USNET Internet Access Package User's Guide

Version 1.1
July 2004

U S SOFTWARE®
EMBEDDED EXCELLENCE

Copyright and Trademark Information

Lantronix, Inc.
15353 Barranca Parkway
Irvine, CA 92618
(949)453-3990
Fax (949) 453-3995

For Support Contact:
Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxinfo.com
www.smxinfo.com

Documentation Conventions

Computer output and code examples: `Courier`, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (*`mt_busy`*).

File names: Times bold (the file **usrclk.asm**), usually in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Documentation History

Revision Number	Date
1.0 (Original)	August 1997
1.1	April 1998

Contents

1. IAP Overview

Overview

The USNET® Internet Access Package (IAP) provides modules for USNET to support dial-up connections, Domain Name System (DNS), Hypertext Transfer Protocol (HTTP), and Internet mail. This manual gives detailed information on the functions that are provided. The files that make up the package and the functions that these files provide are detailed in the **readme.txt** file.

This manual describes the three products included in the USNET Internet Access Package: Automatic dialing, Domain Name System, and e-mail. This is the organization of the manual:

Chapter	**Contents**
1. Overview	Introduces the reader to the Internet Access Package, IAP terminology, and recommended reading.
2. Automatic Dialing	Describes configuring and using the dialer.
3. Domain Name System	Describes the Domain Name System.
4. E-Mail	Describes sending and receiving messages using SMTP and POP protocols, decoding data, and testing.

Any connections between the web server and e-mail (such as automatically e-mailing log files or notices) are configured by the user.

Chapter 1

IAP Terminology

CGI	Common Gateway Interface. CGI reads parameters from forms on the displayed web page to the server, so the server can display different pages depending on the user's actions.
DHCP	Dynamic Host Configuration Protocol, a method for a client to request information on its own configuration from a server.
DNS	Domain Name System, a mechanism that allows the IP address of a system in a TCP/IP network to be determined based on a name assigned to the system, or vice versa.
HTML META commands	Commands embedded in the HTML that return predefined system information to the user.
HTTP	Hypertext Transfer Protocol, a simple application- level protocol used to access hypermedia documents. The protocol is stateless and generic, which allows it to be used for many tasks.
ISMAP	An HTML tag which returns position coordinates within the page image.
MIME	Multipurpose Internet Mail Extensions, which defines how to encode and decode multipart messages and non-ASCII character sets.
POP	Post Office Protocol, a minor variation of SMTP that allows a client to retrieve mail from a remote server mailbox.
PPP	Point-to-Point Protocol, a link between two computer ports.
SLIP	Serial Line Interface Protocol, a link between two points (computer ports).
SMTP	Simple Mail Transfer Protocol, a protocol for transferring mail.
SVA	Server Variable Access, a mechanism for accessing static global variables within an embedded application via HTML.
TCP/IP	Transmission Control Protocol/Internet Protocol, a software protocol for communication between computers.

Recommended Reading

Other U S Software Documents

USNET User's Manual

USNET Web Server User's Guide

Advanced Customization of the Embedded Web Server

On the Internet

RFCs (request for comments) are documents that are available over the Internet via anonymous FTP. The following references will provide more information on topics relevant to IAP:

Topic	RFC Numbers
SMTP	821, 822, 1869, and 2045
POP	1725
MIME	2045 through 2049
HTTP	2068
DNS	1034, 1982, 2065, 1876, 1101

Here is an abbreviated example FTP session:

```
% ftp ds.internic.net
.
Name: anonymous
Password: <your email address>
.
ftp> cd rfc
.
ftp> get rfc1122.txt
.
ftp> quit
```

Books

Foundations of WWW Programming with HTML & CGI
IDG Books
ISBN 1-56884-703-3

CGI Programming in C and Perl
Thomas Boutell
Addison Wesly
ISBN 0-201-42219-0

CGI Developers Guide
Eugene Eric Kim
Sams Net
ISBN 1-57521-087-8

There are many books on web page design. This one is very good for low-level protocols, and has cross-references to RFCs:

Internet Protocols Handbook
Dave Roberts
Coriolis Group Books
ISBN 1-883577-88-8

2. Automatic Dialing

Dialing Overview

The dialing function provided by the IAP is useful for applications where a modem is used to establish a point-to-point connection with another host. The base distribution of USNET is capable of establishing a point-to-point connection between directly connected systems. When a modem is involved in establishing the connection, support must be provided to issue commands to the modem, and possibly to respond to queries from a terminal session before the PPP handshake can begin. The ***Ndial()*** function in **dial.c** provides this support.

This support is especially useful in connecting a system using a PPP account available from many Internet Service Providers.

Ndial()

Dials or hangs up the modem.

```
int Ndial(int netno, char *phonenumber);
```

netno	Network index: 0, 1 and so on.
phonenumber	The number to dial, in any format that is acceptable to the auto-dialer. A zero argument means disconnect.

Ndial() can be used for PPP. ***Ndial()*** is normally not called by the USNET protocol stack, because USNET does not know how to get the telephone number. There is however a way to force this for testing purposes:

1. To use ***Ndial()*** in a PPP connection, set the flag field in **netconf.c** for the serial port to `DIAL`. For instance:

```
“test”, “com2”, C, {192,9,202,1}, EA0, DIAL, PPP, I8250, 0,
“IRNO=3 PORT=0x2f8 CLOCK=115200 BAUD=38400”,
```

2. Define the telephone number, for instance in **local.h**:

```
#define TESTPHONE “5551212”
```

Other than for testing purposes, the dialing and the disconnect would be done by the application.

Return Value

0	Success
-1	Error

Example

Here is an example of how the ***Ndial()*** function is used. The example is from **PPP.C** (included in the core USNET product).

```
#define TESTPHONE “5551212”   /*  Phone number */
#ifdef TESTPHONE
/*
/*  for testing purposes, call the dialing function */
*/
    if (netconf[netp->confix].flags & DIAL)
    {
        i1 = Ndial(netno, TESTPHONE);
        if (i1 < 0)
            return i1;
    }
#endif /* TESTPHONE */
```

The ***TESTPHONE*** macro is already defined, and the user must add this line to **local.h**:

```
#define TESTPHONE “5551212”
```

Configuring the Dialing

The **dial.c** file has three command tables at the top of the code. Table `upcommand` contains the commands needed to establish a telephone connection. The table as shipped should work for Hayes-compatible autodialers in North America. For other devices and other telephone networks, changes may be needed. The table has three elements:

- Command sent to the autodialer, exactly as coded.
- Answer expected from the autodialer, without terminator. ***Ndial()*** accepts either `0x0A` or `0x0D` as terminator, and ignores extra terminators.
- Flags:

`0x01`	Skip on first try, meant for reset commands
`0x02`	Long timeout, needed for the actual dialing
`0x04`	Phone number included in ***sprintf()*** format
`0x08`	Delay before and after
`0x40`	Skip this entry = comment out
`0x80`	Final entry in table

Table `downcommand` is used exactly like `upcommand`, but to disconnect.

Table `logincommand` is meant for servers that check user ID and password before actually entering SLIP or PPP. This practice, while poorly standardized, is widely used in UNIX, and has also spread to other systems. The table format is the same as for the two others, but the use is different:

- A character string that ***Ndial()*** will watch for. The string does not have to be an entire word; actually, it may be better not to include the first letter of the word.
- Answer that ***Ndial()*** will send upon seeing the expected string.
- Flags:

`0x40`	Skip this entry = comment out
`0x80`	Final entry in table

This is the `logincommand` structure:

```
/*
/* use 4-8 characters of the question in this table */
*/
static struct COMMAND logincommand[]={
        {“ogin”, USERID, 0x00},
        {“assword”, PASSWD, 0x80},
};
```

The `logincommand` structure is defined in **ndial.c** and has two entries, `ogin` and `assword` (the first letters are dropped). The application waits for the first string, then responds with the next parameter. The last string is a flag.

This means that the server expects to be asked two questions, such as “please enter your login” and “enter password”. The user ID and password are defined in **local.h** (macros ***USERID*** and ***PASSWD)***. These questions will vary from server to server, and often include a greeting message. If you have no idea what your server will ask, just run ***Ndial()*** with `NTRACE` set to 3, and this will show you what to expect.

If you don’t need the `logincommand` table at all, comment out the entries using flag value `0x40`. Many servers will use the PPP authentication protocols, usually PAP, to validate the user.

3. Domain Name System

DNS Overview

The Domain Name System (DNS) is a mechanism that allows the IP address of a system in a TCP/IP network to be determined based on a name assigned to the system. Referring to a system by a name rather than an IP address allows for friendlier user interfaces, and also provides a layer of indirection that can be used to keep a system's name consistent, even though its IP address may need to be changed.

The **dns.c** module provides a DNS resolver function that will accept the name of a host as a parameter, and return the IP address that is associated with the name.

This function depends on the support of at least one DNS server, which will respond to queries from the resolver to provide the name to IP address mapping. Up to two entries for DNS servers can be entered in the `netdata[]` array in **netconf.c**. The servers are identified by the flag `DNSVER` in the `flags` field of an entry, for example:

```
   “dns1”, “nnet”, C, {192,168,43,21}, EA0, DNSVER, 0, 0, 0, 0,
```

If USNET is compiled to include DHCP support, then DHCP can provide the IP addresses of DNS servers without explicitly entering the servers in `netdata[]`.

The DNS resolver can be invoked automatically as part of calls to ***Nopen()*** or ***gethostbyname()*** when a name given as a parameter is not defined in `netdata[]`. To provide this feature, the constant `DNS` should be set to the value `2` in **local.h**. For example:

```
/* ======================================================

   DNS resolver, 1 = code included, 2 = called
   automatically. */

#define DNS 2
```

If DNS is defined as 1, then DNS-related code will be included in the USNET library functions, but the DNS resolver function will not be called automatically to resolve unknown host names.

Applications can also call the DNS resolver directly using the ***DNSresolve()*** function (described next).

DNSresolve()

Resolves a domain name to an IP address.

```
int DNSresolve(char *fullname, IPaddr *iidp);
```

fullname	domain name
iidp	pointer to the address of the returned IP address

DNSresolve() stores the IP address at this location if *fullname* is non-zero.

DNSresolve() can start with either a domain name or IP address. If there's an @ in the name, ***DNSresolve()*** tries to find a mail host (IP address). If the first letter in the name is between 0 and 9, it's a pointer to an IP address, and ***DNSresolve()*** tries to find the domain name.

Return Value

>= 0	Successful lookup
-1	IP address could not be obtained from the DNS server(s)
ENOBUFS	Not enough buffers available for query (defined in **support.h**)

Example

```
IPaddr ipa;
char *hostname;

hostname="localhost";
stat = DNSresolve(hostname,ipa);
if (stat<0)
      ERROR();
```

4. E-Mail

Overview: SMTP and POP, with MIME Support

IAP uses two mail protocols: Simple Mail Transfer Protocol (SMTP) and Post Office Protocol (POP). SMTP is an application protocol based on TCP. It is used for moving mail from one machine to another machine, while POP allows a user to read their mail from a host machine. Both SMTP and POP can be separated into the receiving side (server) and the sending side (client). IAP supports SMTP as a server or a client, and POP as a client only.

See also: *Recommended Reading* in Chapter 1 for references for SMTP, POP, and MIME.

MIME (Multipurpose Internet Mail Extensions) defines how to encode and decode multipart messages and non-ASCII character sets. USNET programs are mime-aware (can handle mime encoding).

The Internet Access Package supports encoding and decoding base64, and supports decoding quoted extended ASCII.

The USNET support for SMTP, POP, and MIME consists of user-callable subroutines to send and receive mail messages.

Using SMTP to Send Mail

To send a message with SMTP, the client connects to an SMTP server and then transfers the messages.

This is an example of the client SMTP flow of function calls:

```
SMTPsend()                       /* client calls IAP
   SMTPgetdata(NULL) /* IAP calls client to open file
   SMTPgetdata(data) /* IAP calls client to read data
   etc…
                                 /* SMTPsend() returns to client
```

Internally, sending a mail message using SMTP is done with the following sequence:

1. Open TCP port 25.
2. Send command ***MAIL FROM.***
3. Send command ***RCPT TO.***
4. Send command ***DATA.***
5. Send the message.
6. Send a dot in its own line as terminator.
7. Send command ***QUIT.***
8. Close connection.

The message itself is in the SMTP format.

The basic message consists of an envelope (to and from), headers, and text. The headers give information such as the date, the subject, and so on. They are separated from the text body by an empty line.

These functions for sending a message are described in this section:

SMTPgetdata()	Provides mail contents (client-provided, IAP calls).
SMTPsend()	Sends a message (in IAP, client calls).

SMTPgetdata()

Provides mail contents.

```
int SMTPgetdata(char *buff, int buflen);
```

buff	a pointer to the buffer where the data goes
buflen	Values are:

0 =	a new message, or message part, is starting, and the user should open his data file. See also *Return Value* below.
>0 =	a request for up to `buflen` bytes of data into the buffer `buff`. If this is a text (ASCII) file, lines must end in CR-LF.

SMTPsend() calls this user routine to get data for the mail message.

For single-part messages, ***SMTPgetdata()*** will be called once with `buflen=0` to trigger a file open, and then many times with `buflen>0` to obtain data. ***SMTPsend()*** will continue calling ***SMTPgetdata()*** until data is exhausted, at which time ***SMTPgetdata()*** will close its file and return 0.

For multipart messages, ***SMTPgetdata()*** can indicate that the data should be encoded using the base64 representation by returning 1 when asked to open the file. It will return -1 if there are no further parts.

Return Value

With `buflen` of 0 (new message or part), ***SMTPgetdata()*** should return:

0	Text part or single-part message
1	Binary part
-1	No more parts

If `buflen` is >0, *SMTPgetdata()* should return the number of bytes placed into the buffer.

Chapter 4

Examples

```
/* The following example assumes single-part ASCII = non-MIME
/* We use file I/O here, but data from memory is also possible
*/

    unsigned **SMTPgetdata**( char* BfrAdr, unsigned BfrLen )
    {
    static FILE* F = 0;
    if( 0==BfrLen ){
      /* starting a new part, so open file */
      F = fopen("myfile.txt");
      return (F?0:-1);
    }else{                              /* reading more data */
      got = fgets(BfrAdr,BfrLen-1,F);
                                        /* get 1 line */
      if( got ){                        /* fix \n to be CR+LF */
         char* t = strlen(BfrAdr);
         if( t ) -t;
                                        /* back up to the \n */
         *t++ = '\x0D';                 /* CR */
         *t++ = '\x0A';                 /* LF */
         got = t-BfrAdr;                /* minimum 2 */
         return got;
                                        /* length of line with CR+LF */
      } /* endif */
    }  /* end of **SMTPgetdata** */
```

```
/* Here is an example with MIME support
/* Again, we demonstrate with file I/O
*/

   unsigned SMTPgetdata( char* BfrAdr, unsigned BfrLen )
   {
   static FILE* F = 0;
   static isBinary = <you figure out which>;
   if( 0==BfrLen ){ /* starting a new part, so open file */
     char* filespec[15];  /* PartNum is our index (we ++
                             it), PartMax is 'const' */
     if( PartNum > PartMax ) return -1;
                             /* no more data - we're done! */
     sprintf(filespec,"SendPart.%03u",PartNum);
     F = fopen(filespec,(isBinary?"rb":"r"));
     if( 0==F ) return -2;
              /* oh no, something's really messed up! */
     return isBinary;
              /* tell SMTPsend if Base64 encoding is needed */

     }else{   /* reading more data */
        unsigned got;
        if( isBinary ){
           /* just read bytes, SMTPsend will encode them */
           got = fread(BfrAdr,BfrLen,1,F);     /* read raw data */
        }else{ /* isText (e.g. ASCII) */
           got = fgets(BfrAdr,BfrLen-1,F);     /* get a line */
           if( got ){                          /* fix \n to be CR+LF */
              char* t = strlen(BfrAdr);
              if( t ) --t;                     /* backup to the \n */
              *t++ = '\x0D';                   /* CR */
              *t++ = '\x0A';                   /* LF */
              got = t-BfrAdr;                  /* minimum 2 */
           } /* endif */
        } /* endif */
        if( 0==got){ fclose(F); ++PartNum; }   /* end of file */
        return got;
     }  /* endif */
   } /* end of SMTPgetdata */
```

SMTPsend()

Sends a message.

```
int SMTPsend(char *MIMEtype, Iid mailserver, char *to,
             char *subject);
```

type	0 = simple message, no MIME "Multipart/Parallel" = MIME message. To be a multipart message, `MIMEtype` must start with "Multipart..." (not case-sensitive).
mailserver	IP address of mail server
to	full mailing address, for instance `aaa@bbb.com`
subject	subject of message

Normally you would get the *`mailserver`* address with a call to ***DNSresolve(),*** for instance:

```
DNSresolve("aaa@bbb.com", &mailserver);
```

This call is not inside ***SMTPsend(),*** so that the application can, when necessary, use other methods to get the mail server address.

SMTPsend() will open the TCP connection and create the mail headers. Then it will ask the user to provide the message text, by calling ***SMTPgetdata()***.

SMTPsend() uses the ANSI C timing functions ***gmtime()*** and ***localtime().*** If these functions are not supported by your development tools, these functions must be stubbed out.

Return Value

0	Successful
-1	Own domain name not known. This is either given by a DHCP server, or stored by the application into `DNSdomain`.
-2	Handshake problem
`EHOSTUNREACH`	Mail server not reachable

Example

See the **appsrc/smtest.c** file for examples of ***SMTPsend()***.

Using POP to Receive Mail

The Post Office Protocol (POP3) is a minor variation of SMTP, and allows a client to retrieve mail from a remote server mailbox. This protocol is commonly used by computers, which are not connected to the network at various times, to retrieve mail from a permanent SMTP host. The SMTP host holds messages for the client until the client empties its mailbox. This allows the client to be disconnected for a time without generating SMTP delivery errors.

A message received by ***POPreceive()*** will look exactly like a message received by ***SMTPserv()*** (see the section on *Using the SMTP Server to Receive Mail*). ***POPreceive()*** handles the networking protocols, and then passes the data to the user. The user calls the data-handling routines they need, and fills in the variables to fit their data. The following chart illustrates this process. If necessary, the user can call support routines (see *Header-Encoding Support Routines*, in this chapter) to decode the data.

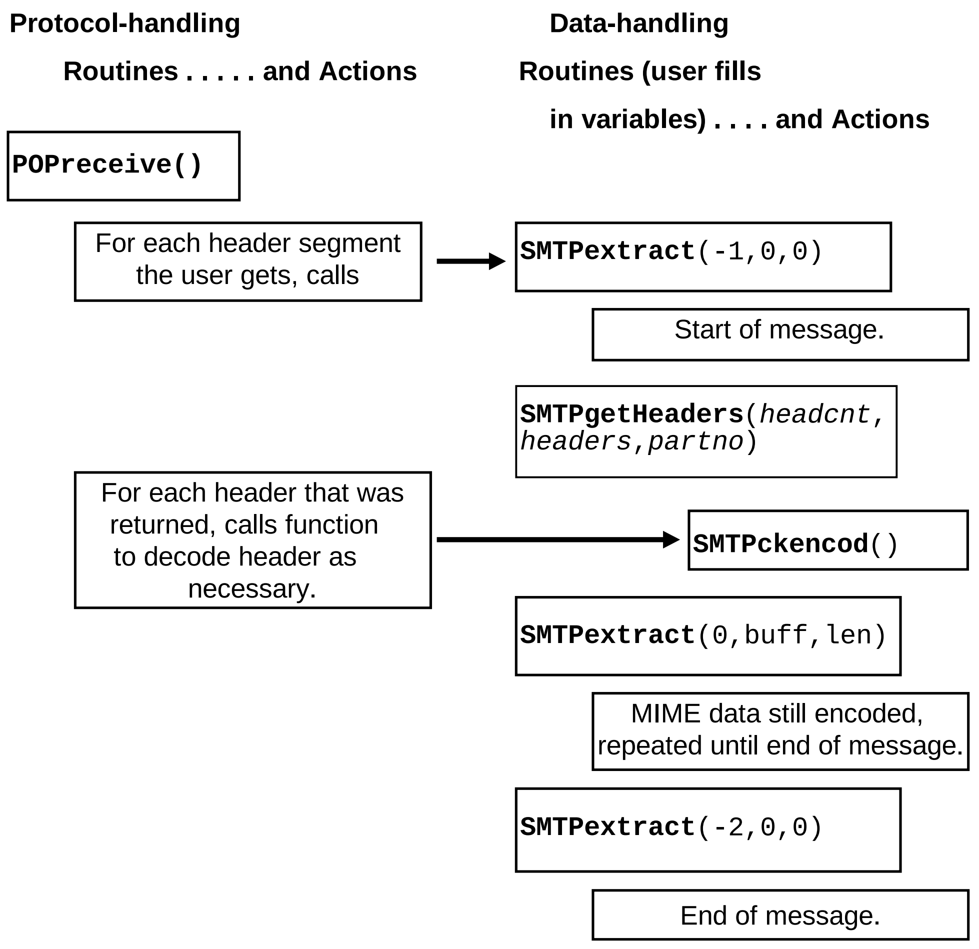
Protocol-handling
Routines . . . . . and Actions
Data-handling
Routines (user fills
in variables) . . . . and Actions
POPreceive()
For each header segment the user gets, calls
SMTPextract(-1,0,0)
Start of message.
SMTPgetHeaders(headcnt, headers,partno)
For each header that was returned, calls function to decode header as necessary.
SMTPckencod()
SMTPextract(0,buff,len)
MIME data still encoded, repeated until end of message.
SMTPextract(-2,0,0)
End of message.

Figure 4-1: POP Flow Chart

Internally, POP is used to request mail from a mail server, as follows:

1. Open TCP port 110.
2. Send command ***USER***.
3. Send command ***PASS***.
4. Send command ***RETR***.
5. Take the message.
6. Delete the message at the server using ***DELE***.
7. Repeat steps 4 thru 6 until all messages have been received.
8. Send command ***QUIT***.
9. Close connection.

This is an example of the client POP flow of functions for receipt of three simple (e.g. single-part) messages:

```
POPreceive()
   SMTPextract(-1)
   SMTPgetHeaders()
   SMTPextract(data)
          SMTPextract(data)
          etc
   SMTPextract(-2)
   SMTPextract(-1)
   SMTPgetHeaders()
          SMTPextract(data)
          etc
   SMTPextract(-2)
   SMTPextract(-1)
   SMTPgetHeaders()
   SMTPextract(data)
          etc
   SMTPextract(-2)
POPlog()
```

Chapter 4

These functions are described in this section:

POPlog()	Logs arriving messages.
POPreceive()	Receives messages from a POP server.

POPlog()

Logs arriving messages.

```
int POPlog(char *buff, int len);
```

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes

POPlog() is called by the server to log the message that's been transferred. All messages are logged with this function, exactly as they arrive. Value *len* -1 means end of message.

Return Value

<0	Error

Otherwise, number of bytes written to log.

Example

```
int POPlog(char *buff, int len)
{
    int status;

    if (len < 0)
     return fflush(logfile);
    buff[len] = '\n';
    status = fwrite(buff, len+1, 1, logfile);
    buff[len] = 0;
    return status;
}
```

POPreceive()

Retrieves messages from a POP server.

```
int POPreceive(char *POPserver, char *mailbox, char *password);
```

POPserver	name of POP server
mailbox	name of mailbox
password	password

POPreceive() will open the TCP connection and ask for messages. It will ask the POP server to delete any messages that were successfully retrieved.

POPreceive() calls the user functions ***SMTPextract()*** and ***POPlog()***.

Return Value

0	Successful
-2	Handshake problem
`EHOSTUNREACH`	Mail server not reachable, defined in **support.h**

Example

```
char *popserver, *user, *password
rslt =POPreceive(popserver, user, password)
```

Using the SMTP Server to Receive Mail

To be a server, the system needs to be running and connected to the network at all times. The system also needs a thread by itself to run. While the SMTP server is doing mail, it can't perform other functions.

IAP handles the networking protocols with ***SMTPserv()***, and then passes the data to the user. The user calls the data-handling routines they need, and fills in the variables to fit their data. The following chart illustrates this process. If necessary, the user can call support routines (see *Header-Encoding Support Routines*, in this chapter) to decode the data.

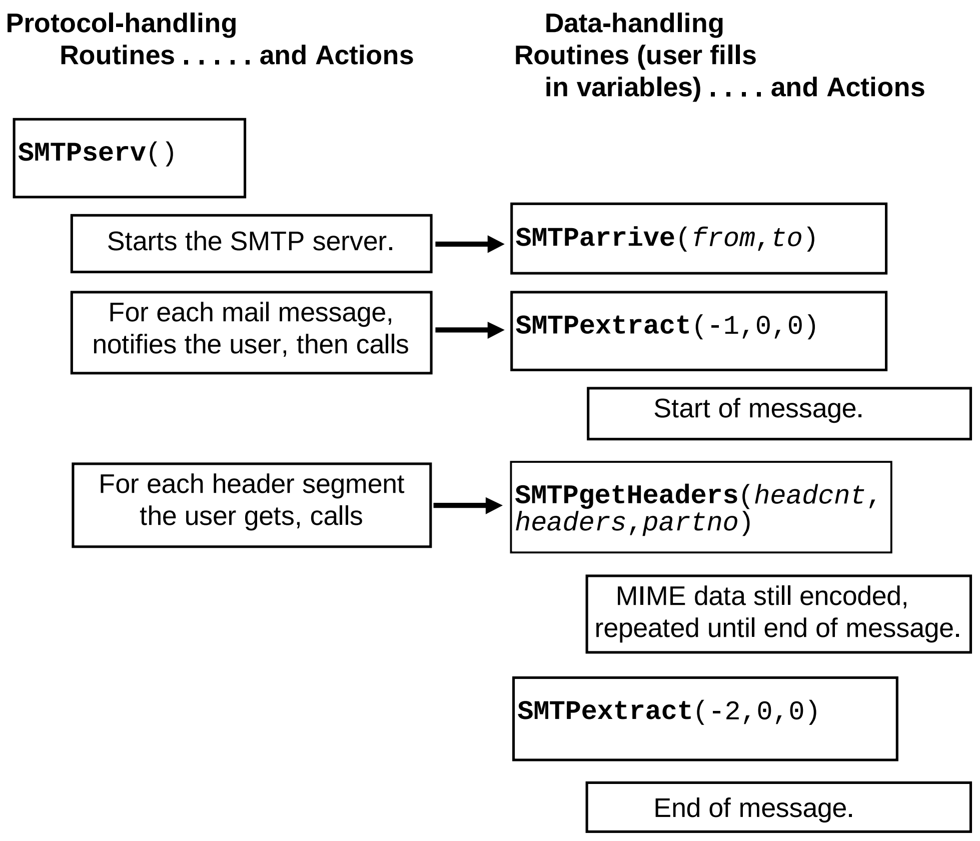
Protocol-handling
Routines . . . . . and Actions
Data-handling
Routines (user fills in variables) . . . . and Actions
SMTPserv()
Starts the SMTP server.
SMTParrive(from,to)
For each mail message, notifies the user, then calls
SMTPextract(-1,0,0)
Start of message.
For each header segment the user gets, calls
SMTPgetHeaders(headcnt, headers,partno)
MIME data still encoded, repeated until end of message.
SMTPextract(-2,0,0)
End of message.

Figure 4-2: SMTP Server-side Flow Chart

This is the flow of function calls for two mail messages (the first has one attachment):

```
SMTPserv()
     SMTParrive()
          SMTPextract(-1)
          SMTPgetHeaders()
          SMTPextract(data)
               SMTPextract(data)
               etc.
          SMTPextract(-2)
          SMTPextract(-1)
          SMTPgetHeaders()
               SMTPextract(data)
               etc.
          SMTPextract(-2)
     SMTParrive()
          SMTPextract(-1)
          SMTPgetHeaders()
          SMTPextract(data)
               etc.
          SMTPextract(-2)
SMTPlog()
```

These functions are described in this section:

SMTParrive()	Announces the arrival of a message.
SMTPchkencod()	Decodes headers.
SMTPextract()	Extracts attachments to a mail message.
SMTPgetHeaders()	Gets headers.
SMTPlog()	Logs arriving messages.
SMTPserv()	Starts the SMTP server.

SMTPgetHeaders() and ***SMTPchkencod()*** are included to allow the user to better customize the operation of the mail reader. The user is responsible for dealing with both headers and data. This improves flexibility for the user to customize their system.

SMTParrive()

Announces the arrival of a message.

`SMTParrive(char *from, char *to);`

from	the sender of the message
to	the intended recipient of the message

This call signifies that a message has arrived. You will be told who the message is from and to, so you can determine whether to accept it.

Return Value

<0	Tells the IAP server to refuse to receive this message.
0	Tells the IAP server to accept this message.

Example

```
/* The server calls this user routine for an arrived mail message.
*/
int SMTParrive(char *from, char *to)
{
    Nprintf(“Mail FROM %s TO %s\n”, from, to);
    return 0;
}
```

SMTPchkencod()

Checks headers for encoding.

```
int SMTPchkencod(char *val,struct HeadValue *hvp)
```

val	a string (the header)
hvp	the returned structure, if there is encoding

Run each header through this routine to detect any special encoding. Special encoding means the message contains data that's a MIME type.

The user is responsible for allocating enough buffer space for their data. The buffer length is configured at compile time, in the include files.

This is the `HeadValue` structure:

```
struct HeadValue {
 char *value;
 char encoding;
 char *charset;
 char *etext;
};
```

Return Value

0	No encoding
1	Yes, there is encoding

Example

There is an example in the file **smtest.c**.

SMTPextract()

Extracts attachments to a mail message.

`SMTPextract(int flag, char *buff, int len);`

flag	The flag values are: `-1` = new message starting `-2` = message complete `0` = *len* bytes of ASCII data with CR+LF at the end
buff	a pointer to the buffer where the data is
len	the length of the data, in bytes

This is a user-supplied routine that is given the data lines as they arrive. For a single-part message, the sequence is:

```
SMTPextract(-1,0,0)              /* start of message */
SMTPgetHeaders(...)              /* here are the headers */
SMTPextract(0,address,length)    /* all the data lines */
...                              /* one at a time */
SMTPextract(-2,0,0)              /* end of message */
...                              /* start of next message */
```

For a multipart message, the sequence is:

```
SMTPextract(-1,0,0)              /* start of message */
SMTPgetHeaders(...)              /* here are the message headers */
SMTPextract(-1,0,0)              /* start of part */
SMTPgetHeaders(...)              /* here are the part headers */
SMTPextract(0,address,length)    /* all the data lines */
...                              /* one at a time */
SMTPextract(-2,0,0)              /* end of part */
SMTPextract(-1,0,0)              /* start of 2nd part */
SMTPgetHeaders(...)              /* here are the part headers */
SMTPextract(0,address,length)    /* all the data lines */
...                              /* one at a time */
SMTPextract(-2,0,0)              /* end of message */
SMTPextract(-2,0,0)              /* end of message */
 ...                             /* start of next message */
```

SMTPextract() will read messages from the POP server one after another. If it sees a multipart MIME header, it will handle the nested calls to step through the parts. Currently, it does not support parts nested several levels deep.

Return Value

<0	Error
0 or >0	Successful

Example

Here's an example of ***SMTPextract(),*** taken from **smtest.c**. This writes the message parts into separate files.

```
int SMTPextract(int flag, char *buff, int len)
{
char buf[32];
if (flag < 0){
  if (flag == -1){
     Nsprintf(buf, “mail.%03d”, usfileno++);
        if (usfileno >= 1000)
           usfileno = 0;
        ffp = fopen(buf, “wb”);
     }else
        fclose(ffp);
     return 0;
  }
  if (flag == 0)
     buff[len++] = ‘\r’, buff[len++] = ‘\n’;
  return fwrite(buff, len, 1, ffp);
}
```

SMTPgetHeaders()

Gets headers.

```
int SMTPgetHeaders(int cnt,struct SMTPHeaders *heads,
                   int part)
```

cnt	the number of headers
heads	a structure containing the array of headers
part	which part of the mail message

This routine will be called each time the headers have been read. Only one level of attachments is allowed (i.e., the mail message can have attachments, but attachments cannot have attachments).

The user is responsible for allocating enough buffer space for their data.

This is the `SMTPHeaders` structure:

```
struct SMTPheaders {
 char *head;
 char      *value;
};
```

Return Value

Always returns zero.

Example

```
/* This is just an EXAMPLE routine that the user could modify to do
something with the mail headers.
 *
 *    cnt   is the number of headers
 * heads is a pointer to the headers
 * part  is which part of a multi-attachment e-mail
 * 0= top headers
 */
int SMTPgetHeaders(int cnt,struct SMTPheaders *heads, int part)
{
int i, len;
char *head, *val, *ptr;
struct HeadValue hv;
for(i=0;i<cnt;i++){
   head = heads[i].head;
   val = heads[i].value;
   printf("%s:  %s\n",head, val);
#ifdef   NOT_USED
   len = SMTPchkencod(val,&hv);
   if(len){
      len = strlen(hv.etext);
      if(hv.encoding == 'B'){
         len = frombase64(hv.etext,len);
      } else if (hv.encoding == 'Q'){
         len = fromquoted(hv.etext,len);
      }
   }
   Nprintf("%s  %s  %s  %s",hv.value,hv.charset,hv.etext);
#endif
   }
return 0;
}
/* DONE with SMTPgetHeaders()
 ————————————————————————*/
```

SMTPlog()

Logs arriving messages.

```
int SMTPlog(char *buff, int len);
```

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes

SMTPlog() is called by the server to log the message that's been transferred. All messages are logged with this function, exactly as they arrive. Value *len* -1 means end of message.

Return Value

<0	Error

Otherwise, number of bytes written to log.

Example

```
/* This routine is called to log arriving mail messages,
   headers and content.
   Each call supplies one line, without end-of-line
   characters (CRLF).
   Value len = -1 means end of message.
*/

    int SMTPlog(char *buff, int len)
    {
    int status;

    if (len < 0)
      return fflush(logfile);
    buff[len] = ‘\n’;
    status = fwrite(buff, len+1, 1, logfile);
    buff[len] = 0;
    return status;
    }
```

SMTPserv()

Starts the SMTP server.

```
void SMTPserv(void);
```

This call starts the SMTP server, and never returns. The server does not use multitasking while receiving a message, for the following reasons:

- SMTP is not interactive, so multitasking will make no difference unless the message is large.
- Function ***SMTPgetHeaders()*** needs a lot of memory for its arguments. This would in any case force a limit on the number of server tasks.

SMTPserv() handles the following SMTP commands:

HELO	new client
MAIL	new mail from sender
NOOP	no operation
QUIT	terminates session
RCPT	names recipient
RSET	resets session
SOML	send-or-mail, treated as mail

SMTPserv() calls the three user functions ***SMTPlog(), SMTPextract(),*** and ***SMTParrive()*** for any arrived mail message.

SMTPserv() gives all arrived messages to the user. It will not do any relaying or resending.

Chapter 4

Header-Encoding Support Routines

These routines are described in this section:

decodetext()	Decides whether to call ***frombase64()*** or ***fromquoted().***
frombase64()	Converts a base64 string to binary.
fromquoted()	Converts quoted text to unquoted.
tobase64()	Converts a binary buffer to base64.

decodetext()

Decides whether to call ***frombase64()*** or ***fromquoted().***

`decodetext(char *buff, int len)`

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes

This is an upper-level routine. When you pass a header to it, it evaluates the header and performs the correct decoding by calling either ***frombase64()*** or ***fromquoted().***

The user is responsible for allocating enough buffer space for their data.

Return Value

The length of the new buffer.

Example

```
newlen = decodetext(buf,len);
if (len)
    printf(“%s\n”,buf);
```

frombase64()

Converts a base64 string to binary. The size shrinks to ¾.

```
int frombase64(char *buff, int len)
```

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes

The user is responsible for allocating enough buffer space for their data, keeping in mind that ***frombase64()*** shrinks the size of the data.

Return Value

The length of the new buffer.

Example

```
newlen = frombase64(tobuf, newlen);
printf(“%s\n”,tobuf);
```

fromquoted()

Converts quoted text to unquoted.

`fromquoted(char *buff, int len)`

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes

Control characters in the stream are within quotation marks. The ***fromquoted()*** routine converts these to unquoted text.

The user is responsible for allocating enough buffer space for their data.

Return Value

The length of the new buffer.

Example

```
newlen = fromquoted(buf,len);
printf("%s",buf);
```

tobase64()

Converts a binary buffer *buf* of length *len* to base64 with a 3->4 expansion.

```
int tobase64(char *buff, int len, char *tobuf)
```

buff	a pointer to the buffer where the message is
len	the length of the buffer, in bytes
tobuf	a pointer to the buffer to hold the decoded data

The user is responsible for allocating enough buffer space for their data, keeping in mind that ***tobase64()*** increases the size of the data.

Return Value

The length of the new buffer.

Example

```
frombuf = “How now brown cow”;
fromlen = strlen(frombuf);
newlen =  tobase64(frombuf,fromlen,tobuf);
printf(“%s\n”,tobuf);
```

Test Program

Smtest is a DOS test program for the mail protocols. It will perform different functions, depending on the command arguments. It can run as either the server or the client.

This command runs the SMTP server, to receive mail messages:

```
SMTEST
```

The messages will be logged into the file **mail.log**. Any extracted message parts will be written into files **mail.000**, **mail.001** and so on.

This command will send a mail message to the specified recipient:

```
SMTEST <recipient>
```

The message content comes from the files listed at the top of **smtest.c**. If the source variable `MULTIPART` is set, all files will be sent, otherwise just the first. The mail server is determined using with a call to ***DNSresolve().*** Source variable `DOMAIN` gives the local domain name.

You can use the source-level parameter `TESTSERVER` to send the mail to **smtest** running as server.

This command calls up the POP server, and retrieves any messages queued for the specified mailbox:

```
SMTEST <POPserver> <mailbox> <password>
```

The messages are logged and extracted exactly as in the first test case.

Any successfully received message is deleted from the mailbox

Index