11 The Network Kit

11 The Network Kit

The Network Kit is divided into two domains:

- The Kit provides a collection of global C functions that let you communicate with other computers through the TCP or UDP protocols. With a few exceptions, the names and intents of the functions adhere to the precedent set by the BSD network/socket implementation. Note, however, that some BSD-defined functions are not yet implemented.
- The Kit also provides C functions and a class (BMailMessage) that let you talk to the mail daemon, and send and receive mail messages. With the functions and the class, you should be able to write a fully-featured mail-reading and -writing application.

The network and socket documentation can be found in the sections “Network Names, Addresses, and Services” on page 5, and “Network Sockets” on page 13. In addition, you can find some further socket examples and tips in the “Be Engineering Insights” column of the Be Newsletter, issues 19 and 30.

Functions that access the mail daemon, the process that makes the mail system run, are documented in “The Mail Daemon” on page 29. The BMailMessage class is documented in “Mail Messages (BMailMessage)” on page 37.

Network Names, Addresses, and Services

Declared in: <net/netdb.h>
<net/socket.h>

Overview

The functions described below let you look up the names, addresses, and other information about the computers and services that the local computer knows about, and let you retrieve information about the current user's account. Also defined here are functions that perform *Internet Protocol* (IP) address format conversion.

You use the functions defined here to find the information you need so you can form a connection to some other machine. Connecting to other machines is described in "Network Sockets" on page 13.

Terms and Tools

Throughout the following function descriptions, an *IP address* is the familiar four-byte, dot-separated numeric identifier. For example,

```
192.0.0.1
```

The bytes in a multi-byte address are always given in *network byte order* (big-endian). The current BeBox is also big-endian, so you don't have to convert IP address values—but for portability and forward-compatibility, you may want to. See the group of functions with the obsessively shortened names (**ntohs()**, **htohl()**, etc.) for more information on such transformations.

An *IP name* is the three-element "machine.domain.extension" case-insensitive text name:

```
decca.be.com
```

The two most important functions described below, **gethostbyname()** and **gethostbyaddr()**, retrieve information about computers ("hosts") that can be reached through the network. Host information is typically (and primarily) gotten from the *Domain Name Server* (DNS), a service that's usually provided by a server computer that's responsible for tasks such as mail distribution and direct communication with the *Internet Provider Service* (IPS).

You can also provide host information by adding to your computer's **/boot/system/hosts** file. This is a text file that contains the IP addresses and names of the hosts that you want your computer to know about. Each entry in the file lists, in order on a single line, a host's IP address, IP name, and other names (aliases) by which it's also known. For example:

```
# Example /boot/system/hosts entries
192.0.0.1 phaedo.racine.com fido phydough
205.123.5.12 playdo.mess.com plywood funfactory
```

The amount of whitespace separating the elements is arbitrary. The only killing point is that there mustn't be any leading whitespace before the IP address.

If you're connected to DNS, then you shouldn't need the **hosts** file. If you're not connected to a network at all, the only way to get information about other machines is through the **hosts** file, but it won't do you much good—you won't be able to use the information to connect to other machines. The archetypal situation in which the **hosts** file becomes useful is if your BeBox is connected to some other machine (we'll call it Brand X), and the Brand X machine is supposed to be connected to a DNS machine, but this latter connection is down (or the DNS machine isn't running). If you have an entry in your BeBox **hosts** file that identifies the Brand X machine, you'll still be able to look up the machine's address and connect to it, despite the absence of DNS.

Functions

gethostbyname(), gethostbyaddr(), herror()

struct hostent ***gethostbyname(**const char **name***)**
struct hostent ***gethostbyaddr(**const char ***address***, int ***length***, int ***type***)**
void **herror(**const char ***string***)**

The two **gethostby...()** functions retrieve information about a particular host machine, stuff the information into a global "host entry" structure, and then return a pointer to that structure. To get this information, the functions talks to the Domain Name Server. If DNS doesn't respond or doesn't know the desired host, the functions then look for an entry in the file **/boot/system/hosts**. See "Terms and Tools" on page 5 for more information on DNS and the **hosts** file.

herror() generates a human-readable message that describes the most recent **gethostby...()** error, and prints it to standard error.

Note: Because **gethostbyname()** and **gethostbyaddr()** use a global structure to return information, the functions are *not* thread safe.

The gethostbyname() Function

gethostbyname()'s *name* argument is a **NULL**-terminated, case-insensitive host name that must be no longer than **MAXHOSTNAMELEN** (64) characters (not counting the **NULL**). The name can be:

- An entire "machine.domain.extension" IP name—"mybox.me.com", for example.
- Just the machine name portion—"mybox" (DNS only). In this case, the domain and extension of the local machine are automatically appended. (If you're looking up an IP name in the **hosts** file, the domain and extension *aren't* appended for you.)
- A host name alias. Aliases are alternate names by which a host is known. Your DNS should provide a means for declaring aliases; you can also declare them in your **hosts** file.

The gethostbyaddr() Function

gethostbyaddr()'s *address* argument is a pointer to a complete IP address given in its natural format (but cast to a **char ***; note that the argument's type declaration *doesn't* mean that the function wants the address converted to a string). *length* is the length of *address* in bytes; *type* is a constant that gives the format of the address.

For IP format, the first argument is a four-byte integer, *length* is always 4, and type is **AF_INET** ("*A*ddress *F*ormat: *I*nter*NET*"). The following gets the **hostent** for a hard-coded address:

```
/* This is the hex equivalent of 192.0.0.1
ulong addr = 0xc0000001;
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

If you have an address stored as a string, you can use the **inet_addr()** function to convert it to an integer:

```
ulong addr = inet_addr("192.0.0.1");
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

The hostent Structure

If a **gethostby...()** function fails, it returns **NULL**; otherwise, it returns a pointer to a global **hostent** structure. The **hostent** structure (which isn't **typedef**'d) looks like this:

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```

The fields are:

- **h_name** is the IP name of the host (or the "official" name given in the **hosts** file).
- **h_aliases** is a **NULL**-terminated array of other names by which the host is known. These names aren't necessarily in IP name format; typically, they're single-word names.
- **h_addrtype** identifies the format of the addresses listed in **h_addr_list**. Currently, the type is always **AF_INET**.
- **h_length** is the length, in bytes, of the host's address. In **AF_INET** format, the address is four bytes long
- **h_addr_list** is a **NULL**-terminated array of pointers to the addresses by which the host is known. Host addresses are given in network byte order.

As a convenience, the global **h_addr** constant is a fake field that points to the first item in the **h_addr_list** field. Keep in mind that **h_addr** must be treated as a structure field—it must point off a **hostent** structure. Also, make sure you dereference the **h_addr** "field" properly. For example:

```
ulong ip_address;
struct hostent *theHost;

theHost = gethostbyname("fido");
ip_address = *(ulong *)theHost->h_addr;
```

As a demonstration of the **h_addr** definition, the final line is the same as

```
ip_address = *(ulong *)theHost->h_addr_list[0];
```

Keep in mind that the **hostent** structure that's pointed to by the **gethostby...()** functions is global to your application's address space. If you want to cache the structure, you should copy it as soon as it's returned to you.

h_errno and the herror() Function

The host look-up functions use a global error variable (an integer), called **h_errno**, to register errors. You can look at the **h_errno** value directly in your code after a host function fails (the potential **h_errno** values are listed below). Alternatively, you can use the **herror()** function which prints, to standard error, its argument followed by a system-generated string that describes the current state of **h_errno**.

The values that **h_errno** can take, and the corresponding **herror()** strings, are:

Value	Meaning
HOST_NOT_FOUND	"unknown host name"
TRY_AGAIN	"host name server busy"
NO_RECOVERY	"unrecoverable system error"
NO_DATA	"no address data is available for this host name"
anything else	"unknown error"

Note that while **h_errno** is set when something goes wrong, it isn't cleared if all is well. For example, if **gethostbyname()** can't find the named host, **h_errno** is set to **HOST_NOT_FOUND** and the function returns **NULL**. If, in an immediately subsequent call, the function succeeds, a pointer to a valid **hostent** is returned, but **h_errno** will *still* report **HOST_NOT_FOUND**.

The moral of this tale is that you should *only* check **h_errno** (or call **herror()**) if the network function call has failed, or clear it yourself before each **gethostby...()** call. Or both:

```
struct hostent *host_ent;

h_errno = 0;
if ( !(host_ent = gethostbyname("a.b.c"))
    herror("Error");
```

Furthermore, **h_errno** might be legitimately set to a new error code even if the **gethostby...()** function succeeds. For example, if DNS can't be reached but the desired host is found in the **hosts** file, **h_errno** will be set to **TRY_AGAIN**, yet the returned **hostent** will be legitimate (it won't be **NULL**).

Be aware that **TRY_AGAIN** is used as a blanket "DNS doesn't know" state, *regardless* of the reason why. In other words, **h_errno** is set to **TRY_AGAIN** if DNS is actually down, if your machine isn't connected to the network, or if DNS simply doesn't know the requested host. You can use this fact to tell whether a (successful) look-up was performed through DNS or the **hosts** file:

```
struct hostent *host_ent;

h_errno = 0;
if ( !(host_ent = gethostbyname("a.b.c"))
    herror("Error");
else
    if (h_errno == TRY_AGAIN)
        /* The hosts file was used. */
    else
        /* DNS was used. */
```

Keep in mind that **h_errno** is global; be careful if you're using it in a multi-threaded program.

gethostname(), getusername(), getpassword()

int **gethostname(**char **name*, unsigned int *length***)**
int **getusername(**char **name*, unsigned int *length***)**
int **getpassword(**char **password*, unsigned int *length***)**

These functions retrieve, and copy into their first arguments, the name of the local computer, the name of the current user, and the current user's encoded password, respectively. In all three case, *length* gives the maximum number of characters that the functions should copy. If the length of the desired element is less than *length*, the copied string will be **NULL**-terminated.

The functions return the number of characters that were actually copied (not counting the **NULL** terminator). If there's an error—and such should be rare—the **gethostname()** and **getusername()** functions return 0 and point their respective name arguments to **NULL**. **getpassword()**, sensing an error, copies "*" into the password argument and returns -1 (thus you can tell the difference between a **NULL** password—which would legitimately return 0—and an error).

All three bits of information (host name, user name, and password) are taken from the settings that are declared through the **Network** preferences application.

A typical use of **gethostname()** is to follow the call with **gethostbyname()** in order to retrieve the address of the local host, as shown below:

```
/* To fill a need, we invent the gethostaddr() function. */
long gethostaddr(void)
{
    struct hostent *host_ent;
    char host_name[MAXHOSTNAMELEN];

    if (gethostname(host_name, MAXHOSTNAMELEN) == 0)
        return -1;

    if ((host_ent = gethostbyname(host_name)) == NULL)
        return -1;

    return *(long *)host_ent.h_addr;
}
```

Keep in mind that since host name information is taken from Network preferences, there's no guarantee that the name that's returned by **gethostname()** will match an entry that DNS or the **hosts** file knows about.

getservbyname()

struct servent ***getservbyname(**const char **name***,** const char **protocol***)**

You pass in the name of a service (such as "ftp") that runs under a particular protocol (such as "tcp"), and **getservbyname()** returns a pointer to a **servent** structure that describes the service.

The **servent** structure is:

```
struct servent {
    char *s_name;
    char **s_aliases;
    int s_port;
    char *s_proto;
};
```

- **s_name** is the name of the service.
- **s_aliases** is a **NULL**-terminated array of other names by which the services is known.
- **s_port** is the port number on which the service runs (given in network byte order)
- **s_proto** names the protocol ("tcp", "udp", etc.) that supports the service.

Currently, the function recognizes only two services: "ftp" and "telnet". Both run under the "tcp" protocol; thus, the only valid calls to **getservbyname()** are:

```
getservbyname("ftp", "tcp");
```

and

```
getservbyname("telnet", "tcp");
```

Such calls point to (separate) pre-defined **servent** structures that look like this:

field	ftp structure	telnet structure
s_name	"ftp"	"telnet"
s_aliases	**NULL**	**NULL**
s_port	21	23
s_proto	"tcp"	"tcp"

If you ask for a service other than these two, the function returns **NULL**. Although the two **servent** structures are separate entities, they are both global to your application. In theory, this means the **getservbyname()** function isn't thread-safe. However, since the structures are hard-coded and separate, there's little danger in using them unprotected in a multi-threaded program.

inet_addr(), inet_ntoa()

unsigned long **inet_addr(**const char **addr***)*
char ***inet_ntoa(**struct in_addr *addr***)**

These functions convert addresses from ASCII to IP format and vice versa. Neither of them consults the DNS or the hosts file to perform the conversion—in other words, they perform the conversions without regard for an address' correspondence to an actual machine.

inet_addr() converts from ASCII to IP:

```
ulong addr = inet_addr("192.0.0.1");
```

The result of this call (**addr**) would be appropriate as the initial argument to **gethostbyaddr()** (for example). The returned address is in network byte order.

inet_ntoa() converts the other way: It takes an IP address and converts it into an ASCII string. Note that the address that you pass in must first be placed in the **s_addr** field of the argument **in_addr** structure (**s_addr** is the structure's only field). For example:

```
in_addr addr;
char addr_buf[16];

addr.s_addr = 0xc0000001;
strcpy(addr_buf, inet_ntoa(addr));
```

Here, **addr_buf** will contain the (**NULL**-terminated) string "192.0.0.1". **inet_ntoa()** isn't thread-safe; if you want to cache the string that it returns you must copy it, as shown in the example. Given the IP format, the string that **inet_ntoa()** returns is guaranteed to be no more than 16 characters long (four 3-character address components, three dots, and a **NULL**).

ntohs(), ntohl(), htons(), htonl()

short **ntohs(**short *val***)**
long **ntohl(**long *val***)**
short **htons(**short *val***)**
long **htonl(**long *val***)**

These macros convert values between host and network byte order:

Macro	Meaning
ntohs()	network short to host short
ntohl()	network long to host long
htons()	host short to network short
htonl()	host long to network long

Network byte order is big-endian; the host byte order is machine-dependent. The current BeBox is big-endian, so these macros are, essentially, no-ops: They return their respective arguments without conversion. To be scrupulous, however, you should convert all multi-byte values that you write to or get from the Internet. For example, a truly "safe" call to **gethostbyaddr()** (for example) would look like this:

```
ulong addr = htonl(inet_addr("192.0.0.1");
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

Network Sockets

Declared in: <net/socket.h>

Overview

Sockets are entry ways onto a network. To transmit data to another machine, you create a socket, tell it how to find the other computer, and then tell it to send. To receive data, you do the opposite: You create a socket, tell it who to listen to (in some cases), and then wait for data to come pouring in.

Socket concepts are mixed in with regular function descriptions; the **socket()** function, which is where any socket user must start, is described first. The description gives a general overview of the different types of sockets, how you use them, and where to go to next. The other socket functions are then listed in a separate section, in the expected alphabetical order.

The socket implementation (and philosophy) follows the precedent established by 4.2BSD. In particular, the API presented here bends many of the Be naming and calling conventions in order to make porting existing programs easier.

The socket() Function

socket(), closesocket()

int **socket(**int *family*, int *type*, int *protocol***)**

int **closesocket(**int *socket***)**

The **socket()** function returns a token (a non-negative integer) that represents the local end of a connection to another machine. Freshly returned, the token is abstract and unusable; to put the token to use, you have to pass it as an argument to other functions—such as **bind()** and **connect()**—that know how to establish a connection (however temporary) over the network. (The function's arguments are examined in a separate section, below.)

A successful **socket()** call returns a non-negative integer—keep in mind that 0 is a valid socket token. Also keep in mind that socket tokens are *not* file descriptors (this violates the BSD tradition). Upon failure, **socket()** returns -1 and sets the global **errno** variable to one of these values:

Value	Meaning
EAFNOSUPPORT	*format* was other than **AF_INET.**
EPROTOTYPE	*type* and *protocol* mismatch.
EPROTONOSUPPORT	Unrecognized *type* or *protocol* value.

closesocket() closes a socket's connection (if it's the type of socket that can hold a connection) and frees the resources that have been assigned to the socket. When you're done with the sockets that you've created, you should pass each socket token to **closesocket()**—no socket, no matter how abstract or how you use it, is exempt from the need to be closed. In regard to this universal need, you should be aware that this extends to sockets that are created through the **accept()** function (which we'll get to later).

closesocket() returns less-than-zero if its argument is invalid.

The socket() Arguments

socket()'s three arguments, all of which take predefined constants as values, describe the type of communication the socket can handle:

- *family* takes a constant the describes the network address format that the socket understands. Currently, it must be **AF_INET** (the Internet address format).
- The *type* constant must be either **SOCK_STREAM** or **SOCK_DGRAM**. The constant describes (roughly) the "persistence" of the connection that can be formed through this socket. The **SOCK_STREAM** constant means the impending connection (which is formed through a **connect()** or **bind()** call) will remain open until told to close. **SOCK_DGRAM** describes a "datagram" socket; the connection through a datagram socket is open while data is being sent (typically through **sendto()**) or received (similarly, **recvfrom()**). It's closed at all other times (note, however, that you still have to call **closesocket()** on a datagram socket when you're done with it).
- *protocol* describes the "messaging" protocol, a description that's closely related to the socket type. Although there are three *protocol* constants (**IPPROTO_TCP**, **IPPROTO_UDP**, and **IPPROTO_ICMP**), values that you would actually use are either 0 or, less commonly, **IPPROTO_ICMP**. More specifically, if you set the *type* to be **SOCK_STREAM**, then a *protocol* of 0 automatically sets the messaging protocol to **IPPROTO_TCP**—this is the "natural" messaging protocol for a stream socket. Similarly, **IPPROTO_UDP** is the natural protocol for the **SOCK_DGRAM** type. Note that it's an error to ask for a "udp stream" or a "tcp datagram"—in other words, you can't specify **SOCK_STREAM** with **IPPROTO_UDP**, or **SOCK_DGRAM** with **IPPROTO_TCP**.

As implied by the preceding description, the most typical socket calls are:

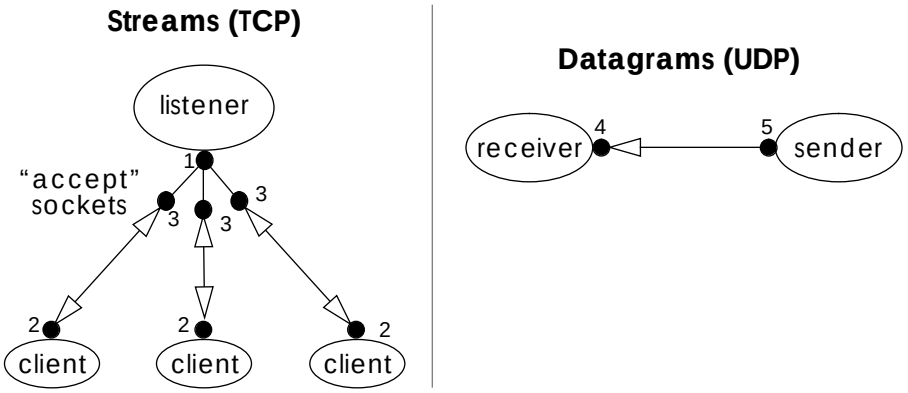
```
/* Create a stream TCP socket. */
long tcp_socket = socket(AF_INET, SOCK_STREAM, 0);

/* Create a datagram UDP socket. */
long udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

ICMP messages are, traditionally, sent through "raw" sockets. The Network Kit doesn't currently support such sockets, so you should use datagram sockets instead:

```
/* Create a datagram icmp socket. */
long icmp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

Sorts of Sockets

There are only two socket type constants: **SOCK_STREAM** and **SOCK_DGRAM**. However, if we look at the way sockets are used, we see that there are really five different categories of sockets, as illustrated below.



The labelled ovals represent individual computers that are attached to the network. The solid circles represent individual sockets. The numbers near the sockets are keys to the socket categories, which are examined in the following:

1. *The stream listener socket.* A stream listener socket provides access to a service that's running on the "listener" machine (you might want to think of the machine as being a "server.") The listener socket waits for client machines to "call in" and ask to be served. In order to listen for clients, the listener must call **bind()**, which "binds" the socket to an IP address and machine-specific port, and then **listen()**. Thus primed, the socket waits for a client message to show up by sitting in an **accept()** call.

2. *The stream client socket.* A stream client socket asks for service from a server machine by attempting to connect to the server's listener socket. It does this through the **connect()** function. A stream client can be bound (you can call **bind()** on it), but it's not mandatory.

3. *The "accept" socket.* When a stream listener hears a client in an **accept()** call, the function call creates yet another socket called the "accept" socket. Accept sockets are valid sockets, just like those you create through **socket()**. In particular, you have to remember to close accept sockets (through **closesocket()**) just as you would the sockets you explicitly create. Note that you can't bind an accept socket—the socket is bound automatically by the system.

4. *The datagram receiver socket.* A datagram receiver socket is sort of like a stream listener: It calls **bind()** and waits for "senders" to send messages to it. Unlike the stream listener, the datagram receiver *doesn't* have to call **listen()** or **accept()**. Furthermore, when a datagram sender sends a message to the receiver, there's no ancillary socket created to handle the message (there's no UDP analog to the TCP accept socket).

5. *The datagram sender socket.* A datagram sender is the simplest type of socket—all it has to do is identify a datagram receiver and send messages to it, through the **sendto()** function. Binding a datagram sender socket is optional.

Returning to the illustration, notice that the paths connecting the stream socket clients to the stream listener (through the accept sockets) are "double arrow-headed." This indicates that TCP communication is two-way: Once the link between a client and the listener has been established (through **bind()**/**listen()**/**accept()** on the listener side, and **connect()** on the client side), the two machines can talk to each other through respective and complementary **send()** and **recv()** calls.

Communication along a UDP path, on the other hand, is one-way, as indicated by the direction of the arrow. The datagram sender can send messages (through **sendto()**), and the datagram receiver can receive them (through **recvfrom()**), but the receiver can't send message back to the sender. However, you can simulate a two-way UDP conversation by binding both sockets. This doesn't change the definition of the UDP path, or the capabilities of the two types of datagram sockets, it simply means that a bound datagram socket can act as a receiver (it can call **recvfrom()**) or as a sender (it can call **sendto()**).

Note: To be complete, it should be mentioned that datagram sockets can also invoke **connect()** and then pass messages through **send()** and **recv()**. The datagram use of these functions is a convenience; its advantages are explained in the description of the **sendto()** function.

Other Functions

bind()

int **bind(**int *socket*, struct sockaddr **interface*, int *size***)**

The **bind()** function creates an association between a socket and an "interface," where an interface is a combination of an IP address and a port number. Binding is, primarily, an in-coming message primer: When a message sender (whether it's a stream client or a datagram sender) sends a message, it tags the message with an IP address and a port number. The receiving machine—the machine with the tagged IP address—delivers the message to the socket that's bound to the tagged port.

The necessity of the bind operation, therefore, depends on the type of socket; referring to the five categories of sockets enumerated in the **socket()** function description (and

illustrated in the charming picture found there), the "do I need to bind?" question is answered thus:

1. **Stream listener sockets *must* be bound**. Furthermore, after binding a listener socket, you must then call **listen()** and, when a client calls, **attach()**.

2. **Stream client sockets *can* be bound**, but they don't have to be. If you're going to bind a client socket, you should do so *before* you call **connect()**. The advantages of binding a stream client escape me at the moment. In any case, the client doesn't have to bind to the same port number as the listener—the listener's binding and the client's binding are utterly separate entities (let alone that they are on different machines). However, the client does *connect* to the interface that the listener is bound to.

3. **Stream attach sockets *must not* be bound**.

4. **Datagram receiver sockets *must* be bound.**

5. **Datagram sender sockets don't *have* to be bound**...but if you're going to turn around and use the socket as a receiver, then you'll have to bind it.

Once you've bound a socket, you can't unbind it. If you no longer want the socket to be bound to its interface, the only thing you can do is close the socket (**closesocket()**) and start all over again.

Also, a particular interface can be bound to by only one socket at a time. Furthermore, in the current Be implementation of sockets, a single socket can only bind to one interface at a time. This differs with the BSD socket implementation which sets the expectation for a socket to be able to bind to more than one interface. Consider it a bug that will be fixed in a subsequent release. If you need to bind to more than one interface, you'll need, instead, to create more than one socket and bind each one separately. An example of this is given later in this function description.

The bind() Arguments

bind()'s first argument is the socket that you're attempting to bind. This is, typically, a socket of type **SOCK_STREAM**. The address/port combination (or "interface") to which you're binding the socket is passed through the *interface* argument. This is typed as a **sockaddr** structure, but, in reality, you have to create and pass a **sockaddr_in** structure cast as a **sockaddr**. The **sockaddr_in** structure is defined as:

```
struct sockaddr_in {
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[4];
};
```

- **sin_family** is the same as the address format constant that used to create the socket (the first argument to **socket()**). Currently, it's always **AF_INET**.

- **sin_port** is the port number that the socket will bind to, given in network byte order. Valid port numbers are between 1 and 65535; numbers up to 1024 are reserved for services such as **ftp** and **telnet**. If you're not implementing a standard service, you should choose a port number above 1024. The actual value of the port number is meaningless, but keep in mind that the port number must be unique for a particular address; only one socket can be bound to a particular address/port combination.

  Note: Currently, there's no system-defined mechanism for allowing a client/sender machine to ask a listener/receiver machine for its port numbers. Therefore, when you create a networked application, you either have to hard-code the port numbers or, better yet, provide default port numbers that the user (or a system administrator) can easily change.

- **sin_addr** is an **in_addr** structure that stores, in its **s_addr** field, the IP address of the socket's machine. As always, the address is in network byte order. You can use an address of 0 to tell the binding mechanism to find an address for you. By convention, binding to address 0 (which is conveniently symbolized by the **INADDR_ANY** address) means that you want to bind to *every* address by which your computer is known, including the "loopback" (address 127.0.0.1, or the constant **INADDR_LOOPBACK**).

  On the BeBox, currently, this global-binding convention isn't implemented; instead, when you bind to **INADDR_ANY**, the **bind()** function binds to the *first* available interface (where "availability" means the address/port combination is currently unbound). Internet interfaces are considered before the loopback interface. If you want to bind to all interfaces, you have to create a separate socket for each. An example of this is given later.

- **sin_zero** is padding. To be safe, you should fill it with zeros.

The *size* argument is the size, in bytes, of the second argument.

If the **bind()** call is successful, the *interface* argument is set to contain the actual address that was used. If the socket can't be bound, the function returns less-than-zero, and sets the global **errno** to **EABDF** if the *socket* argument is invalid; for all other errors, **errno** is set to -1.

The following example shows an unexceptional use of the **bind()** function. The example uses the fictitious **gethostaddr()** function that was defined in the description of the **gethostname()** function in "Network Names, Addresses, and Services".

```
struct sockaddr_in sa;
int sock;
long host_addr;

/* Create the socket. */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    /* error */

/* Set the address format for the imminent bind. */
sa.sin_family = AF_INET;

/* We'll choose an arbitrary port number. */
sa.sin_port = htonl(2125);

/* Get the address of the local machine. If the address can't
 * be found (the function looks it up based on the host name),
 * then we use address INADDR_ANY.
 */
if ((host_addr = (ulong)gethostaddr()) == -1)
    host_addr = INADDR_ANY;
sa.sin_addr.s_addr = host_addr;

/* Clear sin_zero. */
memset(sa.sin_zero, 0, sizeof(sa.sin_zero));

/* Bind the socket. */
if (bind(sock, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    /* error */
```

As mentioned earlier, the bind-to-all-interfaces convention (by asking to bind to address 0) isn't currently implemented. Thus, if the **gethostaddr()** call fails in the example, the socket will be bound to the first address by which the local computer is known.

But let's say that you really do want to bind to all interfaces. To do this, you have to create separate sockets for each interface, then call **bind()** on each one. In the example below, we create a series of sockets, and then bind each socket to an interface that specifies address 0. In doing this, we depend on the "first *available* interface" rule to find the next interface for us. Keep in mind that a successful **bind()** re-writes the contents of the **sockaddr** argument (most importantly, it resets the 0 address component). Thus, we have to re-initialize the structure each time through the loop:

```
/* Declare an array of sockets. we'll create as many as ten. /
#define MAXSOCKETS
int socks[MAXSOCKETS];
int sockN;
int bind_res;

struct sockaddr_in sock_addr;

for (sockN = 0; sockN < MAXSOCKETS; sockN++)
{
    (socks[sockN] = socket(AF_INET, SOCK_STREAM, 0));
    if (socks[sktr] < 0) {
        perror("socket");
```

```
            goto sock_error;
        }

        /* Initialize the structure. */
        sa.sin_family =AF_INET;
        sa.sin_port = htonl(2125);
        sa.sin_addr.s_addr = 0;
        memset(sa.sin_zero,0,sizeof(sa.sin_zero));

        bind_res = bind(socks[sockN],
                        (struct sockaddr *)&sa,
                        sizeof(sa));

        /* A bind error means we've run out of addresses. */
        if (bind_res < 0) {
            closesocket(socks[sockN--]);
            break;
        }
    }

    /* Use the bound socket (listen, accept, recv/send). */
    ...

    sock_error:
        for (;sockN >=0 sockN--)
            closesocket(socks[sockN]);
```

To ask a socket about the address and port to which it is bound you use the **getsockname()** function, described elsewhere.

connect()

int **connect(**int *socket*, struct sockaddr **remote_interface*, int *remote_size***)**

The meaning of the **connect()** function depends on the type of socket that's passed as the first argument:

- If it's a stream client, then **connect()** attempts to form a connection to the socket that's specified by *remote_interface*. The remote socket must be a bound stream listener. A client socket can only be connected to one listener at a time. Note that you can't call **connect()** on a stream listener.
- If it's a datagram socket (either a sender or a receiver), **connect()** simply caches the *remote_interface* information in anticipation of subsequent **send()** and **recv()** calls. By using **connect()**, a datagram avoids the fuss of filling in the remote information that's needed by the "normal" datagram message functions, **sendto()** and **recvfrom()**. Note that a datagram may only call **send()** and **recv()** if it has first called **connect()**.

The *remote_interface* argument is a pointer to a **sockaddr_in** structure cast as a **sockaddr** pointer. The *remote_size* value gives the size of *remote_interface*. See the **bind()** function for a description of the **sockaddr_in** structure.

Currently, you can't disconnect a connected socket. If you want to connect to a different listener, or re-set a datagram's interface information, you have to close the socket and start over.

When you attempt to **connect()** a stream client, the listener must respond with an **accept()** call. Having gone through this dance, the two sockets can then pass messages to each other through complementary **send()** and **recv()** calls. If the listener doesn't respond immediately to a client's attempt to connect, the client's **connect()** call will block. If the listener doesn't respond within (about) a minute, the connection will time out. If the listener's acceptance queue is full, the client will be refused and **connect()** will return immediately.

If **connect()** fails, it returns less-than-zero, and sets **errno** to a descriptive constant:

errno Value	Meaning
EISCONN	The socket is already connected.
ECONNREFUSED	The listener rejected the connection.
ETIMEDOUT	The connection attempt timed out.
ENETUNREACH	The client can't get to the network.
EBADF	The *socket* argument is invalid.
-1	All other errors.

getsockname()

int **getsockname(**int *socket*, struct sockaddr **interface*, int *size***)**

getsockname() returns, by reference in *interface*, a **sockaddr_in** structure that contains the interface information for the bound socket given by *socket*. The **size* argument gives the size of the *interface* structure; **size* is reset, on the way out, to the size of the interface argument as it's passed back. Note that the **sockaddr_in** pointer that you pass as the second argument must be cast as a pointer to a **sockaddr** structure:

```
struct sockaddr_in interface;
int size = sizeof(interface);

/* We'll assume "sock" is a valid socket token. */
if (getsockname(sock, (struct sockaddr*)&interface, &size) < 0)
    /* error */
```

If **getsockname()** fails, the function returns less-than-zero and sets **errno** to one of the following constants:

errno Value	Meaning
EINVAL	The **size* value (going in) wasn't big enough.
EBADF	The *socket* argument is invalid.
-1	All other errors.

listen(), accept()

int **listen(**int *socket*, int *acceptance_count***)**;

int **accept(**int *socket*, struct sockaddr **client_interface*, int **client_size***)**

After you've bound a stream listener socket to an interface (through **bind()**), you then tell the socket to start "listening" for clients that are trying to connect. You then pass the socket to **accept()**; the function blocks until a client connects to the listener (the client does this by calling connect, passing a description of the interface to which the listener is bound).

When **accept()** returns, the value that it returns directly is a new socket token; this socket token represents an "accept" socket that was created as a proxy (on the local machine) for the client. To receive a message from the client, or to send a message to the client, the listener must pass the accept socket to the respective stream messaging functions, **recv()** and **send()**.

A listener only needs to invoke **listen()** once; however, it can accept more than one client at a time. Often, a listener will spawn an "accept" thread that loops over the **accept()** call.

Note that only stream listeners need to invoke **listen()** and **accept()**. None of the other socket types (enumerated in the **socket()** description) need to call these functions.

listen() Closer

int **listen(**int *socket*, int *acceptance_count***)**;

listen() takes two arguments: The first is the socket that you want to have start listening. The second is the length of the listener's "acceptance count." This is the number of clients that the listener is willing to accept at a time. If too many clients try to connect at the same time, the excess clients will be refused—the connection isn't automatically retried later.

After the listener starts listening, it must process the client connections within a certain amount of time, or the connection attempts will time out.

If **listen()** succeeds, the function returns 0; otherwise it returns less-than-zero and sets the global **errno** to a descriptive constant. Currently, the only **errno** value that **listen()** uses, other than -1, is **EBADF**, which means the socket argument is invalid.

accept() Examined

int **accept(**int *socket*, struct sockaddr **client_interface*, int **client_size***)**

The arguments to **accept()** are the socket token of the listener (*socket*), a pointer to a **sockaddr_in** structure cast as a **sockaddr** structure (*client_interface*), and a pointer to an integer that gives the size of the *client_interface* argument (*client_size*).

The *client_interface* structure returns interface information (IP address and port number) of the client that's attempting to connect. See the **bind()** function for an examination of the **sockaddr_in** structure.

The **client_size* argument is reset to give the size of *client_interface* as it's passed back by the function.

The value that **accept()** returns directly is a token that represents the accept socket. After checking the token value (where less-than-zero indicates an error), you must cache the token so you can use it in subsequent **send()** and **recv()** calls.

When you're done talking to the client, remember to call **closesocket()** on the accept socket that **accept()** returned. This frees a slot in the listener's acceptance queue, allowing a possibly frustrated client to connect to the listener.

If **accept()** fails, it returns less-than-zero (as mentioned above) and sets **errno** to one of the following constants:

errno Value	Meaning
EINVAL	The listener socket isn't bound.
EWOULDBLOCK	The acceptance queue is full.
EBADF	The *socket* argument is invalid.
-1	All other errors.

select()

int **select(**int *socket_range*,
struct fd_set **read_bits*,
struct fd_set **write_bits*,
struct fd_set **exception_bits*,
struct timeval **timeout***)*

The **select()** function returns information about selected sockets. The *socket_range* argument tells the function how many sockets to check: It only checks the first (*socket*_range - 1) sockets. You don't have to be exact with this value; typically, you set the argument to 32. Note that a *socket_range* value of 0 *doesn't* select the first socket (which will have a token of 0). You have to pass a value of at least 1.

The **fd_set** structure that types the next three arguments is simply a 32-bit mask that encodes the sockets that you're interested in; this refines the range of sockets that was specified in the first argument. You should use the **FD_***OP***()** macros to manipulate the structures that you pass in:

- **FD_ZERO(***set***)** clears the mask given by *set*.
- **FD_SET(***socket*, *set***)** adds a socket to the mask.
- **FD_CLEAR(***socket*, *set***)** clears a socket from the mask.
- **FD_ISSET(***socket*, *set***)** returns non-zero if the given socket is already in the mask.

The function passes socket information back to you by resetting the three **fd_set** arguments. The arguments themselves represent the types of information that you can check:

- *read_bits* tells you if a socket is "ready to read." In other words, it tells you if a socket has a in-coming message waiting to be read.
- *write_bits* tells you if a socket is "ready to write."
- *exception_bits* tells you if there's an exception pending on the socket.

Note: Currently, only *read_bits* is implemented. You should pass **NULL** as the *write_bits* and *exception_bits* arguments.

select() doesn't return until at least one of the **fd_set**-specified sockets is ready for one of the requested operations. To avoid blocking forever, you can provide a time limit in the final argument, passed as a **timeval** structure.

In the following example function implementation, we check if a given datagram socket has a message waiting to be read. The **select()** times out after two seconds:

```
bool can_read_datagram(int socket)
{
    struct timeval tv;
    struct fd_set fds;
    int n;

    tv.tv_sec = 2;
    tv.tv_usec = 0;

    /* Initialize (clear) the socket mask. */
    FD_ZERO(&fds);

    /* Set the socket in the mask. */
    FD_SET(socket, &fds);
    select(s + 1, &fds, NULL, NULL, &tv);

    /* If the socket is still set, then it's ready to read. */
    return FD_ISSET(socket, &fds);
}
```

If **select()** experiences an error, it returns -1; if the function times out, it returns 0. Otherwise—explicitly, if *any* of the selected sockets was found to be ready—it returns 1.

send(), recv()

int **send(**int *socket*, const char **buf*, int *size*, int *flags***)**
int **recv(**int *socket*, char **buf*, int *size*, int *flags***)**

These functions are used to send data to a remote socket, and to receive data that was sent by a remote socket. **send()** and **recv()** calls must be complementary: After socket A sends

to socket B, socket B needs to call **recv()** to pick up the data that A sent. **send()** sends its data and returns immediately. **recv()** will block until it has some data to return.

The **send()** and **recv()** functions can be called by stream or datagram sockets. However, there are some differences between the way the functions work when used by these two types of socket:

- For a stream listener and a stream client to transmit messages, the listener must have previously called **bind()**, **listen()**, **accept()**, and the client must have called **connect()**. Having been properly connected, the two sockets can send and receive as if they were peers.

  For stream sockets, **send()** and **recv()** can both block: **send()** blocks if the amount of data that's sent overwhelms the receiver's ability to read it, and **recv()** blocks if there's no message waiting to be read. You can tell a **recv()** to be non-blocking by setting the sending socket's no-block socket option (see **setsockopt()**). The no-block option doesn't apply to sending.

- If you want to call **send()** or **recv()** through a datagram socket, you must first **connect()** the socket. In addition, a receiving datagram socket must also be bound to an interface (through **bind()**). See the **connect()** description for more information on what that function means to a datagram socket.

  Datagram sockets never block on **send()**, but they can block in a **recv()** call. As with stream sockets, you can set a datagram socket to be non-blocking (for the **recv()**, as well as for **recvfrom()**) through **setsockopt()**.

The Arguments

The arguments to **send()** and **recv()** are:

- *socket* is, for datagrams and stream client sockets, the local socket token. In other words, when a datagram or stream client wants to send or receive data, it passes its own socket token as the first argument. The recipient of a **send()**, or the sender of a **recv()** is, for these sockets, well-known: Its the socket that's identified by the previous **connect()** call.

  For a stream listener, *socket* is the "accept socket" that was previously returned by an **accept()** call. A stream listener can send and receive data from more than one client at the same time (or, at least, in rapid succession).

- *buf* is a pointer to the data that's being sent, or is used to hold a copy of the data that was received.

- *size* is the allocated size of *buf*, in bytes.

- *flags* is currently unused. For now, set it to 0.

A successful **send()** returns the number of bytes that were send; a successful **recv()** returns the number of bytes that were received. If a **send()** or **recv()** fails, it returns less-than-zero and sets **errno** to a descriptive constant:

errno Value	Meaning
EWOULDBLOCK	The call would block on a non-blocking socket (**recv()** only).
EINTR	The local socket was interrupted.
ECONNRESET	The remote socket disappeared (**send()** only).
ENOTCONN	The socket isn't connected.
EBADF	The *socket* argument is invalid.
EADDRINUSE	The interface specified in the previous connect is busy (datagram sockets only).
-1	All other errors.

sendto(), recvfrom()

int **sendto(**int socket,
char **buf*,
int *size*,
int *flags*,
struct sockaddr **to*,
int *tolen***)**

int **recvfrom(**int socket,
char **buf*,
int *size*,
int *flags*,
struct sockaddr **from*,
int **fromlen***)**

These functions are used by datagram sockets (only) to send and receive messages. The functions encode all the information that's needed to find the recipient or the sender of the desired message, so you don't need to call **connect()** before invoking these functions. However, a datagram socket that wants to receive message must first call **bind()** (in order to fix itself to an interface that can be specified in a remote socket's **sendto()** call).

The four initial arguments to these function are similar to those for **send()** and **recv()**; the additional arguments are the interface specifications:

- For **sendto()**, the *to* argument is a **sockaddr_in** structure pointer (cast as a pointer to a **sockaddr** structure) that specifies the interface of the remote socket that you're sending to. The *tolen* argument is the size of the *to* argument.

- For **recvfrom()**, the *from* argument returns the interface for the remote socket that sent the message that **recvfrom()** received. **fromlen* is set to the size of the *from* structure. As always, the interface structure is a **sockaddr_in** cast as a pointer to a **sockaddr**.

sendto() never blocks. **recvfrom()**, on the other hand, will block until a message arrives, unless you set the socket to be non-blocking through the **setsockopt()** function.

You can "broadcast" a message to all interfaces that can be found by setting **sendto()**'s target address to **INADDR_BROADCAST**.

As an alternative to these functions, you can call **connect()** on a datagram socket and then call **send()** and **recv()**. The **connect()** call caches the interface information provided in its arguments, and uses this information the subsequent **send()** and **recv()** calls to "fake" the analogous **sendto()** and **recvfrom()** invocations. For sending, the implication is obvious: The target of the send() is the interface supplied in the **connect()**. The implication for receiving bears description: When you **connect()** and then call **recv()** on a datagram socket, the socket will only accept messages from the interface given in the **connect()** call.

You can mix **sendto()**/**recvfrom()** calls with **send()**/**recv()**. In other words, connecting a datagram socket doesn't prevent you from calling **sendto()** and **recvfrom()**.

A successful **sendto()** returns the number of bytes that were send; a successful **recvfrom()** returns the number of bytes that were received. If a **sendto()** or **recvfrom()** calls fails, less-than-zero is returned and errno is set to a descriptive constant:

errno Value	Meaning
EWOULDBLOCK	The call would block on a non-blocking socket (**recvfrom()** only).
EINTR	The local socket was interrupted.
EBADF	The *socket* argument is invalid.
EADDRNOTAVAIL	The specified interface is unrecognized.
-1	All other errors.

setsockopt()

int **setsockopt(**int *socket*, int *level*, int *option*, char **data*, unsigned int *size***)**

setsockopt() lets you set certain "options" that are associated with a socket. Currently, the Network Kit only recognizes one option: It lets you declare a socket to be blocking or non-blocking. A blocking socket will block in a **recv()** or **recvfrom()** call if there's no data to retrieve. A non-blocking socket returns immediately, even if it comes back empty-handed.

Note that a socket's blocking state applies *only* to **recv()** and **recvfrom()** calls.

The function's arguments are:

- *socket* is the socket that you're attempting to affect.
- *level* is a constant that indicates where the option is enforced. Currently, *level* should always be **SOL_SOCKET**.
- *option* is a constant that represents the option you're interested in. The only option constant that does anything right now is **SO_NONBLOCK**. (Two other constants—**SO_REUSEADDR** and **SO_DEBUG**—are recognized, but they aren't currently implement.)
- *data* points to a buffer that's used to toggle or otherwise inform the option. For the **SO_NONBLOCK** option (and other boolean options), you fill the buffer with zeroes if you want to turn the option off (the socket will block), and non-zeros if you want to turn it on (the socket won't block). In the case of a boolean option, a single byte of zero/non-zero will do.
- *size* is the size of the *data* buffer.

The function returns 0 if successful; otherwise, it returns less-than-zero and sets errno to a descriptive constant:

errno Value	Meaning
ENOPROTOOPT	Unrecognized *level* or *option* argument.
EBADF	The *socket* argument is invalid.
-1	All other errors.

Keep in mind that attempting to set the **SO_REUSEADDR** or **SO_DEBUG** option won't generate an error, but neither will it do anything.

The Mail Daemon

Declared in: <net/E-Mail.h>

Overview

Every Be machine has a *mail daemon*; this is a local process that's responsible for retrieving mail from and sending mail to a mail server. The mail server that the daemon talks to is a networking application that's either part of your Internet Service Provider's services, or that's running on a local "mail repository" machine. The functions described in this section tell you how to manage the mail daemon's connection with the mail server—how to tell the daemon which mail server to talk to, how to command the daemon to send and retrieve mail, how to automate mail retrieval, and so on.

All the functions that are described here (but one) are promoted to user-land through the E-mail preferences application (the one exception is the **forward_mail()** function). Indeed, the operations that these functions perform are rightly regarded as belonging to the user. The only reason that you would need to call the daemon functions—with the exceptions of **forward_mail()** and, possibly, **check_for_mail()**—is if you want to build your own E-mail preferences application. (**forward_mail()** and **check_for_mail()** could legitimately be worked into a mail-reading or -composing application.)

The architecture of the E-mail message itself isn't discussed here; for such information see "Mail Messages (BMailMessage)" on page 37.

The Mail Daemon and the Mail Server

The mail daemon can talk to two different mail servers:

- The *Post Office Protocol* ("POP") server manages individual mail accounts. When the Be mail daemon wants to retrieve mail that's been sent to a user, it must tell the mail server which POP account it's retrieving mail for.
- The *Simple Mail Transfer Protocol* ("SMTP") server manages mail that's being sent out into the world (and that will, eventually, find its way to a POP server).

The POP and the SMTP servers are identified by their hosts' names (in other words, the names of the machines on which the servers are running). The mail daemon can only talk to one POP and one SMTP server at a time, but can talk to the two of them simultaneously. Typically—nearly exclusively—the POP and SMTP servers reside on the same machine, and so are identified by the same name.

To set the identities of the POP and SMTP mail servers, you fill in the fields of a **mail_account** structure and pass the structure to the **set_mail_account()** function. As the name of the structure implies, **mail_account** encodes more than just the names of the servers' hosts. It also identifies a specific user's POP mail account; the complete definition of the structure is this:

```
typedef struct
{
    char pop_name[B_MAX_USER_NAME_LENGTH];
    char pop_password[B_MAX_USER_NAME_LENGTH];
    char pop_host[B_MAX_HOST_NAME_LENGTH];
    char smtp_host[B_MAX_HOST_NAME_LENGTH];
} mail_account;
```

The POP user information that's stored in the **mail_account** structure (in other words, the **pop_name** and **pop_password** fields) is used only for the POP server; it has no significance for the SMTP server.

Sending and Retrieving Mail

Messages that are retrieved (from the mail server) by the mail daemon are stored in the database, from whence they are plucked and displayed by a mail-reading application (a "mail reader"; Be supplies a simple mail reader called BeMail). Similarly, messages that the user composes (in a mail composition application) and sends are placed in the database until the mail daemon comes along and passes them on to the mail server.

Sending and retrieving mail is the mail daemon's most important function. Both actions (server-to-database and database-to-server transmission) are performed through the **check_for_mail()** function. This is the mail daemon's fundamental "do something" function. All other function either prime the daemon

Other Mail Daemon Features

The other mail structures and functions define the other features that are provided by the mail daemon. These features are:

- *A mail delivery schedule.* The **mail_schedule** structure (passed through the **set_mail_schedule()** function) lets you tell the daemon how often and during which periods (week days only, every day, and so on) it should automatically check for newly arrived mail and send newly composed mail. Technically, the mail schedule tells the daemon how often to invoke **check_for_mail()**.

- *Mail notification.* The **mail_notification** structure (passed through the **set_mail_notification()** function) lets you tell the daemon how you would like it to tap you on the shoulder when it has new mail for you to read. Would you like it to display an alert panel? Squawk at you? Both?

- *A settable mail reader.* The **set_mail_reader()** function lets you identify the application that you would like to use to read in-coming mail. (Be provides a default mail reader/composition program called BeMail.)

- *Mail forwarding*. The **forward_mail()** function lets you re-send in-coming mail to some other account.

All of these features (less mail-forwarding) can also be set by the user through the E-mail preferences panel.

Functions

check_for_mail()

long **check_for_mail(**long **incoming_count***)**

Sends and retrieves mail. More specifically, this functions asks the mail daemon to retrieve in-coming messages from the POP server and send out-going messages to the SMTP server. The number of POP messages that were retrieved is returned, by reference, in the argument. If you don't need to know the in-coming count, you can (and should) pass **NULL** as the *incoming_count* argument; the function is (potentially) much faster if you ignore the count in this manner.

If the mail world is unruffled, the function returns **B_NO_ERROR**; otherwise, it returns one of the following:

- **B_MAIL_NO_DEMON**. The mail demon isn't running.
- **B_MAIL_UNKNOWN_HOST**. The named POP or SMTP mail server can't be found.
- **B_MAIL_ACCESS_ERROR**. The connection to the POP or SMTP mail server failed.
- **B_MAIL_UNKNOWN_USER**. The POP server doesn't recognize the user name.
- **B_MAIL_WRONG_PASSWORD**. The POP server doesn't recognize the password.

In the cases where a name or password is unrecognized (**B_MAIL_UNKNOWN_HOST**, **...UNKNOWN_USER**, and **...WRONG_PASSWORD**), the (mis)information is taken from the **mail_account** structure that was passed to the daemon in the most recent **set_mail_account()** call. Note that the validity of the **mail_account** information isn't checked when you set the structure—it's only checked when you actually attempt to use the information (as, for example, here).

forward_mail()

long **forward_mail(**BRecord **msg*,
char **recipients*,
bool *reset_sender* = TRUE,
bool *queue* = TRUE**)**

Forwards the mail message represented by *msg* to the list of users given by *recipients*. *msg* is a BRecord object that encapsulates a single in-coming mail message. The user account names listed in *recipients* must be separated from each other by whitespace and/or commas; the entire list must be **NULL**-terminated. Both of these entities (the BRecord-as-mail-message, and the recipients list) are further explained in "Mail Messages (BMailMessage)" on page 37.

If *reset_sender* is **TRUE**, the sender of the forwarded message is reset to be the current recipient; otherwise the sender is left as is. For example, if Anton sends a message to Bertrand and Bertrand forwards the message to Camille with *reset_sender* set to **TRUE**, the message that Camille receives will appear to have been sent by Bertrand; if set to **FALSE**, it will appear to have been sent by Anton.

The *queue* argument determines whether the messages is sent now (**TRUE**) or queued for later transmission (**FALSE**). If you send the message now, all other out-going and in-coming mail messages are transmitted as a matter of course (sending now is like calling **check_for_mail()**). If the message is queued, it waits for the daemon to perform its automatic check, or for the next explicit **check_for_mail()** call.

set_mail_account(), get_mail_account()

long **set_mail_account(**mail_account **account*, bool *save* = TRUE**)**
long **get_mail_account(**mail_account **account***)**

set_mail_account() function lets you set the identities of the POP and SMTP mail servers that you want the mail daemon to use, and lets you set the (user-specific) POP account that the daemon should monitor (when it looks for in-coming mail). All this information is set by filling in the fields of the **mail_account** structure which you pass as the first argument to the function. The structure is defined as

```
typedef struct
{
    char pop_name[B_MAX_USER_NAME_LENGTH];
    char pop_password[B_MAX_USER_NAME_LENGTH];
    char pop_host[B_MAX_HOST_NAME_LENGTH];
    char smtp_host[B_MAX_HOST_NAME_LENGTH];
} mail_account;
```

- **pop_name** and **pop_password** are **NULL**-terminated strings (with a maximum length of 32 characters) that identify the user account on the POP server. The account must already exist; you can't create a new POP account simply by filling a **mail_account** structure and passing it through **set_mail_account()**. Creating a POP account is the responsibility of the Internet Service Provider.

- **pop_host** is a **NULL**-terminated string (64 characters, max) that names the machine on which resides the POP server, and **smtp_host** is a similarly constructed string that names the SMTP server's machine. Normally, the servers are run on the same machine. Again, you can't make up a name here; you have to get the host names from the Internet Service Provider.

The *save* argument sets the persistence of the mail account:

- If you save, this account will be used for all subsequent transactions with the mail servers, and also becomes the *default mail account*. In this role, the account information is remembered when you restart your computer (or otherwise kill and restart the mail daemon).
- If you don't save, this account will be used for subsequent transactions, but will be forgotten when you shut down.

You can set the default mail account even if the mail daemon isn't running. Currently, the **set_mail_account()** function always returns **B_NO_ERROR**.

get_mail_account() returns, by reference in its argument, a copy of the mail account information that the daemon is currently set to use. If the daemon isn't running, this function returns the default mail account. In this case, the function returns **B_MAIL_NO_DAEMON**, otherwise it returns **B_NO_ERROR**.

Note that the validity of the **mail_account** that you pass to **set_mail_account()** or that's copied into the **get_mail_account()** argument isn't checked by these functions. The mail account is only checked when you actually attempt to use the information; in other words, when you attempt to send or retrieve mail.

set_mail_notification(), get_mail_notification()

long **set_mail_notification(**mail_notification **notification*, bool *save* = TRUE**)**
long **get_mail_notification(**mail_notification **notification***)**

set_mail_notification() establishes how you would like to be notified when new mail arrives. There are two notification signals: the mail alert panel and the system beep. You encode your preference by setting the fields of the argument **mail_notification** structure:

```
typedef struct
{
    bool alert;
    bool beep;
} mail_notification;
```

The *save* argument, if **TRUE**, registers the notification setting as the default—in other words, the daemon will remember it when you shutdown the computer. This function always returns **B_NO_ERROR.**

get_mail_notification() returns, by reference, a copy of the **mail_notification** structure that's currently being used by the mail daemon. If the daemon isn't running, the function

hands you the default notification setting, and returns (directly) **B_MAIL_NO_DAEMON**; otherwise it returns **B_NO_ERROR**.

set_mail_reader(), get_mail_reader()

long **set_mail_reader(**ulong *reader_sig*, bool *save* = TRUE**)**
long **get_mail_reader(**ulong **reader_sig***reader_sig***)**

set_mail_reader() tells the system which application to launch (or find) to display newly-arrived mail. The application is identified by its signature. The *save* argument, if **TRUE**, registers the reader signature as the default—in other words, the daemon will remember it when you shutdown the computer. This function always returns **B_NO_ERROR**; note that the function doesn't check to make sure that the argument identifies an actual application.

get_mail_reader() returns, by reference, the signature of the application that the mail daemon is currently using (or will next use) to display mail. If the daemon isn't running, the function hands you the default reader, and returns (directly) **B_MAIL_NO_DAEMON**; otherwise it returns **B_NO_ERROR**.

In the absence of any other provision, the mail daemon uses the Be mail reader, BeMail (signature 'MAIL').

set_mail_schedule(), get_mail_schedule()

long **set_mail_schedule(**mail_schedule **schedule*, bool *save* = TRUE**)**
long **get_mail_schedule(**mail_schedule **schedule***schedule***)**

set_mail_schedule() lets you tell the mail daemon during what days and hours it should automatically check for new mail, and how often it should check. You encode this information by filling in the fields of the argument **mail_schedule** structure:

```
typedef struct
{
    long  days;
    long  interval;
    long  start_time;
    long  end_time;
} mail_schedule;
```

- **days** is a constant that encodes the range of days. It can be one of **B_CHECK_DAILY**, **B_CHECK_WEEKDAYS**, or **B_CHECK_NEVER**. The first two should be obvious; setting the **days** field to **B_CHECK_NEVER** turns off the daemon's automatic mail-checking capability (and the other fields of the structure are ignored).

- **start_time** and **end_time** define the range of minutes, within the candidate days, that the daemon checks for mail. For example, if you want the daemon to check for mail only between 8 am and 6 pm, you would set **start_time** to 480 (8 hours * 60 minutes) and **end_time** to 1080 (18 hours * 60 minutes). If **start_time** and **end_time** are the same, then the daemon works around the clock.

- **interval** is the frequency, in minutes, at which the mail daemon checks for mail. For example, setting interval to 15 means that the daemon will automatically check for new mail (and send out any unsent, recently composed messages) every 15 minutes (within the range of minutes of the candidate days, as set in the other fields).

Mail Messages (BMailMessage)

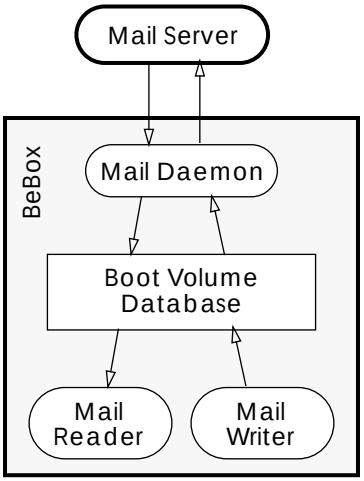
Derived from: public BObject

Declared in: <net/E-mail.h>

Overview

When the mail daemon retrieves new mail from the mail server, it stores the retrieved messages in the boot volume's database, creating a single record (a "mail record") for each message. A mail-reading program can then pull the mail record out of the database (as a BRecord object) and display its contents.

Similarly, when the user composes new mail (on the BeBox) and submits the message for sending, the message-composing application adds the message (again, encapsulated in a mail record) to the database where it waits for the daemon to pick it up and send it to the mail server. The scheme looks something like this:



If you're writing a mail-reading or mail-writing application, then all you really need to know is the definition of the table to which the mail message records conform. With this knowledge, you can retrieve ("fetch") and parse in-coming messages, and create and submit (to the database) out-going messages. The mail message table is called "E-mail", and is described in "The E-Mail Table" on page 43.

The Network Kit also supplies a BMailMessage class that acts as a convenient wrapper around mail records.

The following sections give you a "mail message" tutorial; we'll step through the database and mail message operations that you need to create a generic mail application. If you're already comfortable with database programming (and understand SMTP and POP), you can skip the tutorial and head straight for the "E-mail" and BMailMessage specifications.

Creating a Mail Reader

The design of a Be mail reader should follows this outline:

1. Ask the mail daemon to retrieve mail from the mail server.
2. Get the newly retrieved mail messages from the database.
3. Display the contents of the mail messages.

Throughout the following step-by-step explanations, we'll give both the general approach and also look at what BeMail does.

(Note that this tutorial is incomplete.)

Asking the Daemon to Get New Mail

There are a couple of ways to ask the mail daemon to retrieve newly arrived mail from the mail server:

- You can ask it explicitly by calling **check_for_mail()**
- You can wait for the mail schedule's automatic invocation of **check_for_mail()**.

You probably want to do both of these: You should provide a means for the user to ask that mail be retrieved right now, while also allowing the schedule to do its thing. **check_for_mail()** and **set_mail_schedule()**, which declares the periodicity of automatic mail retrieval, are described in "The Mail Daemon" on page 29. As explained there, the mail schedule "belongs" to the user; its default presentation is through the E-Mail preferences application.

BeMail doesn't actually do anything about retrieving mail. It relies on the mail schedule, and on the mail daemon's "E-Mail Status" alert panel, which provides a "Check Now" button (as in "check for mail now").

Getting Messages from the Database

When new mail arrives, the mail daemon creates a database record to hold each new message, and then commits the records to the database. The table that a mail record conforms to is named "E-Mail". This table is kept in the database that corresponds to the boot volume. As a demonstration of these principles, the following example function counts the number of mail messages that currently reside in the database:

```
#include <Database.h>
#include <Table.h>
#include <Volume.h>
#include <Query.h>

long count_all_email()
{
    BVolume bootVol = boot_volume();
    BDatabase *bootDb = boolVol.Database();
    BTable *emailTable = bootDb->FindTable("E-Mail");
    BQuery *emailQuery = new BQuery();
    long result=0;

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);
        emailQuery->PushOp(B_ALL);
        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();
    }
    delete emailQuery;
    return result;
}
```

Obviously, this example requires some knowledge of how the database works. You can mosey on over to the Storage Kit documentation for the Tolstoy version, or you can read between the lines of the following:

As mentioned above, the mail daemon transforms mail messages into “E-Mail” conforming records, and then “commits” (in database lingo) these records to the boot volume’s database. The first few lines of the example assemble the suspects: The boot volume, the database from the boot volume, and the “E-Mail” table from the boot database. If the table is found, then we construct a “query”—this is the vehicle that will let us retrieve our records. The query is told which table to look in and which records in that table to look for. This is done through **AddTable(**emailTable**)** and the **PushOp(**B_ALL**)** calls; in other words, we tell the query to look for all records in the “E-Mail” table. Then we tell the query to “fetch,” or go out and actually get the records. Technically, it doesn’t actually get records (this would be inefficient); instead, it gets record ID numbers. We count the record ID numbers that it has retrieved (the **CountRecordIDs()** call) and return the count.

The Example Refined—E-Mail Status

For the purposes of a mail reader—in other words, an application that wants to display messages that are received from the mail server—retrieving *all* mail messages isn’t quite right. The “E-Mail” table is used to store both in-coming and out-going messages. So we have to fix our query to only count in-coming messages.

The in-coming/out-going nature of a particular message is stored as a string in the “Status” field of the “E-Mail” table. The mail daemon understands three states: “New”, “Pending”, and “Sent” (a fourth state, “Read”, is used by the BeMail application; we’ll get

to it later). We're only interested in "New" messages, so we change our query accordingly:

```
long count_incoming_email()
{
    /* declarations as above */

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);

        emailQuery->PushField("Status");
        emailQuery->PushString("New");
        emailQuery->PushOp(B_EQ);

        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();
    }
    delete emailQuery;
    return result;
}
```

Here we've replaced the **PushOp(**B_ALL**)** call with a more refined predicate. Again, you can turn to the Storage Kit (the BQuery class, specifically) for the full story on query predicates. Briefly, predicates are expressed in RPN ("Reverse Polish Notation"). According to RPN, the operands of an operation are "pushed" first, followed by the operator. The evaluation of an operation becomes a valid operand for another operation. The series of "pushes" in the example expresses the boolean evaluation

```
(status == "New")
```

In other words, we're going to fetch all records (again, record IDs) that have a "Status" field value of "New".

Let the Browser do the Work

There's one other way to identify mail records: Let the Browser do it. When you "launch" the Browser-defined Mailbox, a query that looks a lot like the one we created above is formed and fetched. The result of the query, the list of found record ID numbers, is turned into a list of BRecord objects that are symbolically listed in the Mailbox window. If the user double-clicks on one of the record icons, the mail daemon passes the record's **record_ref** to the user-defined "mail reader." By default, the mail reader is BeMail. The user can select a different reader (yours) by dropping the reader's icon in the appropriate "icon well" in the E-Mail preferences panel. You can set the reader identity programmatically through the **set_mail_reader()** function, although, as with all E-Mail preferences, it's nicer to let the user make the decision.

A mail reader application needs to be able "catch" the refs that are passed to it. It does this in its implementation of **MessageReceived()**. A simple implementation would look for the message type **B_REFS_RECEIVED**. The rest of this thought is left as an exercise for the mind reader.

Creating BMailMessage Objects

So far, we've determined where we have to go to find in-coming messages, and counted the messages that we found there, but we haven't actually retrieved the messages themselves. Here, we complete our message-retrieving example by fetching record IDs (as before) and then constructing a BRecord for each ID. Having done that, we pass the BRecord to the BMailMessage constructor. In the example, we'll add each BMailMessage to a BList (which is passed in to the function):

```
#include <E-mail.h>
/* and the others */

long get_new_email(BList *list)
{
    BRecord *emailRecord;
    BMailMessage *newMail;
    long count;
    record_id rec_id;

    /* and the others */

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);

        emailQuery->PushField("Status");
        emailQuery->PushString("New");
        emailQuery->PushOp(B_EQ);

        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();

        for (count=0; count < result; count++) {
            rec_id = emailQuery->RecordIdAt(count);
            emailRecord = new BRecord(bootDb, rec_id);
            newMail = new BMailMessage(emailRecord);
            list->AddItem(newMail);
            delete emailRecord;
        }
    delete EmailQuery;
    return result;
}
```

In the **for** loop, we step through the query's "record ID" list, creating a BRecord for each ID. To construct a BRecord from a record ID, you need to pass the appropriate BDatabase object; this is because record ID numbers are only valid within a specific database. Having gotten a BRecord, we pass the object to the BMailMessage constructor; the BMailMessage object copies all the data from the record into itself, such that the BRecord is no longer needed. The BRecord object can (and, unless you have something up your sleeve, should) be deleted after the BMailMessage object is constructed.

When our example function returns, the argument BList will contain all the BMailMessage objects that we constructed, and the function will return the number of messages directly (as before). Note that we should be a bit pickier about checking for

errors; you will, no doubt, correct this oversight in your own mail reader. Also—and here we're just being fussy—keep in mind that by adding the BMailMessages to a BList, we have implied that the BList is now responsible for these objects. More precisely, the entity that called **get_new_mail()** must delete the contents of the list when it's done doing whatever it does.

Displaying the Contents of a Message

The BMailMessage class provides a convenient object cover for mail records. By using BMailMessage objects, you avoid most of the fuss of parsing database records.

When you construct a BMailMessage to represent an in-coming (or otherwise existing) mail message, the contents of the message are copied into the object's "fields." BMailMessage fields are similar to table fields in that they represent named categories of data. The fields that are defined by the BMailMessage class approximate those of the "E-Mail" table; however, you can add new fields that have no complement in the table—adding a field to a BMailMessage object won't extend the "E-Mail" table definition.

The **FindField()** member function retrieves the data that's stored for a particular field within a BMailMessage object. The full protocol goes something like this:

long **FindField(**const char **field_name*, void ***data*, long **length*, long *index*=0**)**

The function works as you would expect: You pass in a field name, and the function points **data* at the contents of that field. The length of the data (in bytes) is returned in *length*. The final argument (*index*) is used to disambiguate between fields that have the same name (the exact value of *index* has no meaning other than ordinal position).

To use **FindField()** properly, you have to know the names of the fields that you can expect to find there. The BMailMessage class defines a number of field names and provides constants to cover them:

Field Name	Constant
"To: "	**B_MAIL_TO**
"Cc: "	**B_MAIL_CC**
"Bcc: "	**B_MAIL_BCC**
"From: "	**B_MAIL_FROM**
"Date: "	**B_MAIL_DATE**
"Reply: "	**B_MAIL_REPLY**
"Subject: "	**B_MAIL_SUBJECT**
"Priority: "	**B_MAIL_PRIORITY**
"Content"	**B_MAIL_CONTENT**

Each of the defined fields stores some number of bytes of "raw" (untyped) data. When you call **FindField()**, the function points the second argument (*data*) to the raw data for the named field, and returns the number of bytes of data in the third argument (*length*).

A particular field (i.e. a field with a particular name) can store more than one entry. The final argument to **FindField()** (the argument named *index*) can be used to distinguish between multiple entries in the same field.

(Here the master died. We'll complete this tutorial and post it on the Be Web site very soon.)

The E-Mail Table

The "E-Mail" database table defines records that hold mail messages. The fields in the table mimic the information that's found in an SMTP or POP mail message header. (See "The Mail Daemon" on page 29 for more information on SMTP and POP). The table's fields are:

- "Status" takes a string that describes the "seen it" state of the message. The mail daemon sets newly arrived in-coming messages to be "New". Out-going messages must have a status of "Pending" (this cues the daemon to send the message). After it has sent a message, the daemon sets the status to "Sent". Beyond these three states, an application is free to invent and use its own—for example, BeMail uses "Read" to mean a message that used to be "New", but which the user has already looked at.
- "Priority" is an integer (a **long**) that rates the message's urgency.
- "From" is a string that names the sender of the message.
- "Subject" is a string that describes the topic of the letter.
- "Reply" is a string that gives the e-mail name to which a response to this message should be sent.
- "When" is a **double** that encodes the date and time at which this message was sent.
- "Enclosures" is an integer count of the number of MIME enclosures that the message contains.
- "header" is the unaltered POP header from a received message; if you're creating mail records yourself (as opposed to using the BMailMessage class), you should construct an SMTP header and add it to this field.
- "content" as a string is the unaltered content of the message.
- "content_file" as a record ID is used if the size of the content threatens to broach the maximum size of a record. In this case, the content is written to a file, and "content_file" gives the ID of that file.
- "enclosures" is a list of attributes (the field itself is typed as raw data) that describe the individual MIME enclosures. There are three attributes per enclosure: a **record_ref** that gives the location of the enclosure, stored as a file; a **NULL-**

terminated string that gives the MIME type, and a **NULL**-terminated MIME subtype string.

- "mail_flags" is a long that encodes the message-is-pending and save-after-sending states of the message. If the message is waiting to go out, the "mail_flags" value is **B_MAIL_PENDING**; if it should be saved after it's sent, then **B_MAIL_SAVE** is added in. After the message is sent, the record is destroyed if "mail_flags" doesn't include **B_MAIL_SAVE**, otherwise the "mail_flags" valued is set to **B_MAIL_SENT**. In all other cases—if the message is in-coming, for example—"mail_flags" is 0.

Constructor and Destructor

BMailMessage()

BMailMessage(void**)**
BMailMessage(BRecord ***record***)**
BMailMessage(BMailMessage ***mail_message***)**

Creates and returns a new BMailMessage object.

The first version creates an empty, "abstract" message: The object doesn't correspond to the second creates an object that acts as a cover for the given BRecord, and the third creates a copy (more or less) of its argument.

~BMailMessage()

virtual **~BMailMessage(**void**)**

Destroys the BMailMessage, even if the object's fields are "dirty." For example, let's say you create a new BMailMessage with the intention of sending a message. You start to edit the object—perhaps you fill in the "To: " field—but then you delete the object. The message that you were composing isn't sent. In other words, the BMailMessage object doesn't try to second-guess your intentions: When you destroy the object, it lies down and dies without whining about it.

Member Functions

CountFields(), GetFieldName(), FindField()

long **CountFields(**char **name* = NULL**)**

long **GetFieldName(**char ***field_name*, long *index***)**

long **FindField(**char **field_name*,
void ***data*,
long **length*,
long *index* = 0**)**

These functions are used to step through and inspect the fields in a BMailMessage object. A field is identified, primarily, by its name. However, a field can have more than one entry, so a secondary identifier (an index) is also necessary. Through the combination of a field name and an index, you can identify and retrieve a specific piece of data. The names of the "standard" mail fields are listed in the **SetField()** description.

CountFields() returns the entry count for the named field. If the name argument is **NULL**, the function returns the number of uniquely named fields in the object. Note that the **NULL** argument version doesn't necessarily return a count of *all* fields. For example, if a BMailMessage contains two **B_MAIL_TO** fields (only), the call

```
CountFields(B_MAIL_TO);
```

will return 2, while the call

```
CountFields();
```

will return 1.

GetFieldName() returns, by reference in the *field_name* argument, the name of the field that occupies the *index*'th place in the object's list of uniquely named fields. If *index* is out-of-bounds, the function returns (directly) **B_BAD_INDEX**; otherwise, it returns **B_NO_ERROR**.

FindField() return the data that lies in the field that's identified by *field_name*. If the object contains more than one entry, you can use the index argument to differentiate them. The data that's found is returned by reference through **data*; the **length* value returns the amount of data (in bytes) that **data* is pointing to. It's not a great idea to alter the pointed-to data, but as long as you don't exceed the existing length you'll probably get away with it.

If *field_name* doesn't identify an existing field (in this object), **B_MAIL_UNKNOWN_FIELD** is returned; if the index is out-of-bounds, **B_BAD_INDEX** is returned. Otherwise, **B_NO_ERROR** is your reward.

See also: **SetField()**

Ref()

record_ref **Ref(**void**)**

Returns the **record_ref** structure that identifies the record that lies behind this BMailMessage object. Not every object corresponds to a record. In general, an in-coming message (a BMailMessages that was constructed from a BRecord object) will have a ref, but an out-going message won't have a ref until the message is actually sent. As always when dealing with refs, you mustn't assume that the ref that's returned here is actually valid—the record may have been removed since the BMailMessage object was constructed (or since the message was sent).

Send()

long **Send(**bool *queue* = TRUE, bool *save* = TRUE**)**

Creates a record for this BMailMessage object, fills in the object's fields as appropriate for an out-going message in SMTP format, and then adds the record to the "E-Mail" table. If *queue* is **TRUE**, the record lies in the database until the mail daemon comes along of its own accord; if *queue* is **FALSE**, the mail daemon is told to send the message (and all other queued messages) right now. The BMailMessage's internal status (as returned by **Status()**) is set **B_MAIL_QUEUED** if queue is **TRUE**.

If save is **TRUE**, the record that holds the message remains in the database after the mail daemon has done its job. Otherwise, the record is destroyed after the message is sent.

The mail record's status is set to "Pending" by this function; when the mail daemon picks up the message, it (the daemon) will destroy the record (if it's not being saved), or change the status to "Sent".

If the BMailMessage doesn't appear to have any recipients, the **Send()** function returns **B_MAIL_NO_RECIPIENT** and the message isn't sent. If *queue* is **FALSE**, the function sends the message and returns the value returned by its (automatic) invocation of **check_for_mail()**. If the message is queued, the function returns **B_NO_ERROR**.

SetField(), RemoveField()

void **SetField(**char **field_name*,
void **data*,
long *length*,
bool *append* = FALSE**)**

long **RemoveField(**char **field_name*, long *index* = 0**)**

These functions add and remove fields (or field entries) from the object.

SetField() adds a field named *field_name*. The *data* and *length* arguments point to and describe the length of the data that you want the field to contain (the length is given in bytes). The final argument, *append*, states whether you want the data to be added (as a

separate entry) to the data that already exists under the same name. If *append* is **FALSE**, the new data (the data that you're passing in this function call) becomes the field's only entry; if it's **TRUE**, and the field already exists, the "old" data isn't clobbered, and the field's "entry count" is increased by one.

RemoveField() removes the data that corresponds to the given field name. If the field contains more than one entry, you can selectively remove a specific entry through the use of the *index* argument. If *field_name* doesn't identify an existing field (in this object), **B_MAIL_UNKNOWN_FIELD** is returned; if the index is out-of-bounds, **B_BAD_INDEX** is returned. Otherwise, **B_NO_ERROR** is returned.

The field names that are defined by the class are:

Field Name	Constant
"To: "	**B_MAIL_TO**
"Cc: "	**B_MAIL_CC**
"Bcc: "	**B_MAIL_BCC**
"From: "	**B_MAIL_FROM**
"Date: "	**B_MAIL_DATE**
"Reply: "	**B_MAIL_REPLY**
"Subject: "	**B_MAIL_SUBJECT**
"Priority: "	**B_MAIL_PRIORITY**
"Content"	**B_MAIL_CONTENT**

See also: **FindField()**

SetEnclosure(), GetEnclosure(), RemoveEnclosure(), CountEnclosures()

void **SetEnclosure(**record_ref **ref*,
const char **mime_type*,
const char **mime_subtype***)**

long **GetEnclosure(**record_ref ***ref*,
char ***mime_type*,
char ***mime_subtype*,
long *index* = 0**)**;

long **RemoveEnclosure(**record_ref **ref***)**

long **CountEnclosures(**void**)**

These functions deal with a BMailMessage's "enclosures." An enclosure is a separate file that's included in the mail message. Enclosures are identified by index only—unlike a BMailMessage's fields, enclosures don't have names. Every enclosure is tagged with a MIME typifier. The MIME typifier is a human-readable string in the form "type/subtype" that attempts to describe the data that the enclosure contains. As shown in the protocol above, the BMailMessage class breaks the two MIME components apart so they can be set (or retrieved) separately.

SetEnclosure() adds an enclosure to the object. The *ref* argument locates the enclosure's data; currently, the refs that you add may only refer to files. The other two arguments let you tag the enclosure with MIME type and subtype strings. (Note that BeMail currently tags all out-going enclosures as "application/befile".)

GetEnclosure() returns, by reference through **ref*, a pointer to the ref that represents the object's *index*'th enclosure. The enclosure's MIME type strings are pointed to by **mime_type* and **mime_subtype*. The MIME strings that the arguments point to are **NULL**-terminated for you. If index is out-of-bounds **B_BAD_INDEX** is returned (this includes the no-enclosure case). Otherwise, **B_NO_ERROR** is returned.

RemoveEnclosure() removes the enclosure that's identified by the argument. If *ref* doesn't identify an existing enclosure, this function returns **B_BAD_INDEX** (look for the error return to change in a subsequent release). Otherwise, it returns **B_NO_ERROR**.

CountEnclosures() returns the number of enclosures that are currently contained in the object.

Status()

long **Status(**void**)**

Every BMailMessage has an internal state (that mustn't be confused with its record's status field) that tells whether the record that represents the object is currently queued to be sent. If it is, the status is **B_MAIL_QUEUED**, otherwise it's **B_MAIL_NOT_QUEUED**.