMOS file unit as returned by SRCH$$ or -1 for C's STDOUT or -2 for C's STDERR if -1 or -2 are used, pathname is ignored
struc	A pointer to the structure to be dumped

X4P$ENCIA5

PL1 Syntax:

```
dcl x4p$encia5 entry(fixed bin(31), fixed bin(31)) returns(fixed bin(31));

dcl status fixed bin(31);
dcl fileunitto fixed bin(31);
dcl fileunitfrm fixed bin(31);

status = x4p$encia5((fileunitto), (fileunitfrm));
```

where:

fileunitto	PRIMOS file unit as returned by SRCH$$ of destination file
fileunitfrm	PRIMOS file unit as returned by SRCH$$ of source file

B

EXAMPLE APPLICATION PROGRAM TO SEND A MESSAGE

Introduction

The following code sends an X.400 message using the Prime X.400 API.

```
/* SEND.C,
   Send a simple mail message
   Copyright (c) 1989, Prime Computer, Inc., Natick, Ma 01760 */

/* TITLE : SEND - Send an X.400 message using the PRIME X.400 API */

/* DESCRIPTION : Example of using the X400 API to send an X400 mail message.

      This example is reasonably robust code, and in particular takes
      care with string handling. Error handling and reporting has been
      kept to a minimum for clarity of exposition of the features of
      using the API.

      Essential information is prompted from the user, but other
      parameters are set to inbuilt defaults: these may not be
      appropriate for any given application, other than as an
      example.

      Only a subset of the X.400 functionality available with the API
      is utilized.
*/

/* START-CODE */

#define TRUE 1
#define FALSE 0
#define TERMBUF 80 /* Max characters safely read from terminal */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <x4_keys.h>
#include <x4_struc.h>
#include <x4_error.h>

/* Global variables for current compilation unit */

static char linebuff[TERMBUF+1]; /* buffer for reading from terminal */
static char strbuff[TERMBUF+1];  /* buffer for output from -
static int debug = FALSE;                    input string conversion */

main ()
{
   X4_IPM_HEADER *hdr;
   X4_IPM_ENVELOPE *env;
   X4_MPDUSTRING *mid;
   X4_MSG *logon_ptr;
   char local_name[X4_SZ_LOCALNAME+1];
   char encodedmailfile[X4_SZ_FILENAME+1];
   int code, qual;
   int rtncode;
   extern cleanup();


   /* Ask the user if this is a debugging session, if so this program
    * uses x4_dump periodically to show the built structures.
    */
    {
       char *dbgflg[2];
         qry_readstr("\nIs this a debugging session ? [Y|N]: ", dbgflg, 1);
       if ( toupper(dbgflg[0]) == 'Y' )
          debug = TRUE;
    }
```

```
/*
 * Initiate session with server process
 */
x4_clear();
if ( (char *) x4_open_uai("", 1) == NULL )
   goto label_open_uai_err;

/*
 * Prompt for localname and logon to UA component
 */
qry_readstr("\nEnter localname: ", local_name, X4_SZ_LOCALNAME);

if ( (char *) ( logon_ptr =
   (X4_MSG *)  x4_logon (local_name, "", X4_K_RECEIVE | X4_K_SEND) ) == NULL )
   goto label_logon_err;

/* Allocate the Send (root) structures */

hdr = (X4_IPM_HEADER *) x4_alloc(X4_ID_IPM_HEADER, 1);
hdr->struc.valdata = TRUE;

env = (X4_IPM_ENVELOPE *) x4_alloc(X4_ID_IPM_ENVELOPE, 1);
env->struc.valdata = TRUE;

env->content_type.struc.valdata = TRUE;
env->content_type.value = X4_CT_P2;

/* Give the message a unique stamp */

{
   char unique_str[X4_SZ_UACONTENTID+1];

   qry_readstr("\nEnter Reference: ", unique_str, X4_SZ_UACONTENTID);

   strncpy1(env->ua_content_id.string, unique_str, X4_SZ_UACONTENTID);
   env->ua_content_id.struc.valdata = TRUE;
   if (debug)
      x4_dump(stdout, &env->ua_content_id);

   strncpy1(hdr->reference.ipstr.string, unique_str,X4_SZ_IPSTRING);
   hdr->reference.struc.valdata = TRUE;
   hdr->reference.ipstr.struc.valdata = TRUE;
   if (debug)
      x4_dump(stdout, &hdr->reference);
}

/* Put in a subject.. */
{  char subject[X4_SZ_SUBJECT+1];

   qry_readstr("\nEnter Subject: ", subject, X4_SZ_SUBJECT);
   strncpy1(hdr->subject.string, subject, X4_SZ_SUBJECT);
   hdr->subject.struc.valdata = TRUE;
   if (debug)
   {
      fprintf(stdout,"Done subject\n");
      x4_dump(stdout,hdr);
   }
}

/* Say who this message is from using freeformname */
{  char nicename[X4_SZ_FREEFORMNAME+1];
      X4_ORDESCRIPTOR *originator =
            (X4_ORDESCRIPTOR *) x4_alloc(X4_ID_ORDESCRIPTOR, 1);
```

```
    qry_readstr("\nEnter Originators usual name: ", nicename, X4_SZ_FREEFORMNAME);
    strncpy1(originator->name.string, nicename, X4_SZ_FREEFORMNAME );
    originator->struc.valdata = TRUE;
    originator->name.struc.valdata = TRUE;

    x4_put(hdr, X4_K_FROM, originator);
    if (debug)
    {
       fprintf(stdout,"Added originator name\n");
       x4_dump(stdout,hdr);
    }
}
/*
 * Fill in Recipient details.
 */
{  X4_ORNAME *orname;

   /* Prompt for Recipient O/R name details, and obtain pointer to completed
      ORNAME structure in variable orname */
   put_p1_recipient(env, &orname);

   /* Construct P2 recipient fields using pointer to P1 ORNAME */
   put_p2_recipient(hdr, orname);
}

/* Mandatory and default parameters that must be present for server */

{ X4_PRIORITY *priority = (X4_PRIORITY *)x4_alloc(X4_ID_PRIORITY,1);
  priority -> struc.valdata = TRUE;
  priority -> value = X4_P_URGENT;
  x4_put(env, X4_K_PRIORITY, priority);
  x4_clear();
}
{ X4_IMPORTANCE *importance = (X4_IMPORTANCE *) x4_alloc(X4_ID_IMPORTANCE,1);
  importance-> struc.valdata = TRUE;
  importance -> value = X4_IMP_HIGH;
  x4_put(hdr, X4_K_IMPORTANCE, importance);
  x4_clear();
}
{ X4_SENSITIVITY *sensitivity = (X4_SENSITIVITY *) x4_alloc(X4_ID_SENSITIVITY,1);
  sensitivity -> struc.valdata = TRUE;
  sensitivity -> value = X4_SEN_PERSONAL;
  x4_put(hdr, X4_K_SENSITIVITY, sensitivity);
  x4_clear();
}


/* Encode message */
{ FILE *unfp;  /* normal ascii text file — can be the terminal */
  FILE *enfp;  /* x409 encoded bodyfile */
  char mailfile[TERMBUF+1];

  qry_readstr("\nEnter mail file name: ", mailfile, X4_SZ_FILENAME);
  qry_readstr("\nEnter encoded file name: ", encodedmailfile, X4_SZ_FILENAME);

  if ( (unfp = fopen(mailfile, "r")) == NULL)
        exit();
  if ( (enfp = fopen(encodedmailfile, "w")) == NULL)
        exit();
  x4_encia5(enfp, unfp);
  x4_clear();
  fclose(unfp);
  fclose(enfp);
```

```
    }

    /* Put in message body */
    {
       X4L_BODY *bodyPtr = (X4L_BODY *) x4_alloc(X4L_ID_BODY, 1);

       strcpy(bodyPtr->value.filename.string, encodedmailfile);
       bodyPtr->struc.valdata = TRUE;
       bodyPtr->value.struc.valdata = TRUE;
       bodyPtr->value.part.struc.valdata = TRUE;
       bodyPtr->value.part.value = X4_BT_IA5TEXT;
       bodyPtr->value.filename.struc.valdata = TRUE;

       x4_enchain(&hdr->body_list, bodyPtr);
    }

    /* Indicate the type of body in envelope */

    /* Set third bit in bitstring for IA5text — see X.411 Para 3.4.1.4 */
    env->encoded.bodytype.value = 0x20000000;
    env->encoded.struc.valdata = TRUE;
    env->encoded.bodytype.struc.valdata = TRUE;

    x4_clear();

    if (debug)
    {
       fprintf(stdout,"\n\nDUMP OF ENVELOPE\n");
       x4_dump(stdout,env);
       fprintf(stdout,"\n\nDUMP OF HEADER\n");
       x4_dump(stdout,hdr);
    }

    /* Send the message */

    mid = (X4_MPDUSTRING *) x4_send(logon_ptr, env, hdr);
    if ((mid != NULL) && (mid -> struc.valdata))
       fprintf(stdout,"Your Message ID is %s\n",  mid -> string);
    else
       goto label_send_err;

    rtncode = 0;
label_return:
    x4_release(env);
    x4_release(hdr);
    x4_clear();
    if (logon_ptr != NULL)
       x4_logoff(logon_ptr);
    x4_close();
    return(rtncode);

/*
 * Exception Handling.
 */
label_send_err:
    fprintf(stderr,"send :"); goto label_seterr;
label_open_uai_err:
    fprintf(stderr,"open :"); goto label_seterr;
label_logon_err:
    fprintf(stderr,"logon :"); goto label_seterr;
label_seterr:
    printf("Failed !;");
    if (x4_error(&code, &qual))
```

```
            fprintf(stderr,"Status %d Qual %d\n", code, qual);
        rtncode = code;
        goto label_return;
    }

    /* put_p1_recipient */

    /*
     * Put a recipient O/R name into the envelope.
     */

    put_p1_recipient(xenv, xxorname)
        char *xenv;
        X4_ORNAME **xxorname;
    {
        X4_P1RECIPIENT *recipient =
                (X4_P1RECIPIENT *) x4_alloc(X4_ID_P1RECIPIENT, 1);

        recipient->struc.valdata = TRUE;
        recipient->orname.struc.valdata = TRUE;
        recipient->orname.stdatt.struc.valdata = TRUE;

        /* Server requires whole of O/R name variant 1 (if used) */
        /* COUNTRY, ADMD, PRMD, ORGANISATION , UNIT, PERSONAL NAME */

        qry_readstr("\Enter CountryName: ",
            recipient->orname.stdatt.cname.string, X4_SZ_COUNTRYNAME);
        recipient->orname.stdatt.cname.struc.valdata = TRUE;

        qry_readstr("\Enter ADMD: ",
            recipient->orname.stdatt.admd.string, X4_SZ_ADMD);
        recipient->orname.stdatt.admd.struc.valdata = TRUE;

        qry_readstr("\nEnter PRMD: ",
            recipient->orname.stdatt.prmd.string, X4_SZ_PRMD);
        recipient->orname.stdatt.prmd.struc.valdata = TRUE;

        qry_readstr("\nEnter OrganizationName: ",
            recipient->orname.stdatt.orgname.string, X4_SZ_ORGNAME);
        recipient->orname.stdatt.orgname.struc.valdata = TRUE;

        { /* Allocate and chain an orgunit list item to the root orgunit */
          X4L_ORGUNIT *lorgunit = (X4L_ORGUNIT *) x4_alloc(X4L_ID_ORGUNIT, 1);

          qry_readstr("\Enter Oranization Unit: ",
              lorgunit ->value.string, X4_SZ_ORGUNIT);

          /* Set the list as valid */
          lorgunit ->struc.valdata = TRUE;

          /* set the value of the list item as valid */
          lorgunit ->value.struc.valdata = TRUE;

          recipient->orname.stdatt.orgunit.struc.valdata = TRUE;

          x4_enchain(&(recipient->orname.stdatt.orgunit),lorgunit);
          if (debug)
              x4_dump(stdout,&(recipient->orname.stdatt.orgunit));
        }

        qry_readstr("\nEnter Surname: ",
            recipient->orname.stdatt.name.surname.string, X4_SZ_SURNAME);
            recipient->orname.stdatt.name.struc.valdata = 1;
```

```
        recipient->orname.stdatt.name.surname.struc.valdata = 1;

    recipient->extension.value = 1;
    recipient->extension.struc.valdata = TRUE;

    /* PerRecipientFlag see Figure 19/X.411 */
    /* Bit 0: Responsibility On - 1 */
    /* Bits 1-2: Report Requested: Audit-And-Confirmed - 11 */
    /* Bits 3-4: User Report Request - Confirmed - 10 */
    recipient->per_recipient_flag.value = 0xF0000000;
    recipient->per_recipient_flag.struc.valdata = TRUE;

    x4_put(xenv, X4_K_RECIPIENT, recipient);

    *xxorname = &(recipient-> orname);
}

/* put_p2_recipient */

/*
 * Put a recipient O/R name into the Send header.
 * Refer to the P2 definition, in figure 3/X.420, for details of the
 * Recipient fields, e.g. ORDescriptor.
 * If either of reportRequest or replyRequest are selected then the
 * O/R Descriptor must contain an O/R name.
 */

put_p2_recipient(xhdr, xorname)
    X4_IPM_HEADER *xhdr;
    X4_ORNAME *xorname;
{
    X4_RECIPIENT *rcp =
              (X4_RECIPIENT *) x4_alloc(X4_ID_RECIPIENT, 1);

    /* Recipient.ORDescriptor */

    x4_copy(&(rcp->ordescriptor.orname), xorname);
    rcp->struc.valdata = TRUE;
    rcp->ordescriptor.struc.valdata = TRUE;

    /* Recipient.reportRequest: Set First bit for receiptNotification */
    rcp->request.value = 0x80000000;
    rcp->request.struc.valdata = TRUE;

    /* Recipient.replyRequest: Set boolean to true */
    rcp->reply.value = TRUE;
    rcp->reply.struc.valdata = TRUE;

    x4_put(xhdr, X4_K_TO, rcp);
}

/* strncpy1 */

/* strncpy1: version of strncpy that guarantees a null terminated result,
             by assuming that the to buffer is one byte larger than the
             size specified by the maxlen argument */
strncpy1( to, from, maxlen)
char *to;
char *from;
unsigned int maxlen;
{
        char *lim;
```

```
        lim = to + maxlen;
        while ( (to < lim) && (*to++ = *from++) );
        *to = '\0';
}

/* qry_readstr */

qry_readstr(prompt, string, max)
char *prompt;
char *string;
int max;
{  char *nl;
   fprintf(stdout, prompt);
   if (fgets(strbuff, sizeof(strbuff),stdin) == NULL)
       kill(0, SIGTERM);
   if ( (nl = strchr(strbuff,'\n')) != NULL)
      *nl = '\0';
   strncpy1(string,strbuff,max);
   if (debug)
   {
      fprintf(stdout,"%s\n",string);
   }
}
/* END-CODE */
```

C

X.400 API LIBRARY ROUTINE RETURN VALUES

Introduction

This appendix lists the return values of each of the X.400 API library routines.

X4_OK (0)
The operation was successful.

X4_ERR_BAD_COPY (19)
Incompatible data structures for copy.

X4_ERR_BAD_KEY (4)
The user has specified an invalid key.

X4_ERR_BAD_MESSAGE (33)
An invalid message format has been received.

X4_ERR_BAD_RESPONSE (36)
An unrecognized message type from the Prime X.400 server.

X4_ERR_BAD_REV (12)
An invalid data structure version was provided.

X4_ERR_BAD_STRUC (3)
An invalid unknown data structure ID was provided.

X4_ERR_END_OF_LIST (18)
There are no more items in this list.

X4_ERR_EXIA5_STR (50)
X.409 IA5 string expected. The error qualifier contains the X.409 type found.

X4_ERR_EXOCT_STR (49)
X.409 octet string expected. The error qualifier contains the X.409 type found.

X4_ERR_EXSET (46)
X.409 set expected. The error qualifier contains the X.409 type found.

X4_ERR_EXSEQ (45)
X.409 sequence expected. The error qualifier contains the X.409 type found.

X4_ERR_EXTAG_INT (47)
X.409 tagged integer expected. The error qualifier contains the X.409 type found.

X4_ERR_FIELD_ERROR (51)
One of the envelope or header fields contains illegal characters. The error qualifier contains the structure ID of the field in error.

X4_ERR_FILE_ERR (27)
A file system error has occurred. The error qualifier contains the PRIMOS® error code.

X4_ERR_INVALID_CHARS (52) (54)
Invalid characters in message or string.

X4_ERR_ISC_ERR (26)
An Inter Server Communication (ISC) error has occurred. The error qualifier contains the ISC error code as defined in SYSCOM>ISC_KEYS.H.INS.CC

X4_ERR_LIST_EMPTY (17)
There is no data item present for this list.

X4_ERR_LOGGED_ON (34)
The user is already logged on.

X4_ERR_MDNP (41)
A mandatory descriptor is missing from the envelope data structure provided. The error qualifier contains the structure ID of the missing descriptor, for example, *X4_ID_GLOBALDOMAINID.*

X4_ERR_NAC (39)
The user does not have access rights to this Prime X.400 user name.

X4_ERR_NO_DATA (20)
No data present/available.

X4_ERR_NO_MESSAGE (10)
There is no message waiting.

X4_ERR_NO_READ (11)
The user does not have an unanswered x4_read request.

X4_ERR_NO_RESOURCE (6)
Insufficient resources to process request.

X4_ERR_NOT_GWI (52)
A communication path to a Prime X.400 server has already been established using the x4_open_uai call.

X4_ERR_NOT_OPEN (1)
A session is not open to the Prime X.400 server.

X4_ERR_OPEN (7)
The user already has a path open to Prime X.400.

X4_ERR_RECONFIGURING (35)
The Prime X.400 server is reconfiguring.

X4_ERR_SYN_ERR (32)
An ISC Synchronizer error has occurred. The error qualifier contains the synchronizer error code as defined in SYSCOM>SYNC_CODES.H.INS.CC

X4_ERR_TERMINATED (2)
Prime X.400 has closed down.

X4_ERR_TOO_LONG (53)
The resulting T.61 string is longer than the maximum specified by *maxlen.*

X4_ERR_UNKNOWN_USER (5)
The user name is not present in the configuration file being used by Prime X.400.

X4_ERR_UXSIZE (48)
Unexpected X.409 size. The error qualifier contains the size found.

INDEX

INDEX

N

O

P

R

S

T

U

X

SURVEYS

READER RESPONSE FORM

X.400 Programmer's Guide
DOC11277-1LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760